SAI: A Service Oriented Autonomic IoT Platform

Master's Thesis in Computer Science

Author: An Ngoc Lam
Advisor: Prof. Øystein Haugen

May 30, 2017
Halden, Norway

Abstract

Recent advances in the Internet of Thing (IoT) provide glimpses into the future of smart systems which exploit sensors and devices to provide necessary support intelligently. With the evolution of these IoT systems, the number of devices is also expected to grow considerably; which makes IoT platforms reach the level of complexity where human operational maintenance is getting out of hand. In order to manage such large scale systems, the resources will need to become increasingly "autonomous", capable of managing themselves and cooperating with each other automatically. Such self-managing ability has been already introduced early in 2000s under the concept of Autonomic Computing.

By automating operations such as installation, protection or healing, Autonomic Computing envisions intelligent computing system evolution without the need of human intervention. Although there is a fair amount of research working on conceptual architectures or theoretical designs, Autonomic Computing is still considered as a "hype topic" as very little of it has been fully implemented. Achieving autonomicity is challenging because of the fact that computing systems could change their states in a nondeterministic way. However, as scalability becomes the considerable problem, Autonomic Computing is one of the potential solutions as it is essential to enable large scaled systems with the abilities of runtime monitoring and adaptation to protect themselves from quickly dissolving into non-reliable environment.

In this thesis, we present our solution where we are using Semantic Web Technology as the core components for achieving self-management properties associated with autonomic computing. Firstly, an ontology model driven approach based on the autonomic computing paradigm (MAPE-K) has been proposed as a reference knowledge framework to unify both the managed data and management procedures. Secondly, Semantic Web Rules (SWRL) are used as operating policies to modify system behavior with respect to changes in their operating environment.

This work also provides the implementation of a Service-oriented Autonomic IoT Platform (SAI) as a proof of concept. Given the overall knowledge about the architectures of all systems in the network, the platform keeps updating the status of each entity in the systems and querying through the rules to determine how these systems should evolve. The platform has been validated under two different perspectives: (1) Processing performance of the platform, (2) Cost of system adaptation during runtime. Experiments shows that our approach achieves satisfactory results with regard to scalability. Although we have no substantial test cases for adaptation plans, there is evidence that the deployed applications also meet the requirements for IoT applications in term of performance metrics.

Acknowledgments

My warmest words of gratitude go to my academic supervisor, Professor Øystein Haugen, for the patient guidance, encouragement and advice he has provided throughout my time as his student. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly, without him this work would never have been completed.

I would also like to say thanks to all of my dear friends for your understanding and interest, for helping me to enjoy my life-besides-work, and for your patience and support during the time I am living in Norway.

Finally, I would like to thank Østfold University College, not only for providing the financial support which allowed me to undertake my master study, but also for giving me the opportunity to attend conferences and meet so many interesting people.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In recent years, the convincing forward steps in the development of Internet of Things (IoT) has kindled the possibility of a lot of applications. One of the major challenges in the realization of IoT applications is interoperability among various IoT entities [43], especially with systems which use numerous, heterogeneous sensors and actuators. As these systems evolve and mature, they are also expected to grow in size and complexity. Thus, the need for a new architecture - comprising of smart control and actuation - has been identified by many researchers. This architecture could not only address heterogeneity of IoT devices and enable seamless addition of new devices across applications, but also support sophisticated adaptation scenarios, such as modifying the actual structure or behavior of the applications at runtime [43, 19]. As an example, consider a situation when several applications are using the platform's notification service (e.g, alarm system). At some point this service crashes, the system should switch to an alternative service (e.g, light system) automatically and transparently to the users. Such self-management ability was introduced by IBM under the term *autonomic computing* [30] in 2001. Autonomic computing aims at an analogous goal today, seeking to improve complex computing systems by decreasing human intervention to a minimum [19]. Even though much effort has been put into the development of self-management mechanisms for IoT, it is as yet immature.

In this thesis, we propose an approach to develop autonomic behaviors for IoT platform by applying the IBM's MAPE-K (see Figure 2.8) reference model. Our approach makes intelligent use of existing solution strategies and products (such as Semantic Web technology stack, IoT Ontologies and SWRL rules) to create a general purpose framework from physical integration to application development.

1.1 Motivation

The main challenge in IoT is to manage and maintain a large number of devices and respond to the generated events in a smart way. In connection with the evolution of IoT infrastructure, increasing the number of sensors and actuators makes the system more intelligent and highly responsive. Higher amount of sensed information and precise control could help in achieving sophistication. Furthermore, there must be many devices (to substitute) to make services fault tolerant and dependable. However, it also increases the difficulties of maintaining and controlling the devices as well as managing enormous amount of generated data. As envisaged in [43], we enlist some of the key factors that dictate the challenges in IoT related research:

- *Heterogeneity*: IoT devices are deployed by different persons/authorities/entities. They also have different communication protocols, functionalities, resolutions, etc. Thus, enabling seamless integration of these devices is a huge challenge. The degree of complexity increases when some of these simple devices are connected to form a complex network.

- *Scalability*: The rapid growth of embedded technologies is leading to enormous deployment of miniaturized devices (sensors, actuators, etc.). As the number of devices increases, the data generated by them also become massive. Thus, handling the growth of number of devices is a big challenge in IoT.

- *Interoperability*: there are many entities comprising of human and non-human objects in an IoT application. Seamless interaction amongst the various entities is crucial to envisage the vision of IoT. The interaction amongst different objects magnifies, especially when each actor is managed differently.

- *Security & Privacy*: Due to the large number of actors involved in IoT, ensuring data authentication, data access control, data consistency and protection of data are a few core issues. To evolve a holistic system design, information security, privacy and data protection need to be addressed properly.

The existing limitations in IoT platforms make it necessary to involve developers in the application life cycle after it has been deployed. That is, they have to monitor the application at run-time, detect any critical situation, propose and implement corresponding adaptation by rewriting, recompiling and redeploying the application [19]. Therefore, as the systems grow into large scale, it is necessary for IoT applications to dynamically adapt their behaviors at run-time in response to the changes in the surrounding physical environment as well as the supporting computing infrastructure with minimal human maintenance. Self-adaptability is emerging as a necessary underlying capability, especially for such highly dynamic systems [27] as these systems have reached the level of complexity where human effort required for deployment and maintenance is getting out of hand [27]. With the increasingly number of digital services added to our surroundings, simplicity is highly required by users and operators [15, 28].

Autonomic Computing envisions computing environments which evolve without the need for human intervention [27]. Inspired by biology, autonomic computing has evolved as a discipline to create software systems and applications that are capable of self-management - a key feature of complex and dynamic computing system [19]. Although there is a fair amount of autonomic computing research working on architectures and theoretical design, very little of it has been fully implemented. There are also fewer work focusing on developing an autonomic IoT platform, thus there is great challenge to create autonomic behaviors in IoT context.

The autonomic behavior of a computer system is typically achieved through implementing closed control loops which iteratively observe the systems surrounding environment and context, and react accordingly [19]. The Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) model from IBM is one the fundamental reference model for such closed adaptation loop [19]. It consists of four steps - Monitoring, Analyze, Plan, Execute - and a unifying Knowledge component which conceptually represents all the information needed to perform these four activities.

In this regards, real-time data monitoring and analysis in the IoT context could be seen as a multi-faceted challenge as they involves following aspects [18]:

- *Volume*: the sensed data could become extremely large as the system evolves.
- *Velocity*: requiring on-the-fly data processing as the sensors produce data continuously.
- *Variety*: the data is generated from different heterogeneous resources and in various forms and formats.
- *Veracity*: the data is often uncertain, flawed and rapidly changing.

In addition, Planing and Execution also deal with the same problem as these processes need to respond to the real-time changes not only within an interval that is acceptable in a particular IoT context but also with the cost satisfying the physical limitation of the deploying hardwares. Finally, a unified Knowledge representation which has good coverage of the IoT aspects also plays an important role in the development of an autonomic IoT platform.

Taking all these challenges into consideration, we presented our approach which makes intelligent uses and improvements of the existing solutions and products which deal with the same situation to develop an autonomic solution for IoT platform towards the aforementioned IoT problems (i.e., heterogeneity, scalability, interoperability, etc.).

1.2 Towards the solution: the SAI platform

Following the MAPE-K model for implementing closed adaptation loops, we develop a **S**ervice-oriented **A**utonomic **I**oT (SAI) platform. One of the proposed approach which applied the MAPE-K model to support self-governance in service-based cloud platform is the EXCLAIM framework designed by Dautov [18]. This framework exploited existing technology in Semantic Sensor Web (SSN) and Big Data Processing to overcome the problem of monitoring and analyzing great scale of data in cloud computing. In order words, it aims at creating a cloud monitoring and analysis framework, which is very similar to our goal - achieving a self-management large-scale IoT platform. In fact, in our work, we also apply the techniques from SSN and corresponding stream processing engines to address the challenges of monitoring and analyzing.

While the problems associated with timely processing of sensor data have been relatively successfully tackled by the advances in networking and hardware technologies [18], the problem of carrying out the autonomic adaptation is still pressing. To address these challenges, , from the perspective of IoT application development, we propose using complex rules to model the applications not only to improve the flexibility in modifying the application but also to simplify the process of analyzing the applications and updating knowledge base.

One important aspect of IoT infrastructure is overcoming the heterogeneity of various hardware and software vendors. However, this lack of a unified data representation has been addressed in the context of Semantic Sensor Web (SSW) - a promising combination of two research areas the Semantic Web and the Sensor Web [44]. Therefore, Web Ontology Language (OWL) from the Semantic Web technology stack could be used to represent data in a uniform and homogeneous manner in order to facilitate representing meaningful

hardware description and situation awareness [18]. Fortunately, in the IoT domain, there are also existing work on designing ontologies to support semantic interoperability which can be extended for our purpose.

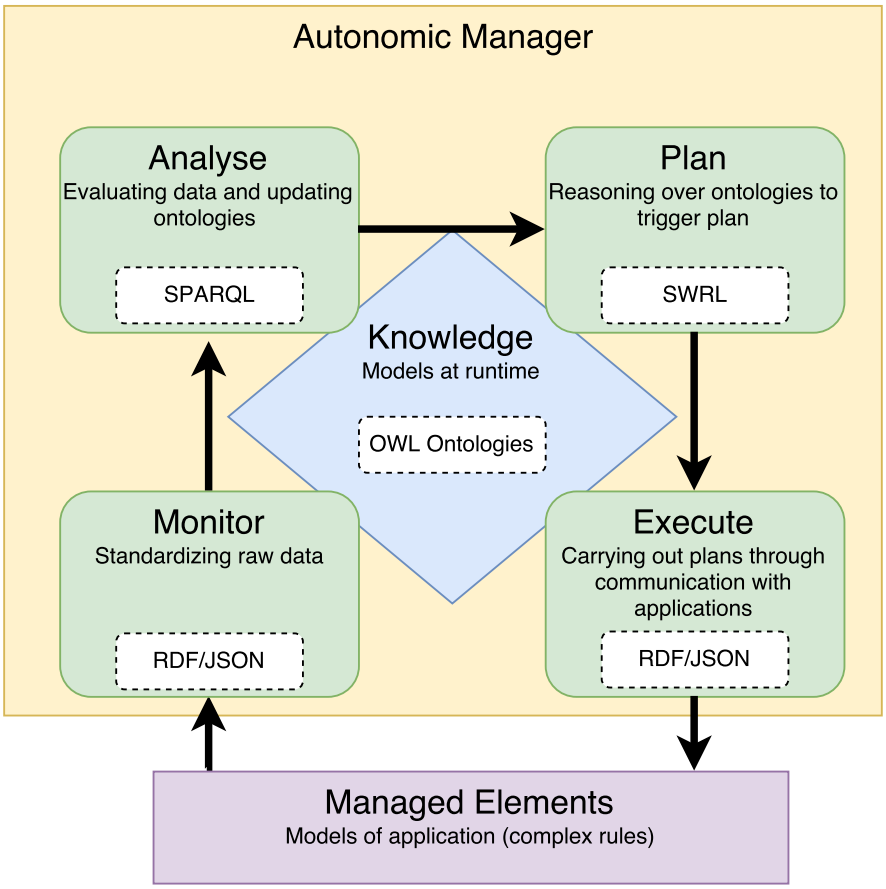

Figure 1.1: Overview of the autonomic manager of SAI platform.

Figure 1.1 illustrates an overview of the functionalities and technologies used in each autonomic component in our solution. Following the presented idea, we created a proof-of-concept prototype and validated it with a Smart Home Case Study. When implementing the prototype, we reused existing technologies from the SSW and IoT domain:

- Web Ontology Language (OWL) to develop a common knowledge model of managed elements (IoT applications).
- Resource Description Framework (RDF) and JavaScript Object Notation (JSON) for representing the semantic data stream as well as SPARQL Protocol and RDF Query Language (SPARQL) for RDF stream processing engine.
- Semantic Web Rule Language (SWRL) for detecting critical situations and planning adaptations.

The experiments show that we are able to successfully detect critical situations and reconfigure the application with acceptable response time. In addition, the results also indicate noticeable efficiency in physical source consumption at the application level.

1.3 Problem Statement

With the presented idea, we aim to address the outlined challenges toward the direction of achieving self-management in IoT system with regard to large amount of heterogeneous entities. In particular, the problem that this thesis addresses can be stated by means of the following problem statements:

- **Research Question 1.** *How to design an autonomic IoT platform addressing the system evolution challenges?* Answering this question requires thorough understanding of architecture of the existing IoT platforms as well as identification of the main supporting features of those platforms toward aforementioned IoT problems (e.g. heterogeneity, interoperability, scalability). It also refers to applying existing IoT infrastructures along with the state-of-the-art autonomic approach into IoT platform development.

- **Research Question 2.** *How to model the internal architecture of IoT applications in order to facilitate adaptation process?* This question involves modeling IoT applications to support building ontology and autonomic management with regard to the challenges of IoT development.

- **Research Question 3.** *How to realize the proposed design for autonomic computing into executable implementation satisfying reasonable IoT performance requirement?* This refers to designing and implementing the framework following the established software engineering practices, including support for scalability, processing performance under the IoT context.

In order to answer these questions, we also present the implementation of our proposed layered IoT platform, called SAI (**S**ervice-oriented **A**utonomic **I**oT platform). We believe that it has potential to tackle many technical challenges described earlier. It also supports the desired characteristics of IoT objects and applications. With the help of a composite case study, we showcase the feasibility of the proposed approach, especially for IoT applications. We believe that the research related to the above questions can contribute to push both researchers and practitioners toward a sound and seamless engineering support for autonomic computing.

1.4 Contribution

The main contribution of this thesis has been developed to answer the research questions presented above. In summary, the main contributions of our research work are:

1. **A software engineering approach for achieving autonomic computing in IoT platform** which combines the MAPE-K reference model and the existing Semantic Sensor Web Technology. This approach exploits the concept of interpreting each IoT entity (e.g., applications, devices, etc.) as a logical sensor and the autonomic manager as distributed network of such sensors, and allowing individual application sensors to be equipped with respective self-governance knowledge (e.g, self-diagnosis, self-adaptation policies, etc.) to enable decoupled, modular and distributed organization of the knowledge base.

On one hand, once deployed properly, all the entities could operate and communicate with each other independently from the autonomic manager. On the other hand, the autonomic manager keeps updating the knowledge base (by integrating new entities or updating the entities status) and suggesting reorganization plan if there are any. Thus, minimizing the amount of unnecessary communication while addressing the heterogeneity of IoT entities.

2. **An extension of IoT ontology and application modeling** for autonomic management. We extended the existing IoT ontologies with the concept of application and rules to support interoperability and autonomic management at semantic level. Furthermore, we suggest modeling application as a set of rules in order to facilitate the process of updating the knowledge base and enhance usability.

3. **The SAI platform** from physical device abstraction to application modeling. As explained in further detail, SAI platform automatically analyzes the application models, updates the knowledge base and determines how an application should be reconfigured with respect to the operating environment changes, and then it modifies the application accordingly. The performance of the whole system has been validated under the perspectives of IoT platform development from two sides:

   - **The SAI platform**. Since IoT systems involves substantial amount of entities which could be physical devices, applications or even operators, the platform should be able to perform at the same efficiency requirements for all entities. We have evaluated the platform from the point of view of latency of the generated reconfiguration.
   - **The SAI Application**. IoT applications might be deployed on hardwares which have limited physical resources while requiring relatively low processing latency. Therefore, we also evaluated the resource consumption of applications deployed on our platform as well as the cost of reconfiguration in term of response time.

1.5 Outline

This thesis consists of 7 chapters as follows:

Chapter 2: *Background.* This Chapter presents the main concepts and characteristics of the approaches related with this thesis. Firstly, it explains the main purpose of the presented research effort - addressing the challenges in IoT research. The chapter starts with introducing some well-known IoT platform as well as discussing their solutions toward the presented challenges. With the evolution of IoT systems, these platform requires the capabilities for self-governance. Accordingly, the next section of this chapter explains the main principles of autonomic computing which is also our approach to support self-governance in IoT platforms. The chapter also introduces the MAPE-K reference model for creating autonomic systems, which will act as the main reference model to our solution.

Chapter 3: *Related Work.* This chapter shows an analysis of the current state of the art approaches to support self-governance. In particular, we present the most important research that we found relevant to the IoT domain as well as our aim. The chapter concludes with several observations and identified research gaps to be addressed by our own research work.

Chapter 4: *SAI platform: A Service-oriented Autonomic IoT Platform.* This chapter firstly introduces conceptual design of our proposed SAI platform. It covers the main building blocks of the platform as well as the process of executing autonomic capabilities of the autonomic manager. The second part of this chapter provides lower level implementation details of the platform. It also brief the reader on the ontologies which was extended in order to model the managed elements.

Chapter 5: *The Smart Home Case Study.* This chapter presents the case study of a Smart Home, which reconfigured its application according to the changes in the physical environment. This case study is intended to demonstrated how the platform would function in order to support autonomic behaviors.

Chapter 6: *Evaluation and Discussion.* This chapter presents approach to evaluate our platform with respect to its physical performance. The chapter concludes with the summary and discussion on the research questions raised in the introduction.

Chapter 7: *Conclusion.* The chapter summaries the whole thesis with an overview of main research contributions as well as potential benefits and shortcomings associated with our approach. It also outlines several directions as well as improvements for future work.

Chapter 2

Background

The background in our case consists of the approaches that are related to the objective of this work: to achieve autonomic computing in IoT platform in order to address IoT development problems. Therefore, this chapter presents the main concepts and characteristic of the approach in order to provide a basic background for understanding the overall thesis work. Specifically, the chapter starts with exploring the various well-known IoT platforms as well as their approach toward the challenges in IoT development. By listing and explaining the main characteristics of these platforms, we also bring to the reader's attention the existing challenge of insufficient capabilities for self-governance, caused by the ever-growing complexity and dynamicity of IoT systems. This challenge can be addressed by applying the principle of Autonomic Computing, whose goal is to reduce the role of human administrators in run-time operation of complex computing systems, which is presented in the second part of this chapter. We also explain in detail the fundamental reference model for implementing autonomic systems, known as MAPE-K, as a baseline for our work. Finally, the chapter concludes with an overview of autonomic features currently present in IoT platforms - this is expected to demonstrate the need for more intensive research efforts to be put into this area.

2.1 Toward the challenges in IoT

In recent years, the Internet of Thing (IoT), a global network of connected devices having identities and virtual personalities operating in smart spaces and using intelligent interfaces to communicate within social, environmental, and user contexts [13], has attracted considerable research attention. This vision of IoT enables the future of a smart world which will comprise of trillions of everyday objects and surrounding environments connected and managed through a range of communication networks and cloud-based servers [8]. However, the heterogeneity of such communications is challenged by the lack of a shared infrastructure and common standards for the IoT [13]. Therefore, it is necessary to have a powerful architecture that offers easy integration, control, communications and useful applications [46]. The ultimate goal is to facilitate and enable the anticipated ubiquitous communications between things with minimal human interventions [25]. To address this problem, many research projects have focused on developing communication standards as well as proposing new software architectures in which Service oriented Architecture (SoA) is a dominant solution as it ensures the interoperability among the heterogeneous devices in multiple ways [36]. Figure 2.1 provides a generic SoA consisting of four layers with

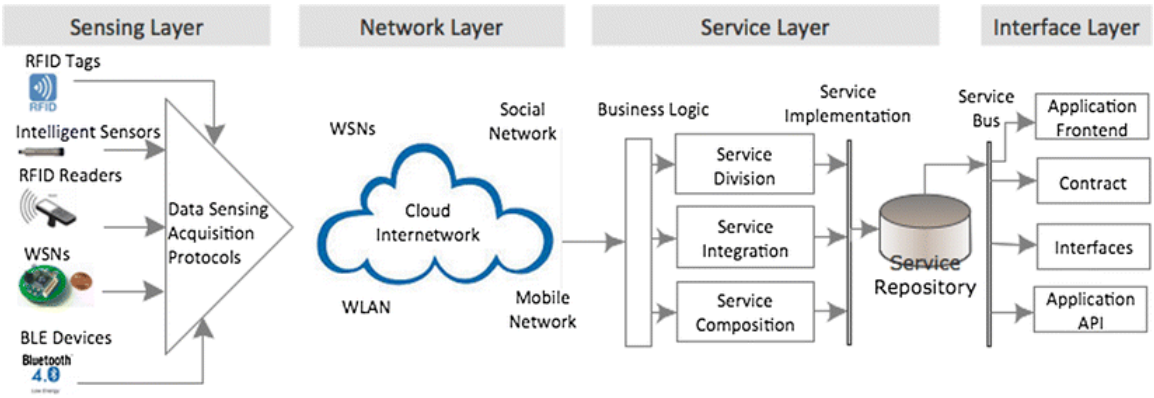

Figure 2.1: Service-oriented architecture for IoT (Adapted from [36]).

distinguished functionalities.

There have been a number of proposed architectures which are developed based on SoA. The following section introduces the most common IoT platforms as well as discusses their proposed solutions towards IoT issues:

1. **CitySense** [38]
   To support the development of wireless system in smart-city, in 2008, Harvard University carried out the CitySense project which deployed about 100 Linuxbased embedded personal computers outfitted with dual 802.11a/b/g radios and various sensors, mounted on buildings and streetlights. By providing an open infrastructure, CitySense can be readily customized to support a wide range of applications. Users can reprogram and monitor CitySense nodes via the Internet, allowing diverse research groups to leverage the infrastructure remotely. However, this is an ad-hoc solution which lacks supports for automatic, end-to-end integration that incorporates devices of all types and scales [19].

2. **Thingworx** [6]
   This is a commercial platform developed by PTC [1] which provides capabilities to create IoT applications through cloud services. The ThingWorx Ready Program [6] allows device vendors to integrate their products with the ThingWorx rapid application development platform in order to reduce the time, cost, and risk required to build innovative Machine-to-Machine (M2M) and IoT applications. Device heterogeneity is also supported by the platform but assuming that the smart objects are capable of communication with the platform over Hypertext Transfer Protocol (HTTP).

3. **SensorCloud** [4]
   SensorCloud is "an infrastructure that allows truly pervasive computation using sensors as an interface between physical and cyber worlds, the data-compute clusters as the cyber backbone and the internet as the communication medium". SensorCloud provides the users abilities to gather, access, process, visualize, analyze, store, share, and search for a large number of sensor data from several types of applications and by using the computational IT and storage resources of the cloud [9]. By virtualizing

[1] `https://www.ptc.com`

the physical sensors, SensorCloud allows the users not to worry about the status of their connected physical sensors (i.e., whether a fault free or not), thus improving the flexibility in application development and addressing problem when increasing size of the sensor network. However, the infrastructure supports auto-integration solution to only limited types of devices and lacks support for larger and heterogeneous types of sensors and devices.

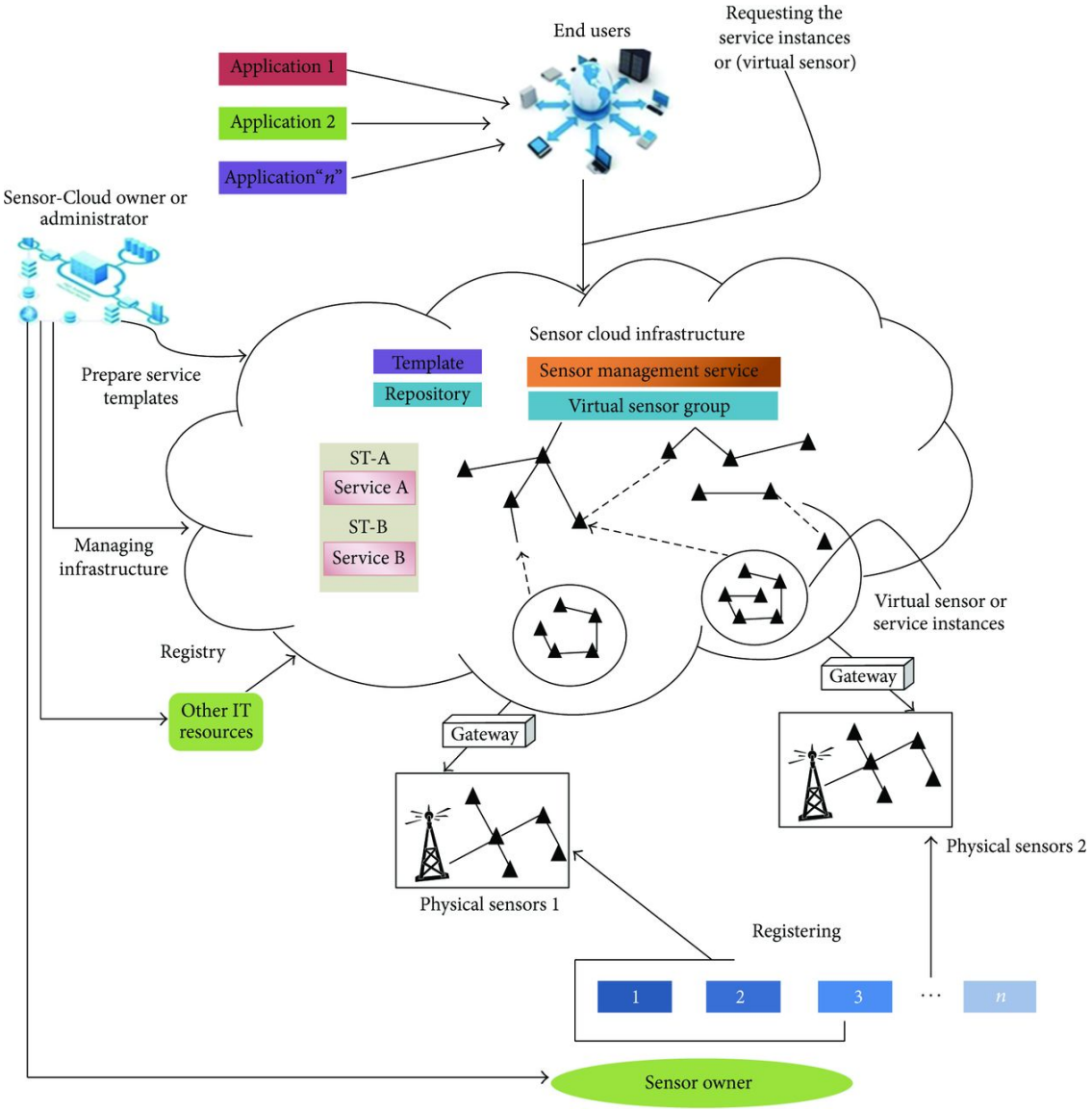

Figure 2.2: SensorCloud Architecture (Adapted from [9]).

4. **Everyware** [1]
   Everyware is a cloud architecture that supports fast development of IoT applications which can be scaled up to thousands of devices. The solutions are a combination of hardware, firmware, operating systems, programming frameworks and external infrastructure that enable customers focus on their core activities. Similar to Sensor-Cloud, Everyware address the autonomic integration problem by leveraging gateway

nodes to mediate the smart objects and the platforms. However, it supports only limited devices which use their M2M technology.

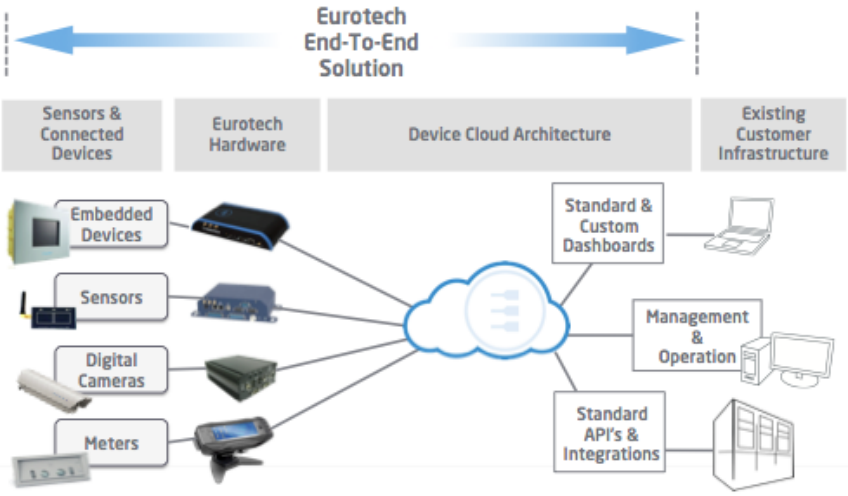

Figure 2.3: Everyware Architecture (Adapted from [1]).

5. **ThingSquare** [5]
   ThingSquare provides both firmware and software platform to support development of IoT applications. Similar to SensorCloud, the platform support autonomic integration with very limited devices. Autonomy is also introduced in the firmware platform which automatically creates a self-healing wireless network that automatically detects and heals wireless problems.

Figure 2.4: ThingSquare Architecture (Adapted from [5]).

6. **Service-Oriented Cross-layer Infrastructure for Distributed smart Embedded devices (SOCRADES)** [45]
   SOCRADES is another IOT architecture that supports integrating devices which use different standards for data and communication through the service-oriented approach. This is part of the European research project *Service-Oriented Cross-layer infRAstructure for Distributed smart Embedded devices SOCRADES*. SOCRADES

implements a device profile for Web services (DPWS) plug-in that can dynamically discover and integrate any DPWS device. DPWS is fully aligned with Web services technology and includes numerous extension points allowing for seamless integration of device-provided services (which is written in the device abstraction layer) in enterprise-wide application scenarios.

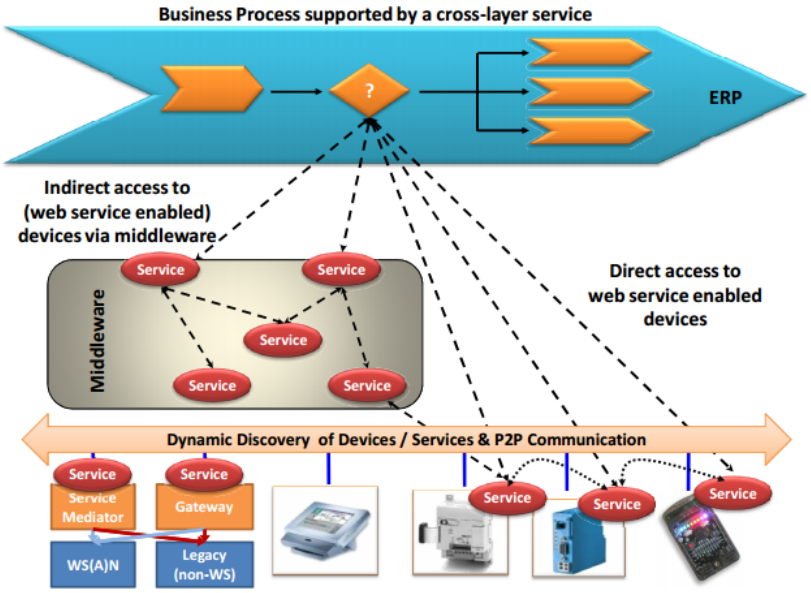

Figure 2.5: SOCRADES Architecture (Adapted from [45]).

7. **Xively** [7]
   Similar to other SOA architectures that provides a Platform as a Service built for the IoT, Xively includes directory services, data services, a trust engine for security and web-based management application. The platform also address scaling problem, however, supports only HTTP devices.

8. **EvryThng** [2]
   EvryThng offers an IoT Platform as a Service that connects consumer products to the Web and manages real-time data to drive applications. The platform concentrates on enterprise services which requires end-to-end security and performance at massive scale. It supports very limited devices which communicate with the platform over HTTP.

9. **IFTTT** [3]
   IFTTT is an abbreviation of "If This Then That" which is a web-based service that allows users to create chains of simple conditional statements, called "recipes", which are triggered based on changes to other web services. Thus, it does not provide any programmability but rule-based services. Auto integration is also supported by the platform but only with HTTP devices.

10. **Cloud-Edge-Beneath (CEB)** [49]
    CEB is a SoA based architecture provided by the Mobile and Pervasive Computing

Laboratory in University of Florida. The framework allows for automatic integration of heterogeneous devices by using an Atlas Device Integration Platform [34] as a universal software/hardware adapter to connect a variety of sensor and other devices into a smart space.

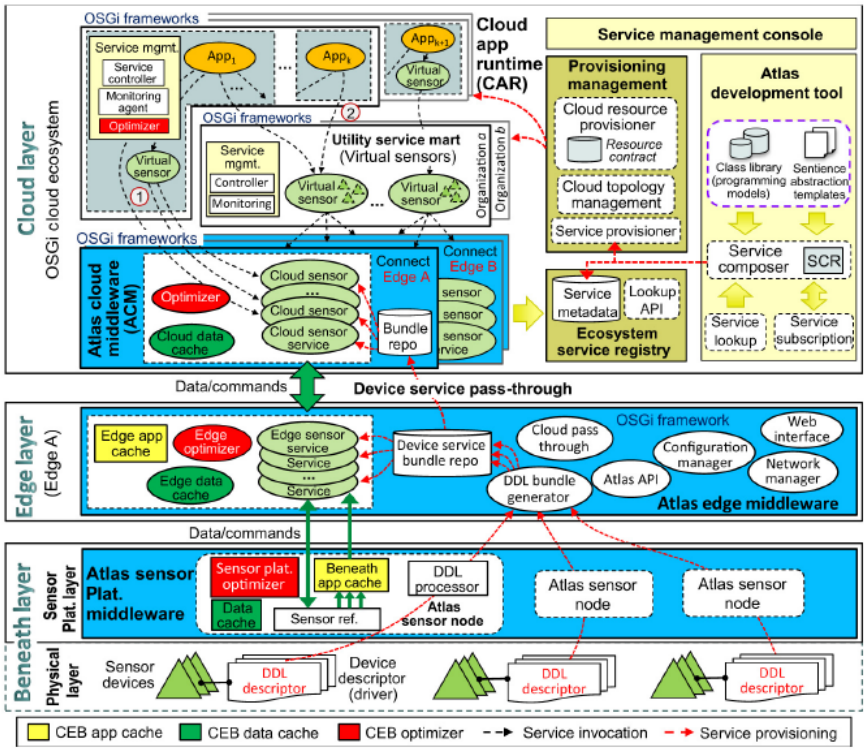

Figure 2.6: CEB Architecture (Adapted from [49]).

11. **Arrowhead** [48]
    Arrowhead is an IoT framework built on the fundamental principles of SOA which address the interoperability issue by abstracting any IoTs to services, thus, improving flexibility and scalability. The creation of automation is based on the idea of local automation clouds which include the devices and systems required to perform the desired automation tasks (see Figure 2.7). A local Arrowhead Framework cloud can be compared to global cloud providing real-time data handling, system security, scalability and automation support. Automation support is achieved through the presence of three core services: Orchestration system stating which consumers could exchange services with which producers, Service Registry System keeping track of all services, Authorization system responsible for devices authentication and authorization. At the moment of conducting this survey, Arrowhead framework only supports devices and systems which follow its authentication solutions.

The above list is not complete as the number of IoT platforms which are either commercial or open-source is increasing with the advent of IoT applications. However, it provides an insight of current progress of IoT technology as well as motivation for developing research idea in this area. In summary, most of the above platforms separate the device

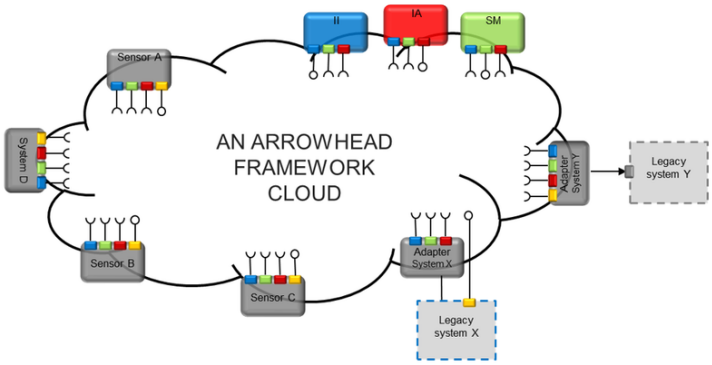

Figure 2.7: Arrowhead Local Cloud Architecture (Adapted from [48]).

integration from service/application development and support auto integration, which enable autonomic device integration stemming from the devices and requiring no engineering specialties. Among these solutions, SensorCloud, ThingSquare and IFTTT provide a closed infrastructure that supports auto-integration solutions to only limited types of devices and lacks support for larger and heterogeneous types of sensors and devices. On the contrary, other platforms which support device heterogeneity either assume that the smart objects are capable of communicating with the platform over HTTP or leverage gateway nodes to mediate the smart objects and the platforms. Some advanced platforms address the scalability and interoperability by virtualizing the physical devices, providing accesses to these devices through services, communication between devices is auto managed internally by the cloud platforms. Autonomic computing is also introduced in some platforms under the level of self-healing or self-configured, however, these solutions are still ad-hoc or immature.

2.2 Autonomic Computing

In October 2001, IBM released a manifesto [30] describing the vision of Autonomic Computing as a method to countermeasure the complexity of software system by making them self-managing, however, probably making the systems even more complex. The complexity is argued to be embedded in the system infrastructure which in turn can be automated.

Autonomic computing is a concept that "brings together many fields of computing with the purpose of creating systems that self-manage"[35]. Inspired by the biological concept of autonomic systems, IBM introduced the term autonomic computing to refer to bodily task which function unconsciously [30]. IBM compared complex computing systems to the human body, and suggested that such systems should also demonstrate certain autonomic properties such as independently taking care of regular maintenance and optimization tasks, thus reducing the workload on system administrators [19]. As stated by Alan Ganek who is on behalf of Autonomic Computing Research in IBM:

> "Autonomic computing is the ability of systems to be more self-managing. The term autonomic comes from the autonomic nervous system, which controls many organs and muscles in the human body. Usually, we are unaware of its workings because it functions in an involuntary, reflexive manner – for example,

> we do not notice when our heart beats faster or our blood vessels change size in response to temperature, posture, food intake, stressful experiences and other changes to which we're exposed. And, by the way, our autonomic nervous system is always working"

An autonomic computing system must be able to configure and reconfigure itself under varying and even unpredictable conditions [11]. System configuration must occur automatically and dynamic adjustments must be made according to that configuration in order to best handle changing environments. Depending on the extent to which modern computing systems support self-management, we can distinguish 5 levels of autonomicity: basic, managed, predictive, adaptive, fully autonomic level [29]. They are explained as follows:

- **Basic Level:** At this level, each system components is managed by IT professionals manually. Their generated data is manually collected, analyzed and transformed into adaptation actions by IT professionals. Additionally, all the management tasks, such as system configuration, optimization, healing, protection, are performed manually. These challenges require the IT staff to be highly experienced and skilled.
- **Managed Level:** At this level, system monitoring is applied. The data from different system components are automatically collected in on place, providing human operators with a more holistic view on the current state of the whole system. Most analysis is done by IT professionals, but it is starting point of automation of IT tasks.
- **Predictive Level:** At this level, individual components have the ability to monitor themselves, analyze and asses situation and context, and offer adaptation plan. Therefore, dependency on persons is reduced and decision making is improved.
- **Adaptive Level:** At this level, the system is equipped with necessary abilities of detecting and diagnosing potential critical situations and take actions accordingly in order to perform self-adaptations.
- **Autonomic Level:** At this level, system operations are solely managed by business policies established by the administrator at design time. The IT staff is only required whenever there is a change in the self-management policies.

According to Paul Horn from IBM, who first suggested the systematic scientific approach of autonomic computing, the four fundamental properties of automatic systems are [30]:

- **Self-configuration**: the system configures itself according to high-level goals. This means it is able to install and set up based on the needs of the platform and the user.
- **Self-optimization**: the system is able to optimize the use of resources. It could decide to initiate changes in order to improve performance or quality of the services.
- **Self-healing**: the system detects and diagnoses a variety of problems automatically or even attempt to fix the problem if possible.
- **Self-protection**: the system protects itself from malicious attacks or inadvertent changes from the users. It could also anticipate security breaches and prevent them from occurring in the first place.

One of the possible ways of achieving these 4 characteristics is through self-reflection. A self-reflective system uses causally connected self-representation to support the inspection and adaptation of that system [16]. Such system is context-aware and self-aware of its internal structure and able to perform run-time adaptations, so that applied adaptations dynamically reflect on the state of the system (thus, possibly, triggering another adaptation cycle) [20]. The motivation behind self-reflection stems from the necessity to have systems which are capable of reacting to various changes in the environment dynamically, based on the provided knowledge. In such scenarios, the capability of a remote system to perform automatic adaptations at run-time within a specific time frame is often of a great importance [19]. The concept of self-reflective system can be summarized in the following characteristics [40]:

- **Self-awareness** or Self-Knowledge of an autonomic system is the ability of understanding its internal structure, relationships between sub-component, available resources, etc.
- **Context Awareness** is the ability to observe the execution environment as well as interpret these observations.
- **Openness** refers to the ability of an autonomic system to operate in a heterogeneous environment and must be portable across multiple platforms.
- **Behavior Anticipatory** is the ability of anticipating the optimal required resources in order to meet emerging users requirements while hiding from them the complexity.

Achieving these four characteristics for a self-reflective system requires some sort of a self-representation knowledge repository, which would provide all the necessary information to enable self-reflection. To facilitate self-awareness and context-awareness, this knowledge base has to be populated with information about the internal structure and organization of the autonomic component, as well as its environment and context and possible ways of perceiving them. To support openness, it has to include information about how to connect and interact with other elements in various environments. Finally, to anticipate behavior, it is important to have an extensive set of policies and rules determining the ability of the system to predict various changes [18]. As explained in the next section, this self-reflective knowledge component plays an important role in implementing closed feedback loops when engineering autonomic systems.

2.2.1 MAPE-K Reference Model

Autonomic computing is implemented by an autonomic manager component and a managed resource component using the MAPE-K control loop (Monitor, Analyse, Plan, Execute, Knowledge) [31]. This MAPE-K control loop is more like a structural arrangement than a sequential control flow.

IBM's vision of autonomic computing was influenced by agent theory, and the MAPE-K model is similar to and was probably inspired by the generic model for intelligent agents proposed by Russell and Norvig, in which an intelligent agent perceives its environment through sensors, and uses these percepts to determine actions to execute on the environment [42]. This process of sensing and acting upon sensed values clearly corresponds to the closed adaptation loop of the MAPE-K model [19]. Applying the model to the domain of self-management in IoT platform, we now consider each of its elements in more details.

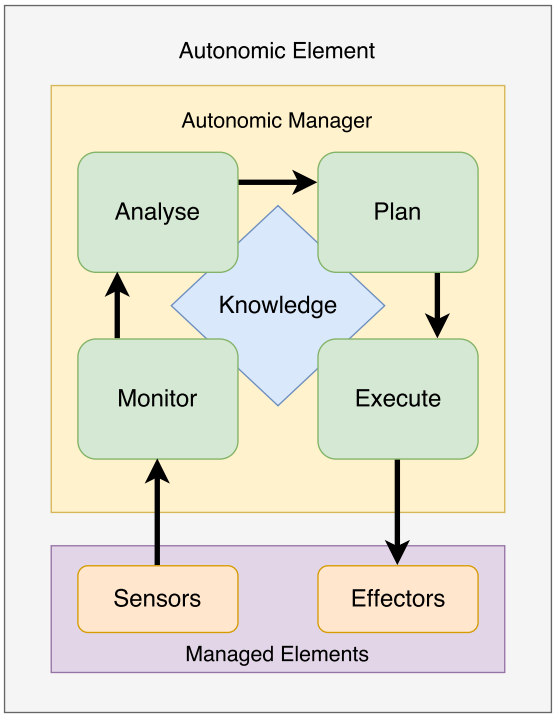

Figure 2.8: MAPE-K feedback loops (Adapted from [32]).

Managed Element

The managed elements represent any software and hardware resources which are enhanced with autonomic behavior by coupling with an autonomic manager. Managed elements are equipped with sensors which are either software or hardware components responsible for collecting information about the managed elements. Sensors are typically associated with metrics – certain characteristics of the managed element, which need to be monitored (e.g., response time, memory utilization, network bandwidth, etc.) [18]. Managed elements are also equipped with effectors which are components responsible for carrying out adaptation actions to the managed elements [11]. Depending on the scale, adaptations can be coarse-grained (e.g., completely substituting a malfunctioning Web service) or fine-grained (e.g., reconfiguring that service to fix it) [18].

Autonomic Manager and Knowledge

The autonomic manager is the core element of the model which is a software component that implements the whole MAPE-K functionality.The autonomic manager can be configured by human administrators using high-level goals. It uses the monitored data from sensors and internal knowledge of the system to plan and execute low-level actions that are necessary to achieve those goals.

The internal knowledge of the system is often an architectural model of the managed element – a formal representation of its internal organization, subcomponents, and connections between them [27]. Another important component of the knowledge base is a set of diagnosis and adaptation goals which serve to analyses critical situations and choosing

a relevant adaptation plan among existing alternatives [18]. These goals are usually expressed using Event Conditional Action (ECA) policies, goal policies or utility function policies [33]:

- **ECA policies** take the form "when event occurs and condition holds, then execute actions". They have been intensely studied for the management of distributed systems. However, the difficulty with ECA policies is that conflicts between policies can arise and be hard to detect [27].
- **Goal policies** are at higher level. They specify criteria that characterize desirable states and leave the task of finding how to achieve those states to the system, thus requiring planing on the part of autonomic manager and are more resource-intensive than ECA-policies. However, they still suffer from the problem that all states are classified as either desirable or undesirable; whenever a desirable state can not be reached, the system can not determine which undesirable state is least bad [27].
- **Utility functions** solve the above problem by defining a quantitative level of desirability to each state. The major problem with utility functions is that they can be extremely hard to define because every aspect that influences the decision of the utility function must be quantified [27].

The knowledge base can also include historical observations, data logs, repositories of previously detected critical situations and applied adaptation solutions, etc., which implies that the knowledge base is not designed to be static (i.e., populated once by administrators at design-time), but rather has to evolve dynamically over time by accumulating new information at run-time.

Monitoring

The monitoring component of the MAPE-K loop involves gathering information about the environment which are of significance to the self-management behavior of the systems [27]. Monitoring could be considered as the process of collecting and reporting relevant information about the execution and evolution of a computer system, and can be performed by various mechanisms [18]. The Autonomic Manager requires these appropriate monitored data to recognize failures or sub-optimal performance of the the Managed Element and execute appropriate changes to the system [27]. The types of monitored properties, and the sensors used, will often be application-specific, just as actuators used to execute changes to the Managed Element are also application-specific [31]. Two types of monitoring are usually specified in the literature as follows:

- **Passive monitoring** also known as non-intrusive, does not require any measurement code in the system to be added, but rather observe the actual interaction of the running system [27]. For example, in order to monitor some metrics of a software component, in Linux there are special commands (e.g., *top* or *vmstat* return CPU utilization per process) [31]. Linux also provides the */proc* directory, which contains runtime system information, such as current CPU and memory utilization levels for the whole system and for each process individually, information about mounted devices, hardware configuration, etc [31]. Similar passive monitoring tools exist for most operating system. The drawback of this approach is that often monitored information is not enough to unambiguously reason about possible sources of problems in the system.

- **Active monitoring** means engineering the software at some level, e.g, modifying and adding code to the implementation of the application or the operating system, to capture function or system calls [27]. This can often be automated to some extend. This kind of monitoring is also known as intrusive, as it inevitably implies making changes to the Managed Element by instrumenting it with probes to facilitate inspection of its characteristics. As with code instrumentation, it is essential that this is done with care, since the instrumentation can itself potentially affect the subject's performance, providing a flawed picture of its inherent capabilities [18].

As self-managing systems grow and the number of sensors increases, monitoring activities may result in a considerable performance overhead [19]. That is, in a system with thousands of probes constantly generating values, the monitoring component may not be able to cope with this overwhelming amount of data. To avoid 'bottlenecks', system architects have to distinguish between values which are relevant to self-managing activities and so called 'noise' data, which can be neglected [18]. Another potential solution to this problem is performing high-level monitoring first, and then, once an anomaly is localized, activate additional monitoring resources [24]. Therefore, computational resources are provisioned to the monitoring component on-demand, only when a problem is detected, thus resulting in a higher efficiency and resource consumption.

Analysis

Analysis component provides mechanisms that model complex situations based on the received details. This allows the central authority element to learn about the environment. This module can also be used to detect failures or sub-optimal behavior of the Managed Element. For instance, a simplest analysis engine based on ECA rules, simply detects situations when a single monitored value is exceeding its threshold (e.g., CPU utilization reaches 100%), and immediately sends this diagnosis to the planning component. However, it could be a challenging task, especially in a distributed environment where the monitored data is coming from multiple remote sources. Based on the internal knowledge, the Autonomic Manager should decide whether a particular combination of monitored values represents or may lead to a failure [18]. One possible solution is utilizing techniques from the area of Complex Event Processing (CEP) which considers the changes in the environment as atomic events [17]. Sequences of atomic events build up complex events, which in turn, may be part of an even more complex event, thus building event hierarchies. For example, when CPU and memory utilization levels of several VMs running on the same physical machine reach 100% (i.e., atomic events) within a short period of time, this indicates that the utilization of the whole physical machine has reached its limit (i.e., the complex event) [18].

Planning and Execution

Plan component provides mechanisms that guide actions to be effected on the Managed Element with the help of higher level policies, rules, and regulations. This module plans further action on the basis of the constraints that have been imposed in the system. The action is performed to achieve system goals and objectives. For instance, ECA rules directly produce adaptation plans from specific event combinations. Execute component carries out the adaptation plan generated at the previous stage to the managed element by means

of effectors. Once changes have been applied to the system, a new adaptation cycle is triggered, newly generated values are monitored and analyzed, an appropriate adaptation plan is generated and executed, and so on.

However, applying adaptation plan in a stateless manner where the Autonomic Manager keeps no information on state of the Managed ELement and relies solely on the current sensor data could be problematic [27]. Indeed, it is necessary that the autonomic should keep information about the state of the Manage Element in a context model that can be updated progressively through sensors[27]. This model can be used to reason about the Managed Element in order to plan adaptations. The advantage of this model-based approach is that, under the consumption that the Manged Element is correctly manifested in the model, the architectural model can be used to verify that the system integrity is preserved when applying an adaptation, thus guaranteeing that the system will operate properly after executing the adaption [27]. The use of the model-based approach can result in the delay of the time when an adaption plan occurs in the Managed Element. In fact, if the delay is sufficiently high and the system changes frequently, verification step where the adaption plan applied to the model can be skipped, thus, the plan may be created and sent for execution under the belief that the actual systems was in a particular state [27].

2.3 Autonomic computing in IoT platforms

From a system-level perspective, IoT could be considered as a highly dynamic and radically distributed networked system which is comprised of many smart objects producing and consuming information. The ability to interact with the physical realm is achieved through the presence of sensors which are able to detect input from the physical environment and transform them into digital signal as well as through the presence of actuators which are able to trigger actions onto the physical environment [37]. As scalability is expected to become a major issue due to the extremely large scale of the resulting system, and considering also the high level of dynamism in the network (as smart objects can move and create adhoc connections with nearby ones following unpredictable patterns), the quest for inclusion of self-management and autonomic capabilities is expected to become a major driver in the development of IoT solutions [21, 26].

In recent years, achieving atomicity in the IoT has begun to attract much research attention. In particular, most of the research focus on the practical implementation of autonomy using self-star (self-*) behavior in IoT devices in order to reduced the need for manual intervention and management at physical level. Table 2.1 summarizes some work that has been done in this direction. There are some other work initiating the self-management concept into IoT platform in order to achieve autonomic behaviors for the IoT application. For example, Cetina et al. [27] propose leveraging variability models made at design time to enable the application reconfiguring capabilities at runtime. Sarkar et al. [43] introduced very simple contextual information about the smart object such as: current location, operating state, etc. in order to initiate and execute an automated services intelligently. However, these studies are either targeting at general purpose application or adhoc solutions.

Besides aforementioned IoT issues (such as heterogeneity, interoperability, interoperability) which are attracting much effort toward self-management approaches, there are also other aspects in IoT that need more attention. As an example, security is also an important criterion in IoT development. IoT is extremely vulnerable to attacks for several

No	Title	Summary
1	Autonomic wireless sensor network topology control [47]	Method based on interpolation algorithm to maximize devices operational lifetime by automatically hibernating and activating in response to data perceived from a dynamic environment.
2	Self-organizing network with decision engine and method [23]	Encapsulates information (e.g. priority, number of hops, etc.) and packet handling commands into the data packet in order to enable the source node adjust its operation (e.g. transmitting power, frequency, bandwidth, etc.) in a way that allows the network to be self-organizing, self-configuring, and self-healing so that data packets are retransmitted from source nodes to destination nodes with a minimum of hops and delay.
3	Plug-and-play sensors in wireless networks [22]	Achieving self-configure network by proposing a self-identification protocol that allows the network to configure dynamically and describe itself instead of a network host (Cluster head), which keeps track of sensors when new ones are added or which ones are in range etc. However, keeping the configuration table updated is difficult. Also the solution is restricted to Bluetooth specific technology.
4	Autonomic protocol and architecture for devices in internet of things[12]	Proposed a conceptual architecture based on MAPE-K framework and an associated protocol with specific communication procedures for the IoT devices.
5	Auto-configuration system and algorithms for big data-enabled internet-of-things platforms [41]	Target at self-configuration for M2M systems and proposes four algorithms for achieving self-configuring gateway parameters toward large number of end devices.

Table 2.1: Comparison of concepts of autonomicity applied to wireless sensor networks.

reasons. First, its components are often unattended and remotely located; which raises the possibility of physical attacks. Second, IoT uses wireless technology for communication which is easier to compromise [11]. Regarding this problem, Ashraf et al. [11] conducted a survey regarding threat mitigation approaches in IoT using an autonomic taxonomy and

finally sets down future directions toward self-security. In summary, these directions toward achieving self-* behaviors in IoT platforms are still immature and promising to be developing in the future.

In the following chapter, we will explain in more detail some of these work which we found most important and closely related to the context of our research, followed by a discussion about their disadvantages and our approach of bridging those technology gaps toward developing an autonomic IoT platform.

Chapter 3

Related Work

In this chapter, we present the state of the art on the aspect of introducing automation to IoT platforms as well as autonomic approaches which we found relevant to the objectives of our work. The goal is to provide the reader with an understanding of existing techniques, tools and approaches as well as to identify technology gaps in the existing research body and position our own work respectively. Specifically, the identified gaps will serve to outline the fundamental requirements for our proposed platform and will help to evaluate the benefits of the platform with respect to other approaches.

3.1 Sarkar et al. approach (DIAT architecture)

Sarkar et al. [39, 43] proposed a layered and distributed architecture for IoT, called Distributed Internet-like Architecture for Things (DIAT) which promised to solve many technical challenges in IoT such as scalability, heterogeneity and heterogeneity (see Figure 3.1). The design principle of the approach is that any physical/real world object in this world can have a virtual representation through a Virtual Object (VO). A VO includes a semantic description of the functionality, and hides the heterogeneity of the real world object. Several VOs can be aggregated to form a Composite Virtual Object (CVO), which can provide more comprehensive and resilient services. In general, CVOs are composed to accomplish a specific service request. The functionalities of IoT infrastructure are grouped into three layers: Virtual Object Layer (VOL), Composite Virtual Object Layer (CVOL) and Service Layer (SL), which are put together as a stack, called IoT Daemon:

- *VO layer:* is responsible for semantic modeling of physical objects or entities into the digital domain as Virtual Object (VO). Through semantic technologies, the VO layer provides universal methodologies to access all physical objects. Furthermore, the VOL plays the role of bridging the gap between the physical and the cyber world, thus, helping to tackle the heterogeneity and ensures interoperability and re-usability of objects.

- *CVO layer:* In this layer, VOs are mashed together to address a specific service request generated by the user or by the system. This layer provides functionality to semantically search and query specific types of CVOs for service accomplishment.

- *Service layer:* accepts the request from the users and analyzes the service requests to determine the types of CVOs required for service accomplishment. This layer also

handles service composition and orchestration in dynamic environments.

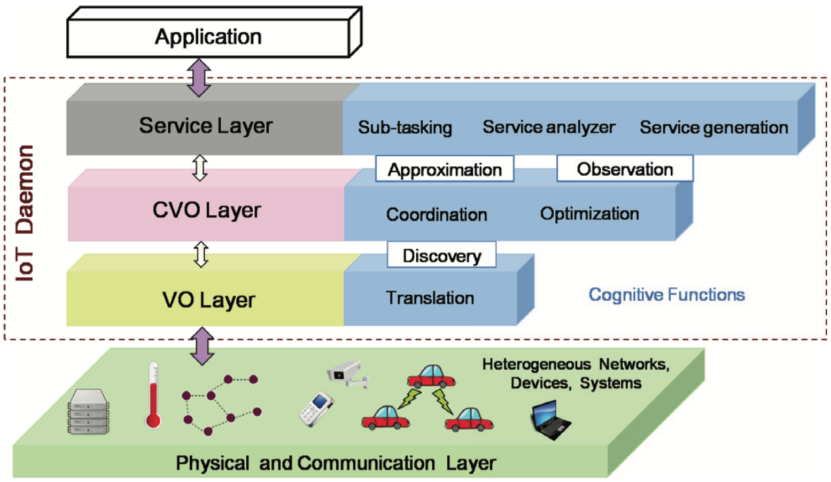

Figure 3.1: DIAT Architecture (Adapted from [43]).

The authors of this work also showed their effort in creating a unified knowledge base for IoT which extend various common ontologies. They specially focused on modeling dynamic environments in which IoT entities operate. The ontologies was integrated into their architecture and serve as a common standard for knowledge representation and communication between layers (see Figure 3.2 for the mappings between various ontologies and layers of the architecture).

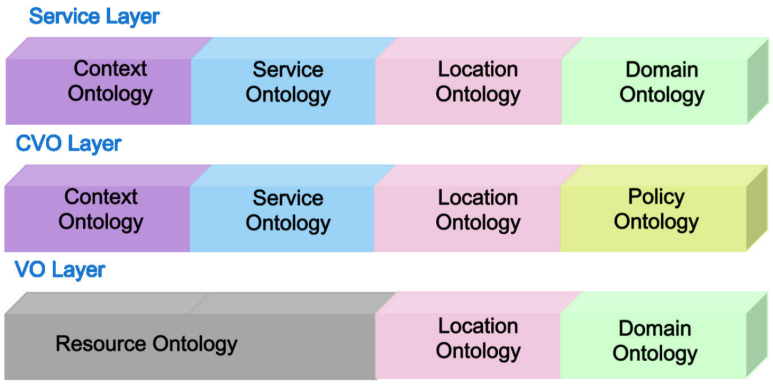

Figure 3.2: Mapping between various ontologies and layers of DIAT (Adapted from [39]).

Autonomic capabilities were not explicitly mentioned in the work. However, a workflow which is similar to the MAPE-K model was also introduced as a description of the functionalities of the observer which is responsible for automation of machine-to-machine(M2M) communication in order to provide a service intelligently (see Figure 3.3). The functionalities and the workflow of an observer are spread across the CVOL and SL. The observer continuously monitors objects and assesses their situation. Based on available knowledge and current situation, it may decide to initiate a service request and some necessary matching service(s).

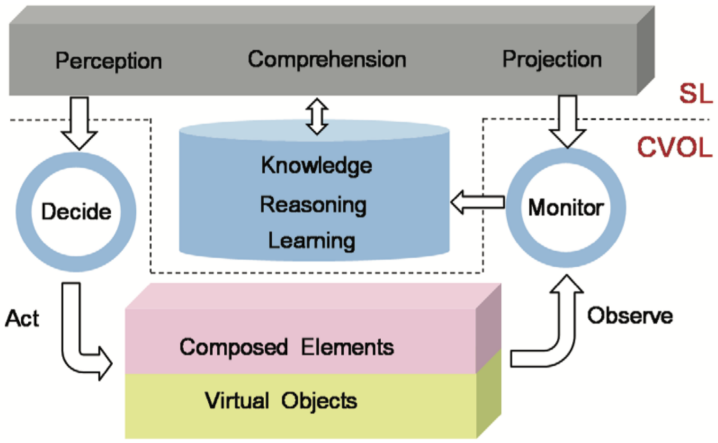

Figure 3.3: functionalities and workflow of a DIAT observer (Adapted from [43]).

With this regard, Sarkar et al. also applied a closed adaptation loop for their observer in order to achieve self-governance with the use of ontology for knowledge representation. However, this approach is relatively simple and still at the state of conceptual design without detail discussion about the functionalities of the components in the loop. As an example, there was no description of how the ontologies could be exploited for reasoning or which techniques were used for motoring and making decision.

Sarkar et al. also presented how their proposed approach could be to support interoperability and intelligent decision making by demonstrating a use case which include multiple entities (e.g., smart home, smart car, smart car, smart fridge, etc.). However, this is only a theoretical architecture which is incomplete and lacks details about implementation and validation.

3.2 Alaya et al. approach (IoT-O ontology)

Aiming at semantic data interoperability for M2M communication, Alaya et al. [10] introduced an ontology for IoT, called IoT-O, which merges to together a set of popular ontologies and is enriched with new relevant concepts and relationships in the area such thing, node, actuator and actuation. According to the authors, prior M2M standards only focus on achieving interoperability at communication level while lacking of interoperability at semantic level. Therefore, communication between heterogeneous entities (e.g., servers, devices, applications, etc.) can be achieved seamlessly and independently from the underlying network and vendor-specific device technologies; however, the "meaning" of the exchanged message is not understood without prior conventions (i.e., data formats, encapsulation and semantics). The IoT-O ontologies serves as a unified knowledge-base which support semantic data for M2M communication.

IoT-O consists of five main parts: sensor, observation, actuator, actuation, and service models. Figure 3.4 shows how the selected ontologies are merged together to form this new ontology. The DUL upper ontology represents either physical or social contexts. The SSN ontology was selected to represent sensors in terms of measurement capability and properties, observations and other related concept. The SAN ontology was designed by the authors to describe actuators respectively. The QUDV ontology was selected to

represent quantities, units, dimensions, and values. The OWL-TIME ontology was selected to provide a vocabulary for expressing facts about topological relations among instants and intervals, together with information about duration, and about date time information. The MSM ontology was selected to describe relevant aspects about services.

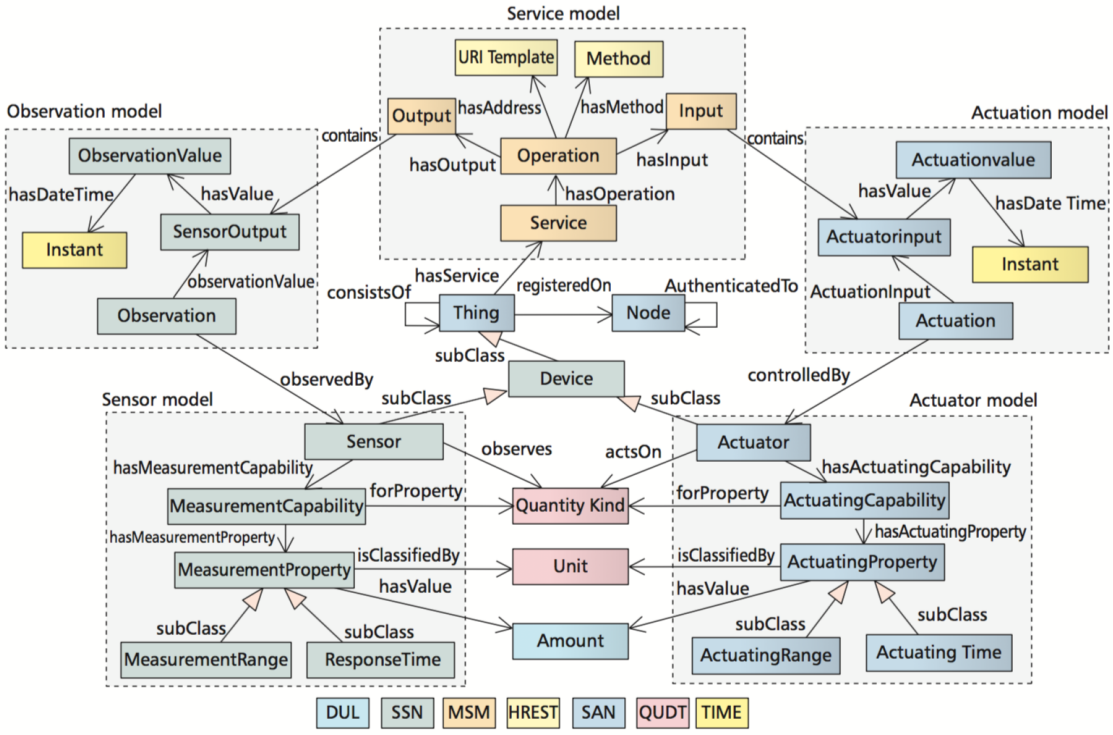

Figure 3.4: IoT-O ontology model (Adapted from [10]).

Alaya et al. also validated their approach by deploying it to their OM2M platform (see Figure 3.5 for the high level architecture of OM2M platform). The platform includes different sensors (e.g., temperature, humidity, luminescence, presence, etc.) as well as actuators such as electric plugs attached to different elements (e.g., lamps, fans, humidifier, etc.) which all gathered around different gateways that are connected to one central server.

The authors of this work were successful to demonstrate the use case of seamless device discovery and interaction by using the proposed ontology. In particular, the platform was able to discover newly plugged devices, browse the exposed attributes and methods, and finally interact with them by retrieving sensed data or triggering actions. Furthermore, all exchanged messages are augmented with semantics, thus applications not only have access to the data but also understand these data by matching them with the defined ontology.

Alaya et al. also introduced the idea of exploiting their ontology together with inference rules (i.e, SPARQL) to achieve self-configuring M2M resources. The objective here is to help applications discover relevant devices and exchange data with the correct communication mode based on its description, role and relationship. This use case was not implemented yet and was also domain specific. However, it showed their very first step to achieve autonomic computing in IoT after obtaining a relatively complete ontologies which covers almost necessary IoT aspects. Similar to the DIAT approach, this work is

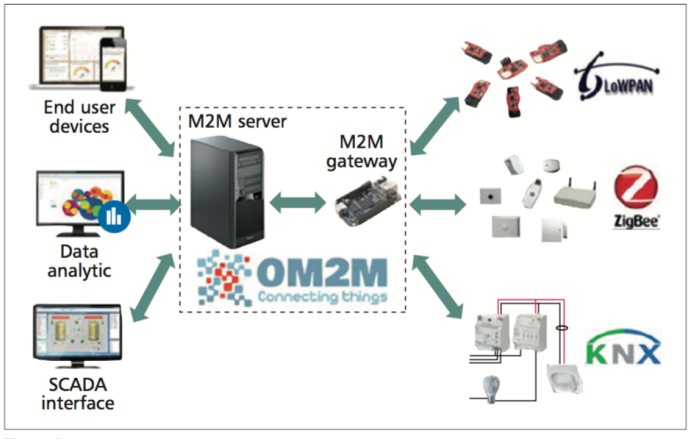

Figure 3.5: OM2M platform (Adapted from [10]).

also missing an effective conceptual model for implementing autonomic behavior. However, IoT-O is an important step to achieve autonomic computing as knowledge representation is vital in any autonomic system. Currently, the IoT-O ontology is one of the most complete ontology which covers almost necessary aspects of IoT area. The following chapters will describe how we extend IoT-O ontology with the concept of application and context to support semantics at application level.

3.3 Cetina approach (MoRE Engine)

Cetina [27] addressed the approach of reconfiguration by reusing variability models at runtime to provide a richer semantic base for decision making. In this way, the model made at design time is used for not only producing the system but also providing a richer semantic base for autonomic behavior during execution. Specially, his approach has two aspects:

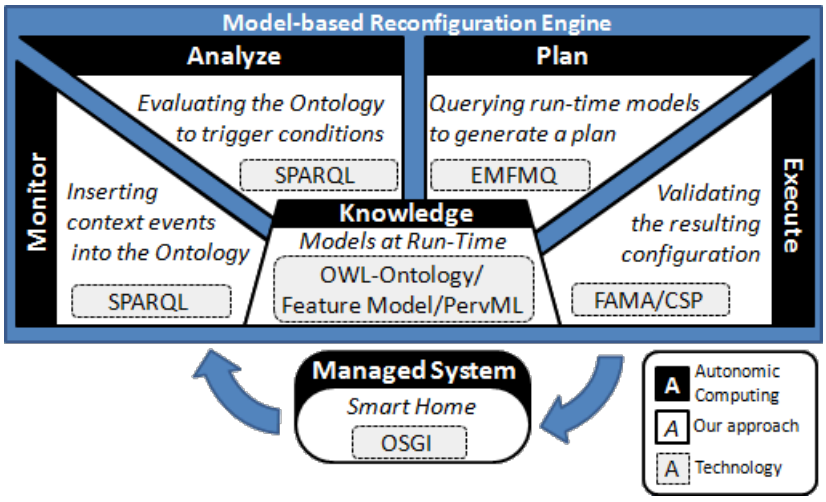

Figure 3.6: Overview of the runtime reconfiguration of Cetina et al. approach (Adapted from [27]).

- *Reuse of design knowledge to achieve Autonomic Computing*: reusing the knowledge previously captured in variability models to infer the variants in which a system can evolve. Therefore, given a context, the system itself can query these models to determine the necessary modifications to its architecture.
- *Reuse of existing model-management technologies at runtime*: using the same model representation - the XML Metadata Interchange standard - at design time and runtime, thus, making it possible to apply the same technologies used at design time to manipulate XMI models at runtime.

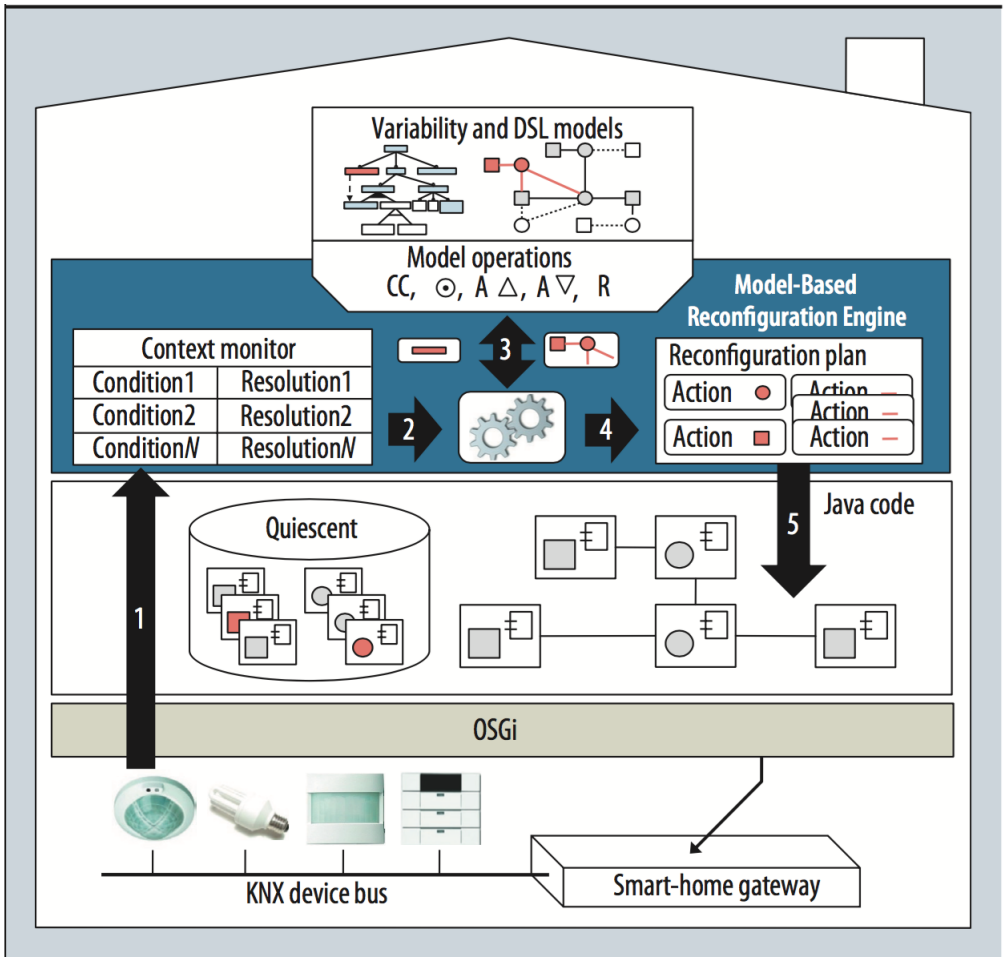

Figure 3.7: Model-based reconfiguration process. MoRE translates contextual changes into changes in the activation/deactivation of features. (Adapted from [27]).

The approach was deployed and demonstrated feasibility for smart homes domain, especially for self-healing and self-configuring capabilities. Figure 3.6 illustrated the overall reconfiguration steps of the approach which also applied the MAPE-K reference model for autonomic control. Detail steps of the reconfiguration steps was presented in Figure 3.7, which is also developed as an Model-Based Reconfiguration Engine (MoRE). A Context Monitor uses the runtime state as input to check context conditions (step 1). If any of these conditions is fulfilled (e.g., home becomes empty), MoRE queries the runtime

models about the necessary modifications to the architecture (step 2). The response of the models is used by the engine to elaborate a Reconfiguration Plan (step 3). This plan contains a set of Reconfiguration Actions, which modify the system architecture and maintain the consistency between the models and the architecture (step 4). The execution of this plan modifies the architecture in order to activate/deactivate the features specified in the resolution (step 5). In order words, Context events are represented as OWL ontology and system variability are captured by means of variability models which are used as policies that drive the system's autonomic evolution at runtime.

The reconfiguration of the system is performed by executing reconfiguration actions that deal with the activation/deactivation of components, the creation and destruction of channels among components and the update of models accordingly to keep them in sync with the system state. MoRE makes use of the OSGi framework for implementing the reconfiguration actions. This Framework implements a complete components model that extends the dynamic capabilities of Java.

MoRE framework is relatively similar to our approach in a way that they apply the same MAPE-K control loop as the base reference model for autonomic computing. In fact, as discussed in previous section, MAPE-K model is an effective model for developing autonomic capabilities. However, while we use rules and IoT ontologies as the policies to trigger adaptation, Cetina exploited variability model at design time for deciding the reconfiguration plan, which could make their solution limited to specific kinds of applications since not every system has a complete design model. In addition, using variability models could make it less flexible to policies changes comparing with using rules where upgrading adaptation plans could be done by just adding new rules.

3.4 Dautov approach (EXCLAIM framework)

Dautov addressed the problem of scalability in cloud platforms evolution by creating an Extensible Cloud Monitoring and Analysis (EXCLAIM) framework which utilizes techniques from the Sensor Web research community [18] to achieve autonomic behavior. Figure 3.8 illustrated the conceptual architecture of the EXCLAIM framework. Similar to MoRe approach, this framework also employed the MAPE-K model as an underlying reference model for Autonomic Computing. However, they use different technique from the Semantic Web Stack (i.e., SWRL rules) to specified the policies for reconfiguration. This approach mainly focused on monitoring and analyzing continuously flowing data with cloud platforms in a timely manner.

Within the framework, raw data generated pass through three main processing steps:

- *The triplification engine* is responsible for consuming and 'homogenising' the representation of the incoming raw observation values. In order words, the main function of this component is to transform raw data into information by representing collected heterogeneous values (e.g., JSON messages, SQL query results, and text files) using a single uniform format (i.e., Resource Description Framework (RDF) format).

- *The continuous SPARQL query engine* supports situation assessment by taking as input the continuous RDF data streams generated by the triplification engine and evaluating them against pre-registered continuous SPARQL queries.

- *The OWL/SWRL reasoning engine* is responsible for generating an appropriate adaptation plan whenever a critical condition is detected and the identification of multiple

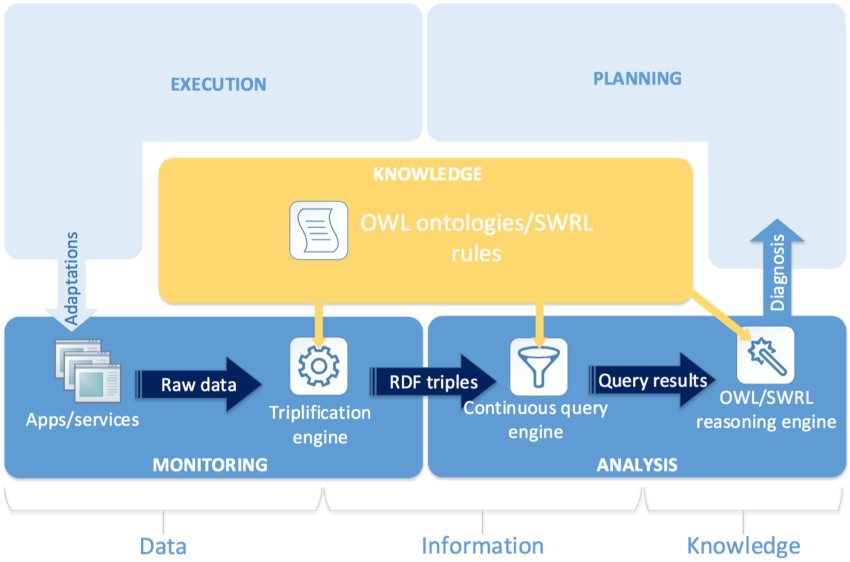

Figure 3.8: Conceptual architecture of the EXCLAIM framework (Adapted from [18]).

potential adaptation strategies.

Dautov also introduced a prototype implementation of the EXCLAIM framework in Java and deployed it on Heroku cloud platform as a proof of concept. Experiments also showed the potential of this approach in term of performance as well as the analysis capabilities of the autonomic framework. Although this work only focused on monitoring and processing real-time stream data, it showed the potential of applying Semantic Sensor Web and its related techniques in stream processing into autonomic computing, especially for large scale systems with highly dynamic operating environment. Therefore, in our work, we extend this approach with a forward step toward planing and execution. In the next chapter, we will discuss in more detail about this in the implementation of our SAI platform.

3.5 Toward the technology gaps in autonomic IoT platform development

In this chapter, we have surveyed existing relevant work which is the state-of-the-art in the research area of self-governance in IoT in general. Although some of them are still immature or even not fully implemented yet, these approaches helped us to understand current research situation in this topic as well as initiate the idea of a general purpose autonomic IoT platform, which is identified as the technology gap that is addressed by our proposed approach. Firstly, we found that self-management support in IoT Platform is either mature or limited to a specific application while it is feasible to enhance these platform with autonomic capabilities at highly efficient and more general level by reusing and extending existing solutions in the area of Autonomic Computing. In particular, we propose using techniques in Semantic Sensor Web into the MAPE-K reference model in order to implement a powerful engine for large-scale data processing and highly flexible planing. Secondly, IoT applications require rapid reaction to changes with relative efficient

resource consumption. Therefore, execution mechanism is also important in the design of autonomic IoT platform. Toward this problem, our proposal makes use of the complex event processing technique to model the IoT application as a set of complex events which are triggered under specific changes in the environment. Thus, the analysis and reconfiguring of applications could be done in a simple and efficient manner. Finally, by extending existing well-known IoT ontologies, our solution is capable of model fairly enough aspects of IoT application while achieving relatively quick solution to the problem of knowledge unification in autonomic computing. These considerations helped us to devise a novel approach to develop autonomic behaviors in IoT frameworks. With these ideas at hand, we first designed a conceptual architecture of the SAI platform, and then implemented a prototype version to demonstrate viability of the proposed hypothesis. In the next chapters, we will discuss our proposed approach in more details.

Chapter 4

SAI platform: A Service-oriented Autonomic IoT Platform

This chapter presents the conceptual design of the SAI platform in a top-down manner. To do so, we first introduce overview of the protocol stack of the SAI platform as well as explain in detail the role of each component in every layer. Our proposed design is relatively similar to most well-known IoT platforms presented in Chapter 2 except for the presence of the autonomic manager which is the key component that supports self-managing. In the second section, we then discuss our solution of applying the MAPE-K model to the design of the autonomic manager as well as concrete approach of using existing SSN technologies for each MAPE-K components, especially application modeling mechanisms which are exploited by planing and execution steps for reconfiguring applications. The chapter concludes with the implementation details of the SAI platform, where we describe our usage of the tools and techniques for realizing the conceptual idea.

4.1 Conceptual design of the SAI platform

SAI framework has adopted the following high-level system architecture which includes the concept of the M2M device domain and the network and application domain. Figure 4.1 describes the high level architecture of the SAI platform deployment. The architecture includes physical devices, the SAI platform which could be deployed on a server or any cloud platform and SAI applications which support cross platform smart devices.

The devices could come from different vendors with heterogeneous communication standards. It can connect directly to SAI platform via the Access Network or to a M2M Gateway via the M2M Area Network. M2M Area Network (e.g., Zigbee, 6lowan, etc.) provides connectivity between Devices and M2M Gateways. M2M Gateway acts as a proxy between devices and the network and may provide service to other devices connected to it that are hidden from platform. As stated in the previous chapters, IoT system may consist of millions of interconnected devices, providing and consuming information available on the network and cooperate. As these devices need to interoperate, the service-oriented approach seems to be a promising solution. Our platform also apply the same service-oriented infrastructure, which means each devices should offer its functionality as standard devices. In addition, sophisticated services can be created by the platform and based on only the provided functionality of other entities that can be provided as a service. This SOA paradigm promises to improve the reactivity and performance of the system since

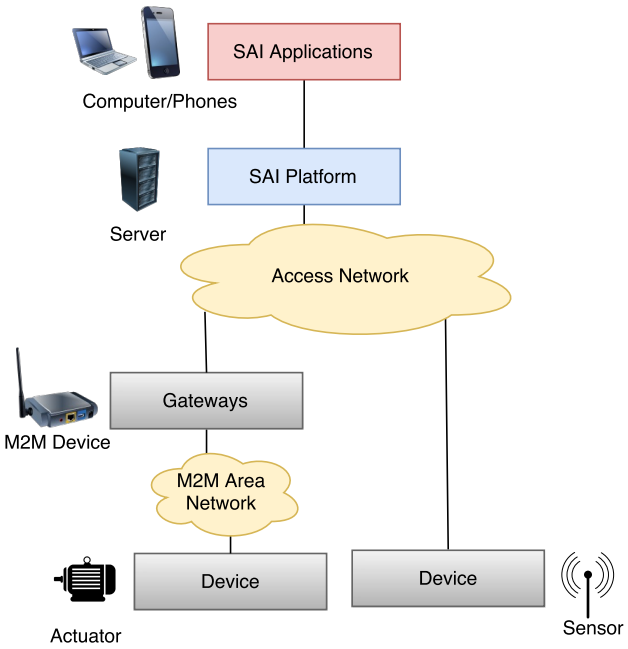

Figure 4.1: High level overview of the system architecture.

the information is available in near real-time based on asynchronous events. It also help to reduce the cost and effort required to realize a given scenario as it will not require any device drivers or third-party solution [45]. Accordingly, to properly accomplish the task through the usage of devices, SAI platform provides applications with service accessibility, whatever in the network, server or gateway. It provides services through a set of open interfaces to manage subscriptions and notifications pertaining to events, thus simplifying and optimizing application development and deployment through hiding of network specificities.

4.1.1 SAI architecture

Based on the above discussion, we propose a service-oriented IoT architecture that consists of fours layers which we refer to as the SAI platform as well as the connection with the applications and devices. Figure 4.2 depicts overview architecture of the SAI platform. This architecture hides the heterogeneity of hardware, software, data formats and communication protocols that is present in today's embedded systems. It foster open and standardized communication via web services at all layers. In addition, the platform is also equipped with the autonomic manager which is developed by exploiting functionalities provided in each layer of the platform.

Devices: This layer represents heterogeneous devices which are expected to connect to the platform. These include industrial devices, home devices, or IT systems such as mobile phones, PDAs, production machines, robots, building automation systems, cars, sensors and actuators, RFID readers, barcode scanners, or power meters, etc.

Devices Abstraction: This layer carries out the physical devices abstraction. Specifically, devices either offer services directly or their functionality is wrapped into a service representation which will then be accessed through the framework. In the best case, a

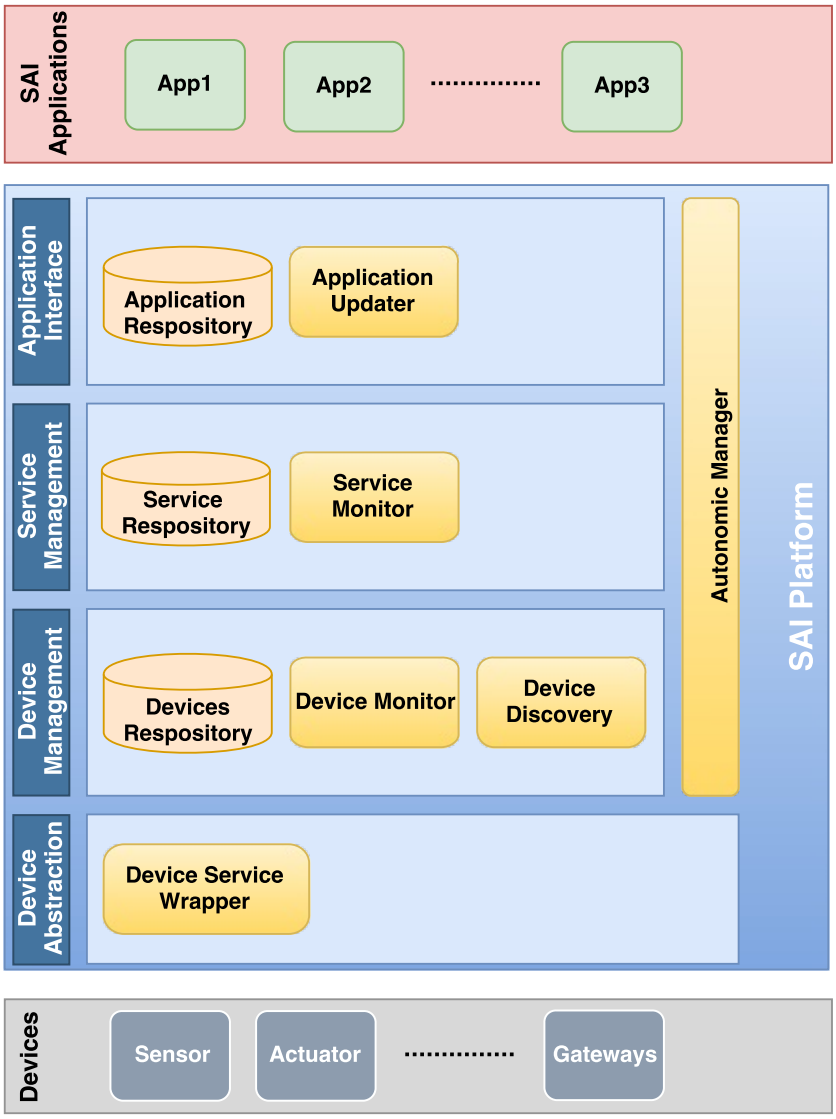

Figure 4.2: SAI platform architecture.

device offers discoverable web services on an IP network, then no wrapping is needed because services are available already. In this case, the application could also interact with these devices directly through the use of services without the support of the platform, thus reduce the burden of communication processing on the platform. In fact, similar efficiency could also be achieved if the device abstraction is deployed on an intermediate node (e.g., gateway) or a separate server which is responsible only for communicating with heterogeneous devices. Whenever the device does not support service based communication (e.g., it might use a message-based or data-centric communication mechanism), the abstraction into services that offer operations and emit events is required to achieve standardization. This could be done in the ***Devices Service Wrapper*** component by adding to it new plugin for that specific type of device. Thus the support of new technology or protocols is simply achieved through the implementation of the plugin module. In this way, any application in the system could access any heterogeneous physical technologies using

standardize restful operation. This can be done without any knowledge of the underlying network technology or its low level mechanism. In addition to enabling the communication with devices, this layer also provides a unified view on remotely installing or updating the software that runs on devices and enables the devices to communicate natively (i.e. in their own protocol with back-end devices).

Devices Management: This layer is responsible for dynamically discovering, monitoring and updating the status of all devices integrated into the system. Specifically, the ***Devices Repository*** component supports storing information about the devices available in the system. As discussed in previous section, this repository also contribute to the unified knowledge based of the autonomic computing and is stored under instances of OWL ontology. A detail description about the structure of these ontologies will be introduced later in this chapter. The ***Devices Monitor*** component keeps monitoring the devices and updating their attributes or status into the repository. It is also responsible for detecting events like failures and triggered corresponding event to the interested components. The ***Devices Discovery*** component is able to find devices present on the network, and retrieve information about them and their hosted services. This information will then be stored in the ***Devices Repository*** to be used by other components from the architecture.

Service Management: All functionality offered by the physical devices is abstracted by services. As discussed earlier, either devices offer services directly or their functionalities are wrapped into service representations. From the perspective of this layer and all layers above, the notion of devices is abstracted and only visible under services. An important insight into the service landscape is to have a repository of all currently connected services instances. This is done in the ***Service Repository*** component. Similar to the ***Devices Repository***, this repository also make use of the OWL ontologies technology. The ***Service Monitor*** keeps tracking all the services which are either used by an application or available for a new connection, and updating the corresponding information into ***Service Repository***. Flexibility is highly required in this component since the set of service offered at the device level is dynamic; as devices connect and disconnect or are re-configured, thus the available service set changes frequently. In addition to motoring services, the ***Service Monitor*** also supports the composition of services in order to obtain highly efficient services and simplify logic at application level. This could be by just introducing the specification of the complex service to the ***Service Repository*** database.

Application Interface: As discussed above, the SAI platform enables applications to interact and consume data from a wide range of physical devices using high-level abstract interface that features web services standards. This allows networked devices to directly participate in the operation without the knowledge about the details of the underlying hardware. In this layer, the concept of application reconfiguration is also introduced in the ***Application Updater*** component through the communication with the deployed applications about the new adaptive configuration whenever needed. This component is also responsible for monitoring the status and attributes of the application and updating them to the ***Application Repository***.

Applications: This layer represents the applications which are developed based on the SAI platform. These application could be deployed on various frameworks from computers, laptops, smart phones to limited resources embedded devices. As will be discussed later in this chapter, SAI application logics are described as a set of rules which are executed by an interpreter.

In conclusion, this proposed architecture exploit the concept of SOA to allow application to be composed by loosely coupled services and provide ecosystem-wide discoverability for developers to easily wire up services as building blocks into application. By doing this, the approach also address the heterogeneity in physical devices. Furthermore, the support of seamlessly discovery and integration of new devices also improves the performance of the platform toward scalability issue. Interoperability challenge is also addressed by exploiting techniques in Semantic Web to support semantic understanding of communication data. In addition to addressing these challenges in IoT platform design, the approach is also enhanced with the autonomic mechanisms which are achieved by extending the functionalities of the components in the first three layers. The following section will discuss in more detail of how we exploit these components to implement the MAPE-K closed adaptation loop.

4.1.2 The Autonomic Manager

In this section, we present the conceptual architecture of the Autonomic Manager which plays as an key component to achieve autonomic behaviors in the SAI platform. This architecture was designed based on the MAPE-K reference model for implementing the closed adaptation loop. In addition, the functionality of each component of the loop was deployed by adopting existing concepts of Semantic Web Technology and facilitating other components in the SAI platform. See figure 4.3 the illustration of overview of the Autonomic Manger.

In order to support both *self-awareness* and *context-awareness* of the managed elements (e.g., the IoT applications that was deployed on heterogeneous framework), we need to employ an architectural model, which would be able to describe the internal organization as well as operating environment of the IoT application (i.e., devices, services, attributes, etc.). For this purpose, we propose using OWL ontology which will contain the required self-reflective knowledge of the system. Inspired by the SSN ontology, such architecture model represented with OWL will serve as a common vocabulary shared across the whole SAI platform. In addition, by using OWL ontologies, we could also reuse existing tools and technology in Semantic Web for processing purpose. In fact, in IoT area, there have been a lot of effort putting on developing OWL ontologies to support semantic understanding of exchanged data at application level. Therefore, adapting this OWL technology helps us not only to reduce effort on defining the IoT ontologies but also to quickly obtain the knowledge vocabulary reflecting the newly appeared aspects since OWL ontologies are extensible easily. Later in this chapter, we will describe the ontology which we used to accomplish autonomic behaviors, which was defined by combining extending existing OWL ontologies on different IoT aspects.

As can be seen in figure 4.3, apart from the knowledge representation, we distinguish 4 main elements of the Autonomic Manger: RDF Standardization Engine, SPARQL Query Engine, SWRL Reasoning Engine and Adaption Executer, corresponding to the four elements of the MAPE-K loops:

- The **RDF Standardization Engine** is responsible for consuming and "homogenising" the data generated by the deployed application, services, devices, etc with the support of the *Service Monitor, Device Monitor, Device Discovery* and *Application Updater* components. As described earlier, these components are responsible for monitoring every dynamic changes in the system and updating to the knowledge

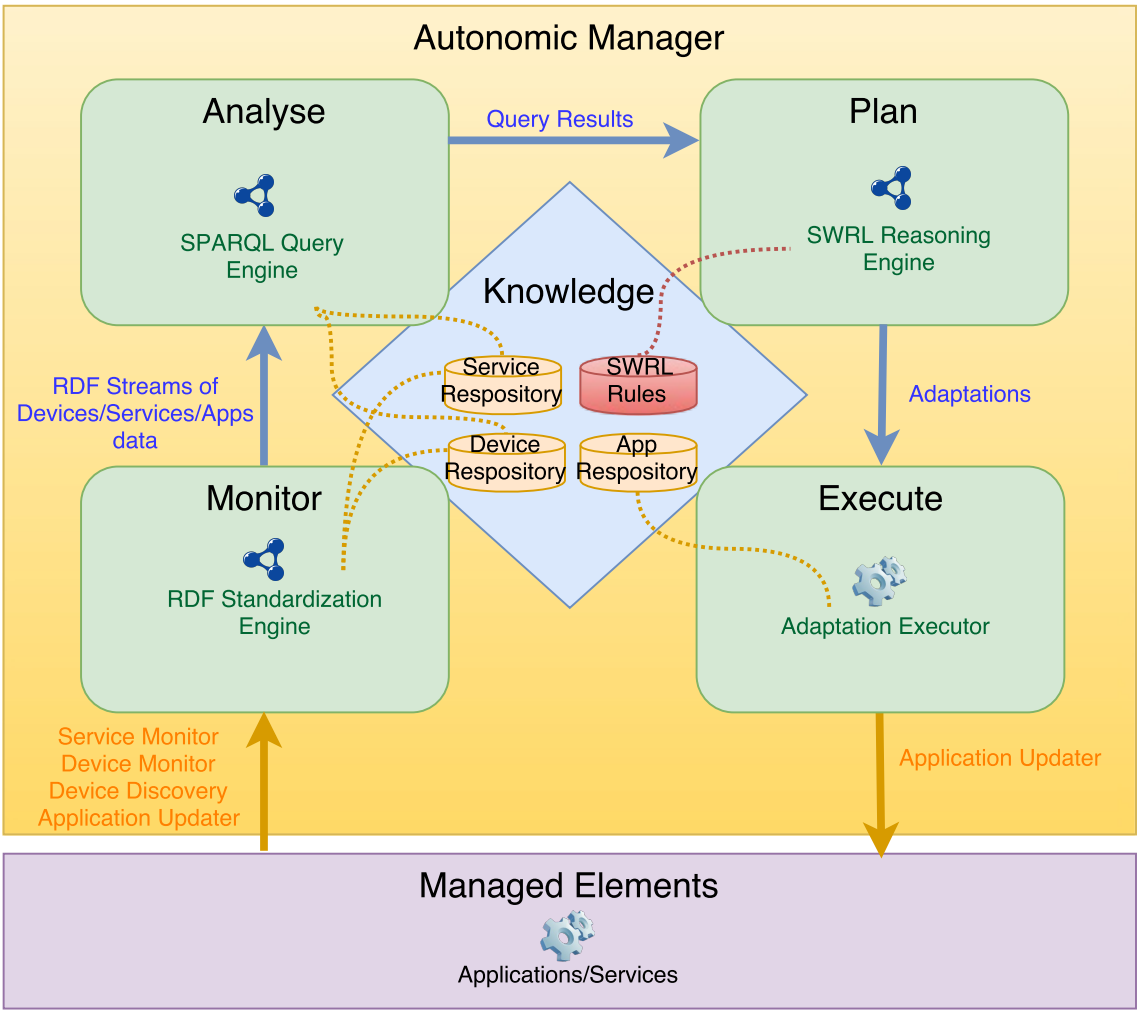

Figure 4.3: Conceptual architecture of the Autonomic Manager.

base accordingly. Since the observed IoT entities change their status very frequently, the corresponding data are generated in real time; thus putting more pressure on the analysis step. Thanks to the development of stream processing in SSN, we could exploit their query engines for analyzing the generated data. Therefore, in addition to monitoring the environment, the RDF Standardization Engine takes as input streams of data from the monitors and generates streams of RDF triples which manifest the defined OWL ontologies. As an example, this engine could measure the observed values of sensors as well as latency of the observation processes and generate respective stream of RDF triples. Using RDF as a common format for representing streaming data, and OWL concepts as subjects, predicate and object of the RDF triples allows us to benefit from human-readability and extensive support of query languages as explained above.

- In the analysis step, the **SPARQL Query Engine** takes as input the flowing RDF

data stream generated by the RDF Standardization Engine and evaluate preregistered continuous SPARQL queries against them in order to support situation assessment. In the first instance, situation assessment could be distinguish between usual operational behavior and critical situations by matching them against critical condition patterns. As an example, this engine could takes as input stream of the sensor observed values and triggers a malfunctioned indication whenever encounter values exceeding the operational range. Since different devices could have different operating conditions, thus varying in diagnosis steps, machine learning techniques could also be applied to assist with maintaining the list of critical condition patterns. Adding new detection patterns is just simply registering new SPARQL query into the engine. This could be done with minimum delay by the support of existing SPARQL query engine. To some extent, this step can be seen as a filtering stage since the outcome of this step is just a sub-set of RDF triples, which represent potentially critical situations. The querying engine will only trigger in response to potentially critical situations, thus saving the next step from reasoning over numerous non-critical, 'noisy' information. 'On-the-fly' processing of constantly flowing data, generated by a great number of sources, as well as employing one of the existing RDF streaming engines, are expected to help us in achieving near real-time behavior of the Autonomic Manager.

- After recognizing a critical situation, a corresponding confirmation as well as appropriate adaptation plan will be generated by the **SWRL Reasoning Engine**. This engine is not only checking simple "if-then" statements but also performing even more complex and sophisticated reasoning for a problem and possibly identify corresponding actions. To address this challenge we propose using SWRL rules which provides a sufficient level of expressiveness for defining policies as well as exempts us from the effort of implementing our own analysis engine from scratch. Accordingly, we will rely on the built-in reasoning capabilities of OWL ontologies and SWRL rules, thus the routine of reasoning over a set of possible alternatives is done by an existing, tested, and optimized mechanism. Similar to the analysis step, introducing new policies could be done by simply registering new SWRL rules in to the engine with minimum delay and effort, and taking effect immediately. Continuing with previous example where a malfunctioned sensor is detected, new SWRL rule should be added to the reasoning engine to check whether there exists any application using this sensor and generate proper adaptation (e.g., by substituting with an equivalent active sensor). The outcome of this planning step is a confirmed occurrence of a detected critical situation as well as the most relevant adaptation actions.

- Once the adaptation has been planned, the corresponding actions has to be carried out in order to make changes to the relevant components and application. This is the step where autonomic behaviors are implemented and should be achieved automatically and transparently to the users. The **Adaptation Executor** is responsible for executing any planning actions corresponding to the autonomic behaviors supporting by the SWRL Reasoning Engine. From the perspective of SOA architecture where applications make use of services for implementing their logics, we hypothesize that self-governance in IoT application could be done by reconfiguration at service level. Accordingly, the Adaptation Executor will support autonomic behaviors with new service reconfiguration. This approach helps simplify the process of executing

adaption while not restricting the autonomic level achieved in IoT platform since services could be combined into a new form of higher abstraction service in order with complicated logic, thus supporting to achieve complex autonomic behavior. In the next section, we will describe how we apply the same perspective to the the IoT application in order to support reconfiguration at application level. As an example of the functionality of the Adaptation Executor, whenever there is a need for an application to substitute its currently used sensor which was reported to be malfunctioning, the Adaption Executor will generate new configuration with an equivalent sensor, update the Application Repository and communicate the configuration to the corresponding application through the *Application Updater*.

4.1.3 The Application Model

After executing the adaption actions from planning step, the *Application Updater* sends the new configuration to corresponding application for reconfiguring. This step is also a part of the execute element in the MAPE-K model and carried out seamlessly at application level. As discussed earlier, the new configuration is actually a service adaptation plan (e.g., service substitution or upgrade, introducing new logic could be done at Service Management layer by combining multiple services.), thus a mechanism for modeling IoT application is require in order to not only improve the semantic understanding of the application but also helps reduce the cost of reconfiguration. With this respect, we propose to model the application as a collection of complex rules. As an example, listing 4.1 presents the rule that specifies the thermostat functions.

Listing 4.1: An example of a rule used in SAI application.

```
/**********************************************************************
 *  Rule to observe the temperature measured by the sensor1
 *  and control the actuator connected to the heater accordingly (on/off)
 *  in order to maintain the temperature around desired value (25 degC)
 **********************************************************************/
begin
   rule "thermostat_operation"
   use
      sensor temp_sensor@Sensor1;
      actuator heater_act@Actuator1;
   declare
      var desire_temp;
   action
      desire_temp = 25;

      if(temp_sensor.Sen1GetTempService() > desire_temp)
         heater_act.Act1SetPowerService(heater_act.Act1PowerMinValue);
      else
         heater_act.Act1SetPowerService(heater_act.Act1PowerMaxValue);
end
```

Accordingly, SAI application is also equipped with an interpreter to execute those rules as well as other components necessary for the reconfiguration process. Figure 4.4 described the overview of components in an application. Overview of the communication flow between the platform and application is illustrated in figure 4.5. Specifically, an application deployed

on SAI platforms comprises of 3 elements: Application Rules, Interpreter and Application Monitor:

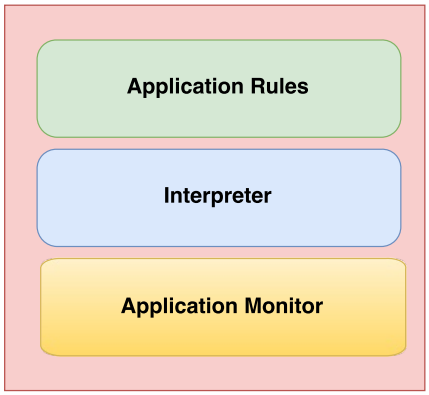

Figure 4.4: Architecture of a SAI application.

- The **Application Rules** specifies the functionalities of SAI application. Similar to any scripts, these rules also have specific syntax with predefined keywords. Listing 4.1 shows an example of a rule used in SAI application. Details of the structure of the rules will be described in the implementation section. Using rules could improve the human-readability and development effort while also support semantic understanding of the application, thus helping auto-reconfiguration. Accordingly, reconfiguring a SAI application is simply re-writing application rules, thus improving the cost of reconfiguration.

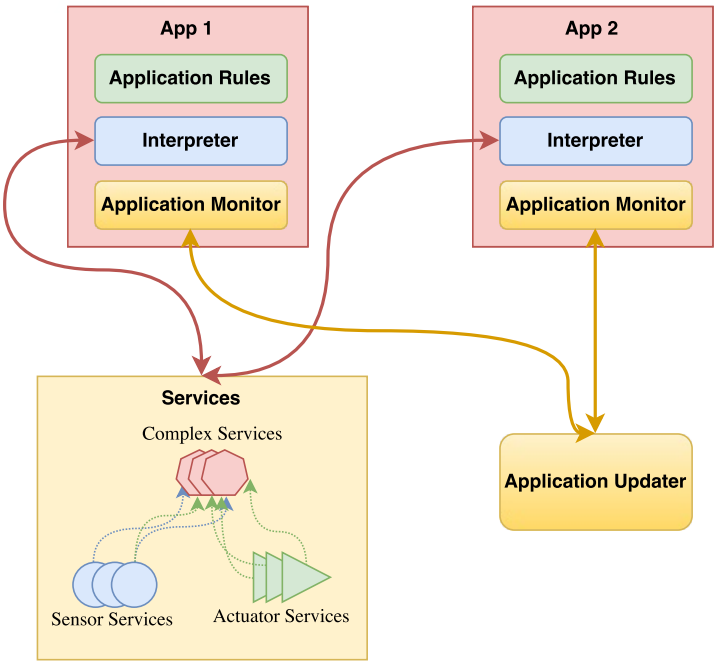

Figure 4.5: Overview of the communication between the applications and SAI platform.

- The **Interpreter** executes the scripts described by application rules. Similar to any interpreter, it analyzes and parses the rules into internal representation in order to understand the semantic of the rules and support execution. It also collects necessary information which is specified in the *Application Repository* for monitoring purposes. Once an application is deployed onto the platform, the interpreter keeps periodically checking the conditions and carrying out the actions specified in the rules. As discussed earlier, the communication between applications and physical devices is established under the mean of services. Specifically, the interpreter interacts with devices by using their web services api. These services could be deployed either on the SAI platform (e.g. through Device Service Wraper) or separate server (e.g. the devices support hosting their web services). In the latter scenario, the interpreter could communicates directly with the devices, the SAI platform acts as an monitor indicating adaptation when needed. In addition to executing applications rules, the interpreter also helps to changing these rules for reconfiguration purpose whenever receiving adaptation plan from the SAI platform.

- **Application Monitor** communicates with the *Application Updater* in order to update information about the application to the *Autonomic Manager* for monitoring purposes. This information could be the status and attributes of application as well as the used services which are specified in the *Application Repository*. Additionally, the Application Monitor also takes part in communicating the adaptation plans from the SAI platform to the deployed applications.

4.2 Implementation details of the SAI platform

Having introduced the conceptual architecture of the SAI platform in the previous section, in this section, we continue to discuss the implementation aspect of the the platform. Specifically, we focused on the main three elements of our work: the SAI Ontology, the Autonomic Manger, and the SAI Application.

4.2.1 Overview of the technical details

The prototype version of the SAI platform is implemented on Java framework, which was developed using Eclipse Neon IDE [1] as well as the following external libraries and tools:

- **Protege** [2] is a free open source ontology editor and knowledge-base framework with full support for the OWL 2 which is also the latest OWL specification developed by the W3C OWL Working Group. Protege supports creation and editing of one or more ontologies as well as their instances. In addition, it also provide simple interface for inserting SWRL and SPARLQ queries. There are also integrated reasoners for reasoning and tracking down inconsistencies as well as visualization support. In this project, we used Protege for importing and extending various IoT ontologies. The final ontologies was exported as a file loaded by the repository components.

- **OWL-API** [3] is an open source Java API and reference implementation for creating,

[1] `http://www.eclipse.org/downloads/packages/release/Neon/3`
[2] `http://protege.stanford.edu`
[3] `http://owlapi.sourceforge.net`

manipulating and serializing OWL Ontologies. The latest version of the API is focused towards OWL 2. The OWL API includes the following components:

- An API for OWL 2 and an efficient in-memory reference implementation.
- RDF/XML parser and writer.
- OWL/XML parser and writer.
- OWL Functional Syntax parser and writer.
- Turtle parser and writer.
- KRSS parser.
- OBO Flat file format parser.
- Reasoner interfaces for working with reasoners such as FaCT++, HermiT, Pellet and Racer.

In this project, we used the OWL-API for manipulating the ontologies as well as their instances. It provides methods for creating, modifying and deleting entities in the ontologies which are necessary for updating the the information stored in the repository. In addition to that, the library also provide various reasoners' implementations which supports reasoning over the knowledge base. In this work, we used the Pellet reasoner to test the ontologies and implement the SWRL Reasoning Engine as it works well with SWRL and SPARLQ queries.

- **C-SPARQL ReadyToGo Pack** [4] is a Java library for C-SPARQL which is a language for continuously querying over streams of RDF data developed by the Polytechnic University of Milan. C-SPARQL queries consider windows, i.e., the most recent triples of such streams, observed while data is continuously flowing. We used the library for the analyze phase of the Autonomic Manager to process RDF stream. C-SPARLQ querying is a designated filtering step before the actual reasoning step which is quite computational demanding.

- **Gson** [5] is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code available. The main supports provided by Gson are:

  - Provide simple toJson() and fromJson() methods to convert Java objects to JSON and vice-versa.
  - Allow pre-existing unmodifiable objects to be converted to and from JSON.
  - Extensive support of Java Generics.
  - Allow custom representations for objects.
  - Support arbitrarily complex objects (with deep inheritance hierarchies and extensive use of generic types).

[4] `http://streamreasoning.org/resources/c-sparql`
[5] `https://github.com/google/gson`

In this project, we used Gson for converting and exchanging RDF triples within the platform. Specifically, Gson was used to serializing the RDF data when exchanging data with the applications.

- **Alljoyn Framework** [6] is an open source software framework that enables interoperability among connected products and software applications across heterogeneous manufacturers. The framework simplifies the process of developing applications for devices and applications to discover and communicate with each other. The software supports popular platforms such as Linux and Linux-based Android, iOS, and Windows as well as other lightweight real-time operating systems. It has language bindings for C++, Java, C, JavaScript, C#(general) and C#(Unity) in development. In this project, we exploited the framework for the mean of communication between the SAI platform and applications. Since Alljoyn support various platforms, our implementation of SAI application as well as communication protocol could be reused later on different smart devices.

- **JavaCC** [7] is an open source parser generator for use with Java applications, which is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building, actions, debugging, etc. In this work, we use JavaCC for building the interpreter at the application side. The generated code is pure java language which can be reused for implementing the application infrastructure on other framework which support java (e.g., Android).

4.2.2 The SAI Ontology

Representing the central Knowledge element of the MAPE-K reference model (see Figure 2.8, the SAI ontology is also the core component of the SAI platform. Its vocabulary of terms is accessed and used at every step of the information processing workflow within the platform. In this section, we will explain the ontologies in details as well as their usages in the implementation of our MAPE-K model.

As mentioned above, the SAI ontology was developed using Protege IDE which is is a free, open-source, feature-rich ontology editing environment with full support for OWL 2, and integration of several logic reasoners such as HermiT and Pellet. See Figure 4.6 for overview of the IDE user interface. The IDE serves to create and edit one or more ontologies in a single workspace via a customizable user interface (i.e., users can arrange instruments and panels according to their individual preferences). Supported refactoring operations include ontology merging, moving axioms between ontologies, renaming of multiple entities, and many others. Visualization tools allow for interactive navigation through ontological class hierarchies (both explicit and inferred). It also supports debugging features – e.g., advanced explanation support aids in tracking down inconsistencies.

As discussed previously, our aim is to design an ontology that sufficiently covers most of IoT concepts. The ultimate goal is to enable semantic interoperability among IoT entities. Fortunately, there have been existing several works focusing on defining IOT ontologies which could be selectively exploited in order to obtain an efficient ontology for our purpose.

[6] `https://allseenalliance.org`
[7] `https://javacc.org`

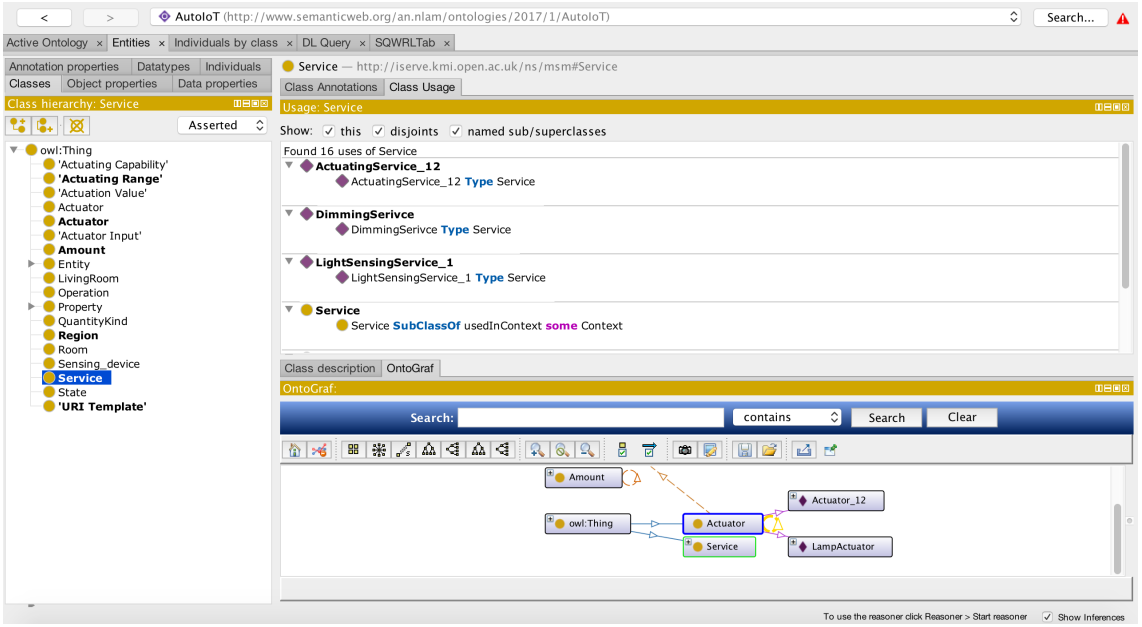

Figure 4.6: Protege IDE GUI

Specifically, our SAI ontology is developing based on the IoT-O ontology [10] which is also an extension of various well-defined ontologies representing different IoT aspects (e.g. SSN ontology for defining every concept related to sensors, SAN ontology for defining concepts of actuators). Reusing these ontologies helps us to reduce the effort for defining the model from scratch since each ontology showed to cover almost every concept in its designated aspect in IoT. In addition, we can always extend these ontologies with required subclasses or sub-properties in order to describe our intended situation if it has not been specified before, thus avoiding redundancy and repetitions. In fact, our SAI ontology extends the IoT-O with Application Model representing the concept of critical situation in Autonomic Computing which will then be used to reason for a new reconfiguration at application level.

The SAI ontology consists of six main parts: Sensor, Observation, Actuator, Actuation, Service and Application models which are mainly defined in SSN [8], SAN [9] and MSM [10] ontologies. Figure 4.7 illustrates the simplified SAI ontology and connections of every parts in the ontology.

The **Sensor Model** is used to described sensors and their related concepts - that is entities which are expected to generate data for later monitoring and analysis. This model can be used to present almost every aspect which is necessary for IoT application such as *SensingCapability* (e.g. type of sensors), *SensingProperty* (e.g. unit of the measuring value), *ResponseTime* and *MeasurementRange*. Together with Sensor Model, **Observation Model** is also defined in the SSN ontology, which represents dynamic aspects of sensor (e.g. the measuring value in real time), which will also be used as the vocabulary for RDF streams of observed data.

Aligned with the Sensor and Observation Model, **Actuator** and **Actuation Model**

[8] `https://www.w3.org/2005/Incubator/ssn/ssnx/ssn`
[9] `http://lov.okfn.org/dataset/lov/vocabs/SAN`
[10] `http://kmi.github.io/iserve/latest/data-model.html`

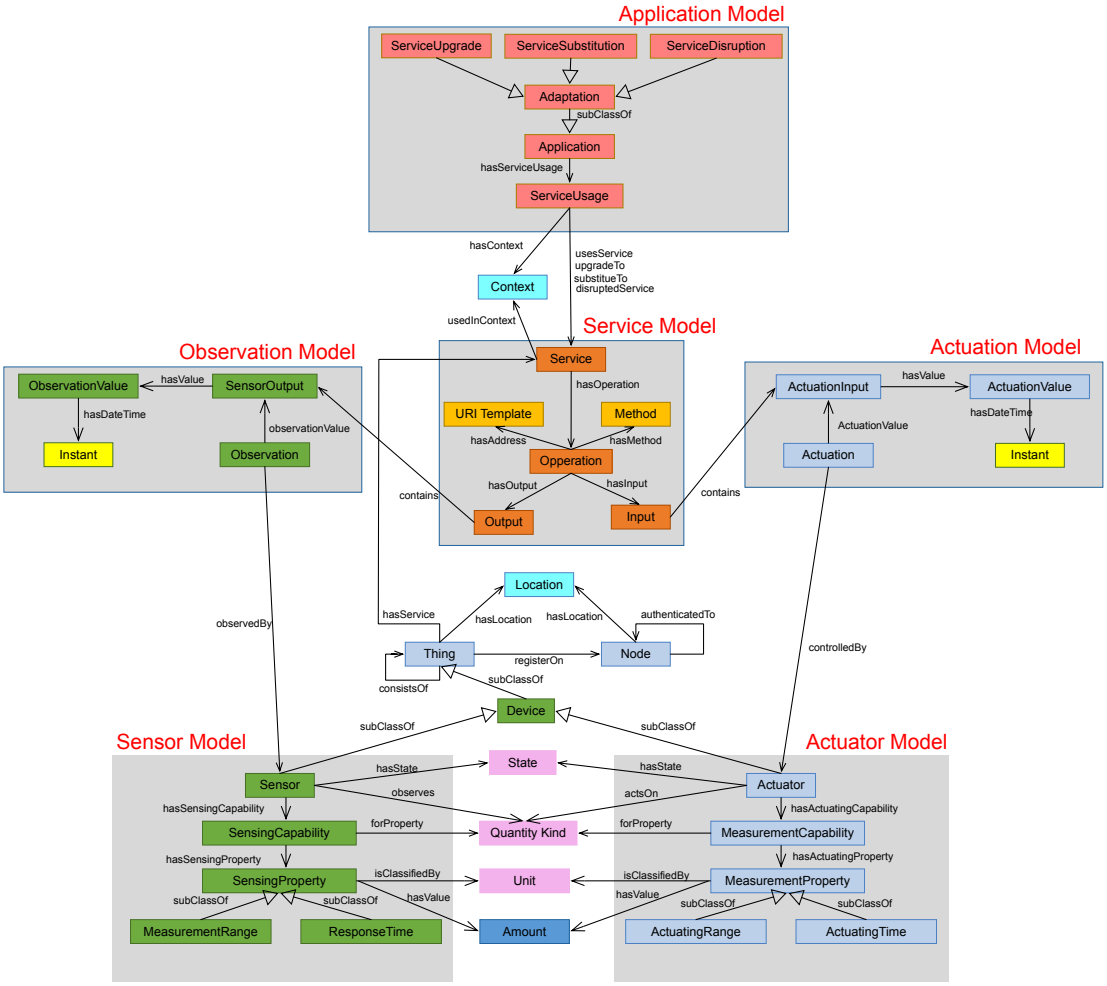

Figure 4.7: The SAI ontology Model

are used to represent every aspect of actuators. These two models are defined in the SAN ontology, which is part of IoT-O ontology, to describe actuators in terms of actuating capabilities and properties, actuation and related concepts. Along with SSN and SAN ontology, the QUDV [11] is also used to represent quantities, units, dimensions and values (e.g. *QuantityKind*, *Unit* and *State* Classes). In addition, the OWL-TIME [12] ontology is selected to express facts about topological relations among instants and intervals, together with duration and date time information (e.g. *Instant* Class). Furthermore, in order to described the physical location of devices, we extend both Sensor and Actuator Models with concept of *Location* which sufficiently specifies all necessary positioning aspects. This Location class is defined in the DOGONT [13] ontology.

In order to enable seamlessly interactions between application and services, it is important to represent how these services can be requested without any ambiguity in order

[11] `http://qudt.org/`
[12] `https://www.w3.org/TR/owl-time/`
[13] `http://iot-ontologies.github.io/dogont/`

to reduce amount of manual effort required for discovering and using them. The **Service Model** is used to provide vocabulary of all necessary aspect of these services. This model is defined in the MSM ontology that is able to capture the core semantics of both Web services and Web APIs in a common model, which is really close to our conceptual design of services. In this model, each service is described using a number of operations that have address, method, input, and output message contents. Furthermore, we also extend the Service Model with the concept of *Context* which represents the exact situation in which services is being used. Under the observation that physical devices may not be used in their intended situation (i.e. a bulb can also be used for indicating warnings), this Context class is used to describe different relevant using contexts of services, thus enhancing the knowledge base with meaningful facts for further intelligent decision making. In fact, this property plays an important role in approximating services purpose which will be discussed shortly in this section.

Figure 4.8 illustrates a concrete example representing a real actuator using the proposed ontology. The actuator is a digitally dimmable wireless lighting bulb having power ranging from 0 Watt to 50 Watt. The luminosity level can be dimmed by requesting the required power value. The light bulb offers a web service to enable remote luminosity control. The luminosity can be dimmed instantaneously by sending a create request to the address *https://192.168.1.10/LampActuator/Create/[value]* with a message body containing the required power. The ontology instance also specified the location of the actuator (Room1) as well as the contexts in which the actuator could be used (Notification and Lighting control)

The Application Model is used to described all concepts of the SAI application which are necessary for reasoning about the adaptation plan. Specifically, this class is intended to be used by the SWRL reasoning engine to distinguish between ordinary situation and situation which are critical and required certain responsive actions to be taken. The main subclass of the critical situation is *Adaptation* representing a situation which may threat the stability of the system or individual deployed application, and therefore has be reported and acted upon. Accordingly, there are several adaptation plan which represents each specific critical situations:

- *ServiceUpgrade* indicates a situation where the currently occupied service should be replaced with another one which provides the equivalent functionality with higher "quality" (e.g. low latency, low power consumption).

- *ServiceDisruption* indicates a situation where services being used by an application stop working or keep generating invalid values. In this case, the platform communicates the situation to the application as an exception indication. Further actions could be handled at the application level.

- *ServiceSubstitution* also indicates service disruption situation. However, the platform is able to specify an alternative service which at least approximately provides similar functionalities without affecting the logics of the applications. This process of approximating services is actually an interesting topics in many research works. In this work, it is primarily done by reasoning over the ontology with the help of SWRL rules. However, since the ontology could be easily extended and SWRL rules could be injected in run time, existing practices of other service approximating approaches may be applied to improve precision of the process.

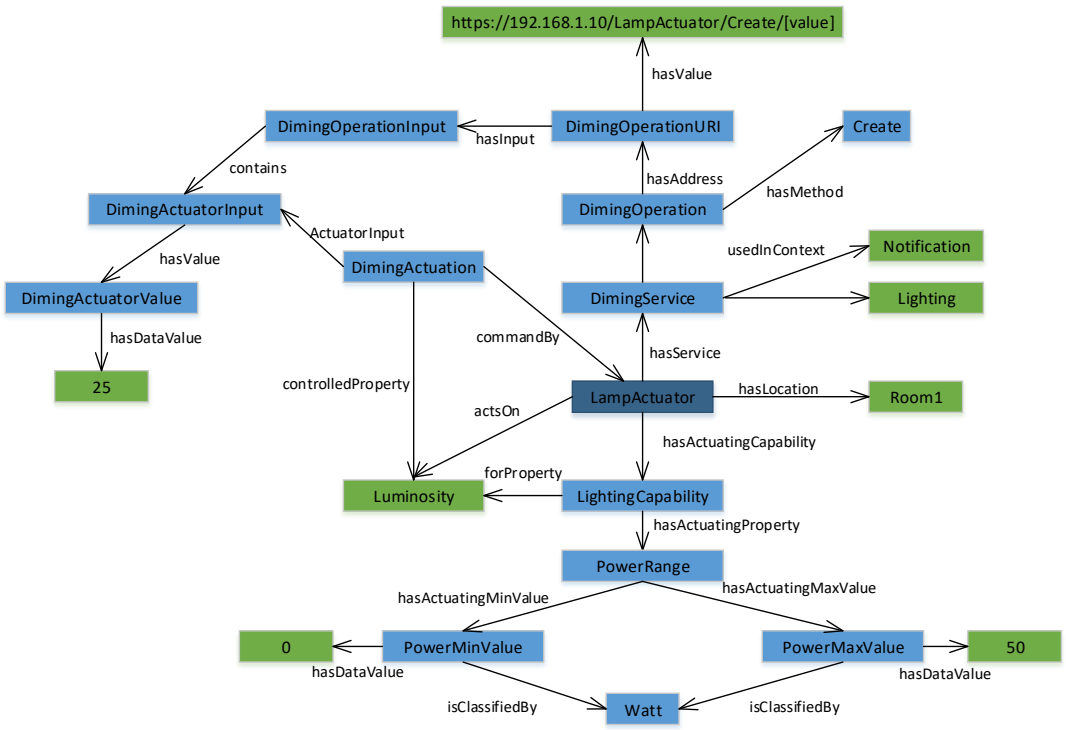

Figure 4.8: An Example Of Devices Ontology Instance

At the moment, our prototype is developed to support those three adaption plans. However, the above list of adaptation is not complete and indeed could be extended or enhanced with more details of adaption actions as explained previously. Additionally, in order to model the relation between applications and services, the Application Model is also equipped with *ServiceUsage* class. Accordingly, an application could have some ServiceUsage instances which indicate the services being used as well as their respective using situations specified by the *Context* instances. This ServiceUsage is also used to communicate with the application about new services for upgrading or substitution whenever there is adaptation plan through the use of methods such as *upgradeTo* and *substituteTo*.

4.2.3 The Autonomic Manager

In this section, we will explain implementation details of the SAI platform, especially the Autonomic Manager. This explanation is aligned with the main components of the conceptual architecture presented in the previous section and accompanied by several code snippets, which are intended to demonstrate how the key functions are implemented within the platform.

Figure 4.9 presents overview of the data flow in each component of the Autonomic Manager. The **RDF Standardization Engine** is the initial step of the data flow within the Autonomic Manager. As described earlier, this engine is part of the Monitor component and is responsible for extracting raw data from the managed elements and transforming

them into semantic RDF triples using the SAI ontology.

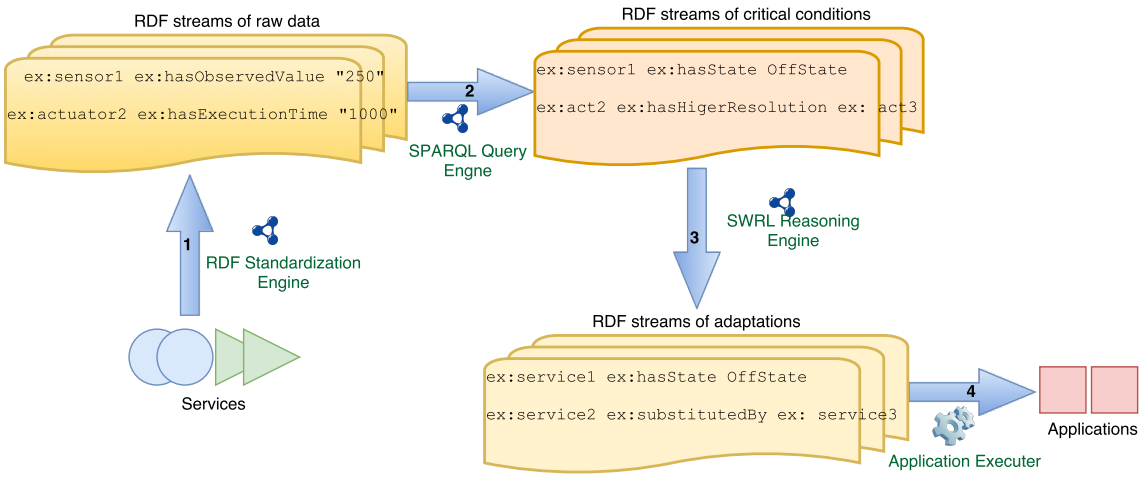

Figure 4.9: Overview of the data flow of the reconfiguration process.

Following the non-intrusive principle to data collection, this engine relies on already existing APIs provisioned by services to gather required metrics. For example, in order to check the status of a particular sensor, it is necessary to retrieve the current sample value observed by the sensor periodically by using the provided services specified in the Service Repository. Furthermore, the sampling rate of the SAI platform for data extraction should also be configured in accordance with the operating rate of the devices in order to provide the actual real time data and avoid redundant calls to the services. The following code snippet, for example, fetch the current temperature value observed by an Grove Temperature Sensor which supports Restful API call. The response of this service is in Json format which is then parsed into `SensorEntity` object that is defined according to Sensor Model.

Listing 4.2: Sample code to fetch the temperature value from Restful API.

```
1 Client client = Client.create();
2 WebResource webResource = client.resource("https://us.wio.seeed.io/v1/node/
    GroveBME280I2C0/temperature");
3 ClientResponse response = webResource.accept("application/json")
4                                      .get(ClientResponse.class);
5 SensorEntity output = response.getEntity(SensorEntity.class);
```

An important benefit of monitoring phase of the SAI platform is that there is no intrusion to the source code of the monitored services and applications. The framework only requires user credentials to get access to individual instances of services – an acceptable requirement given that the platform is assumed to be a trusted entity for the consumers. Once the raw data is extracted, it then needs to be uniformly represented using the predefined ontology. At the moment, this step is done manually by mapping between the source raw data and target semantically-annotated triples.

It needs to be explained that a single raw value could be transform into multiple RDF

triples, which form an RDF graph. Depending on the requirements, additional RDF triples serve to provide a more unambiguous and context-aware information to the next step. For example, Listing 4.3 below demonstrates how observed temperature value is translated into the RDF representation to be further processed by Autonomic Manager. Specifically, this Listing presents the existence of an instance of `Service` class which has an `Operation` with output value 60 degC.

Listing 4.3: A single temperature value is represented in multiple RDF triples.

```
sai:service-10 rdf:type msm:Service
sai:service-10 sai:hasId "TempSensingService_1"
sai:service-10 msm:hasOperation sai:operation-10
sai:operation-10 msm:hasOutput sai:output-124
sai:output-124 ssn:hasValue "60"^xsd:int
```

Accordingly, the newly-generated RDF triples are sent to the stream which is consumed by the **SPARQL Query Engine** which is responsible for detecting potentially critical situations and passing them further to the **SWRL Reasoning Engine**. This engine is part of the Analysis component which handles the RDF triple form the Monitoring component and pushes into the engine stream. An instance of the C-SPARQL engine is configured as shown in Listing 4.4 below.

Listing 4.4: Initializing the C-SPARQL Engine and registering a stream.

```
1 // Initialize C-SPARQL Engine
2 CsparqlEngine engine = new CsparqlEngineImpl();
3 engine.initialize();
4
5 // Getting the RDF stream from Monitor component
6 CsparqlStream csparqlStream = Monitor.getStream("http://myexample.org/
      stream1");
7 engine.registerStream(csparqlStream);
```

In order for the SPARQL Query Engine to function, it is required to assign a data stream and register a standing C-SPARQL query against this data stream, as illustrated in Listing 4.5. Using the WHERE clause, the standing C-SPARQL queries serve to fetch potentially critical values from the incoming data stream. From this perspective, they can be seen as a filtering element whose responsibility is to let 'noisy' data pass through, keeping only critical triples passing on to the SWRL Reasoning Engine. Once the query is registered against the stream, the associated listener (i.e., streamFormatter) will start triggering every time RDF triples observed on the stream within the specified window frame satisfy the WHERE condition.

Listing 4.5: Registering a C-SPARQL query for the engine.

```
1 //Registering query
2 String query = "REGISTER QUERY invalid_value AS"
3              + "PREFIX ex: <http://myexample.org/>"
4              + "SELECT ?output ?value"
```

```
5               + "FROM STREAM <http://myexample.org/stream1> [RANGE 5s STEP
                    1s]"
6               + "WHERE { ?output msm:hasValue ?value"
7               + "?value > 50"
8               + "}";
9   CsparqlQueryResultProxy result = engine.registerQuery(query);
10
11  //Adding listener
12  RDFStreamFormatter streamFormatter = new RDFStreamFormatter("http://
        myexample.org/stream2");
13  proxy.addObserver(streamFormatter);
```

As illustrated in 4.5 which registers a query checking whether the observed value from sensors is greater than 50 and forwarding the invalid data to the observer, different types of critical situation check can be dynamically added to the SPARQL Query Engine by registering new queries. In fact, since the devices can vary in operating attributes, different techniques and parameters should be applied to the query in order to testing their operating status, thus resulting in various types of queries. The query results are processed into RDF triples representing the critical situations which will then be dynamically added to the Repository as ontology instances, and the reasoning process is initialized. As an example, after recognizing an abnormal value observed by a sensor from the query result, the Analysis component will constructing corresponding RDF triples to indicate potentially critical condition of the sensor, as illustrated in Listing 4.6. These RDF triple will then be used to update operating status of the corresponding devices. As an example, after detecting invalid value observed by a service, the Analysis component would constructing RDF triples to indicate the corresponding sensor is not working properly by changing its status to `OfflineState`, as illustrated in Listing 4.10.

Listing 4.6: Multiple RDF triples indicate a sensor has stopped working.

```
sai:sensor-13 rdf:type ssn:Sensor
sai:sensor-13 sai:hasId "TempSensor_1"
sai:sensor-13 msm:hasState sai:OfflineState
```

Depending on the set of registered SWRL rules, the application which is related to the newly-populated individuals may be classified as critical situations, thereby calling for a corresponding reactive action. As explained in previous section, same application may be classified into different types of Adaptation (i.e. ServiceUpgrade, ServiceSubstitution). As an example, Listing 4.7 presents a sample SWRL rules to classify a particular application into ServiceDisruption type. This rule checks whether the service `ser` which is used by application `app` is belong to a sensor with `OfflineState`, and updates the `app` to `ServiceDisruption` class as well as specifies the relation between service usage `usg` and `ser` to `ServiceDisrupted`.

Listing 4.7: An example of SWRL rule indicating a ServiceDisruption.

```
Application(?app), hasServiceUsage(?app, ?usg), usesService(?usg, ?ser),
Sensor(?dev), 'has service'(?dev, ?ser), 'has state'(?dev, ?state),
SameAs (?state, OfflineState)
-> ServiceDisruption(?app), ServiceDisrupted(?usg, ?ser)
```

The following listings below illustrate i) how an ontology is loaded and initiated within the SAI platform (see Listing 4.8), ii) how to dynamically modify different type of Repository (see Listing 4.9 and 4.10) and, finally, iii) how the Pellet reasoner, when queried, can infer if there are any instances of the class Adaptation (see Listing 4.11).

Listing 4.8: Loading and initializing the ontology.

```
1  OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
2  OWLOntology ontology = manager.loadOntologyFromOntologyDocument(IRI.
       create("file:" + ontologyFileName));
3  OWLReasonerFactory reasonerFactory = PelletReasonerFactory.getInstance();
4  OWLReasoner reasoner = reasonerFactory.createReasoner(ontology, new
       SimpleConfiguration());
5  OWLDataFactory factory = manager.getOWLDataFactory();
6  PrefixDocumentFormat pm = manager.getOntologyFormat(ontology).
       asPrefixOWLOntologyFormat();
7  pm.setDefaultPrefix(BASE_URL + "#");
```

Listing 4.9: Adding a new service individual into the knowledge base.

```
1  /*********************************************************************
2   * Adding new service individual to the knowledge base with name
3   * "ServiceName"
4   *********************************************************************/
5  OWLClass serviceClass = factory.getOWLClass("MSM:Service", pm);
6  OWLNamedIndividual newService = createIndividual(ontology, pm, manager,
       serviceName);
7  manager.addAxiom(ontology, factory.getOWLClassAssertionAxiom(serviceClass
       , newService));
8
9  OWLDataProperty hasIdDataProp = factory.getOWLDataProperty(":hasId", pm);
10 manager.addAxiom(ontology,factory.getOWLDataPropertyAssertionAxiom(
       hasIdDataProp, newService, serviceName));
```

Listing 4.10: Updating status of the devices.

```
1  /*********************************************************************
2   * Changing status of the the device specified by
3   * the parameter "device".
4   * "stateValue" indicates the value to change
5   * (0 = OfflineState, 1= OnlineState)
6   *********************************************************************/
7  public void ChangeStateOfDevice(OWLNamedIndividual device, int stateValue
       )
8  {
9
10    OWLNamedIndividual state;
11    if(stateValue == 0)
12       state = factory.getOWLNamedIndividual(":OfflineState", pm);
13    else
14       state = factory.getOWLNamedIndividual(":OnlineState", pm);
15
16    OWLObjectProperty hasStateProperty= factory.getOWLObjectProperty("
          DOGONT:hasState", pm);
17
18     for (OWLNamedIndividual ind : reasoner.getObjectPropertyValues(device
          , hasStateProperty).getFlattened())
```

```
19      {
20          manager.removeAxiom(ontology, factory.
                getOWLObjectPropertyAssertionAxiom(hasStateProperty, device,
                ind));
21      }
22
23      manager.addAxiom(ontology, factory.getOWLObjectPropertyAssertionAxiom
            (hasStateProperty, device, state));
24  }
```

Listing 4.11: Reasoning whether there are adaptation instances.

```
1  //Querying over all applications existing in the platform
2  for (OWLNamedIndividual app : reasoner.getInstances(applicationClass,
       false).getFlattened())
3  {
4      String appName = app.toStringID();
5
6      //Checking whether the application is also an adaptation class (e.g.
           need adaption)
7      OWLClassAssertionAxiom axiomToExplain = factory.
           getOWLClassAssertionAxiom(adaptationClass, app);
8      boolean needAdapting = reasoner.isEntailed(axiomToExplain);
9
10     //if there is new adaptation
11     if(needAdapting)
12     {
13         //Checking type of adaptation
14         OWLClassAssertionAxiom axiomToExplain1 = factory.
               getOWLClassAssertionAxiom(serviceUpgradeClass, app);
15         OWLClassAssertionAxiom axiomToExplain2 = factory.
               getOWLClassAssertionAxiom(serviceSubstitutionClass, app);
16         OWLClassAssertionAxiom axiomToExplain3 = factory.
               getOWLClassAssertionAxiom(serviceDisruptionClass, app);
17
18         //Service upgrade
19         if(reasoner.isEntailed(axiomToExplain1))
20         {
21             //Corresponding processing for service upgrade
22         }
23         //Service substitution
24         else if(reasoner.isEntailed(axiomToExplain2))
25         {
26             //Corresponding processing for service substitution
27         }
28         //Service disruption
29         else if(reasoner.isEntailed(axiomToExplain3))
30         {
31             //Corresponding processing for service disruption
32         }
33
34     }
```

The final step of the Autonomic Manager is carried out through the **Application Executer** which is part of the Execute component. At the moment, this component is responsible for collecting all information about the adaptation plan which is necessary for new reconfiguration, transforming those information into RDF triples, and sending

them to the corresponding application. For example, whenever the reason specifies an application as an instance of ServiceUpgrade, the Application Executer will then identified the service which is being used as well as the respective service which the application should upgrade to. This process can be implemented in the code snippet presented in Listing 4.12. Accordingly, the corresponding RDF triples will be constructed based on this information and communicated to the application for reconfiguration. A simplified list of RDF triples indicating a `ServiceUpgrade` adaptation is illustrated in Listing 4.13. Specifically, these triples notify the application to upgrade from `service-10` to `service-15` provided by the sensor with id `TempSensor_15`.

Listing 4.12: Finding corresponding service to upgrade to.

```
1
2  //Finding all services used by the application app
3  for (OWLNamedIndividual usage : reasoner.getObjectPropertyValues(app,
       hasServiceUsageProperty).getFlattened())
4  {
5     OWLNamedIndividual fromService, toService;
6
7     //Finding the the currently used service
8     for (OWLNamedIndividual service : reasoner.getObjectPropertyValues(
          usage, usesToProperty).getFlattened())
9      {
10       fromService = service;
11       break;
12     }
13
14     //Finding the new service to upgrade
15     for (OWLNamedIndividual service : reasoner.getObjectPropertyValues(
          usage, upgradeToProperty).getFlattened())
16     {
17       toService = service;
18       break;
19     }
20
21 }
```

Listing 4.13: RDF triples represent a ServiceUpgrade Adaptation.

```
sai:sensor-15 rdf:type ssn:Sensor
sai:sensor-15 sai:hasId "TempSensor_15"
sai:sensor-15 msm:hasService msm:service-15
sai:sensor-15 msm:hasId "TempSensingService_15"
msm:service-15 msm:hasOperation msm:operation-15
msm:service-15 msm:hasAddress "https://localhost:80/TempSensor/Read"
.
.
.
sai:usage-3 sai:useService msm:service-10
sai:usage-3 sai:upgradeTo msm:service-15
```

4.2.4 The SAI Application

Once receiving the reconfiguration plan under RDF triples from the Application Executer, the deployed application starts reconfiguring according to these plan. As discussed from previous section, SAI application is described as a set of rules in order to simply the process of semantic analyzing and reconfiguration. At the moment, a SAI application would be specified according to the syntax illustrated in Listing 4.14.

Listing 4.14: Syntax of the SAI application.

```
//Application definition
PROGRAM ->  "begin" BLOCK "end"

//Application body
BLOCK   ->  "rule" <STRING_LITERAL>
            "use" USE_BL
            "declare" DEC_BL
            "action" ACT_BL

//Defining sensor or actuator for use
USE_BL  ->  (USE_SEN)* | (USE_ACT)*
USE_SEN -> "sensor" <ID> "@" <ID> ["for" <ID>] ";"
USE_ACT -> "actuator" <ID> "@" <ID> ["for" <ID>] ";"

//Declaring variable
DEC_BL   ->  ("var" <ID> ";")*

//Defining rule
ACT_BL   ->  (STMT)*

//Statement can be assignment, if statement or service call
STMT    ->  (ASS_ST)* | (IF_ST)* | (SER_CAL ";")*

//Variable can be assigned value of other variable, literal,
//service call and property of devices
ASS_ST  ->  <ID> "=" EXP ";"
EXP     ->  <ID> | <LITERAL> | SER_CAL | PROP_ACC
SER_CAL ->  <ID> "." <ID>  "(" [<ID> | <LITERAL>] ")"
PROP_ACC->  <ID> "." <ID>

//If statement
IF_ST   ->  "if" "(" CONT ")" "{" ACT_BL "}"
            ["else" "{" ACT_BL "}" ] ";"

CONT    ->  EXP <OP> EXP
```

Listing 4.15 presents the Fire Alarming application which observes the value from a smoke sensor and turns on the speaker whenever fire detected. As illustrated, A SAI application is typically described with `begin` and `end` keywords along with four compulsory block:

- The **Rule** block describes the name of the application. Currently, an application consists of only one rule. However, the rule may provide multiple functions which are defined in the Action block.
- The **Use** block specifies the sensors or actuators which would be used in the application along with their ids (e.g. after the @ sign) which were defined and stored in the Devices Repository. The statements in this block could also be used to defined a specific usage context of the devices, which as explained earlier will be used for approximating services. As an example the statement in line 10 of Listing 4.15 declares the application would use the actuator `Speaker1` for `NotificationContext`.
- The **Declare** block for declaring variables used inside the Action block.
- The **Action** block describes the functions of application. Typically, these functions are presented under conditional statements which trigger actions by service calls. This block could also contain assignment blocks and device properties access. At the moment, the names of services or properties are manually mapped to the ids of corresponding instances existing in the ontology, thus calling services or accessing devices properties is actually retrieving the data value of those instances. For example, in line 16 of Listing 4.15, the value of `Speaker1PowerMaxValue` instance related to `Speaker1` is assigned to variable `speaker_freq`. In addition, whenever encountering line 18, the interpreter actually accesses the URL value of the Service with id `SmokeSensor1GetService` in order to make a web service call.

Listing 4.15: Rule used in the Fire Alarming Application.

```
1  /*********************************************************************
2   *  Rule to observe the air quality value measured by the SmokeSensor1
3   *  and control the speaker by setting the input frequency to 2000Hz
4   *  whenever the observed value exceeds the threshold.
5   *********************************************************************/
6  begin
7     rule "fire_alarming_operation"
8     use
9        sensor smoke_sensor@SmokeSensor1;
10       actuator speaker_act@Speaker1 for NotificationContext;
11    declare
12       var alarming_thereshold;
13       var speaker_freq;
14    action
15       alarming_thereshold = smoke_sensor.SmokeSensor1Threshold;
16       speaker_freq = speaker_act.Speaker1PowerMaxValue;
17
18       if(smoke_sensor.SmokeSensor1GetService() > alarming_thereshold)
19       {
20          speaker.Speaker1SetService(speaker_freq);
21       }
22 end
```

Once deployed, the application would send necessary information to the Autonomic Manger for monitoring and reasoning purpose. This information should be corresponding to the Application Model defined in the ontology and transformed into RDF triples. As an example, Listing 4.16 presents partial information would be sent by the Fire Alarming application in Listing 4.15.

Listing 4.16: RDF triples represent an application update.

```
sai:application-20 rdf:type sai:Application
sai:application-20 sai:hasServiceUsage sai:usage-34
sai:usage-34 sai:usesService sai:service-123
sai:service-123 sai:hasId sai:SpeakerSetService
sai:usage-34 sai:hasContext sai:NoficationContext
....
```

After receiving information about the newly deployed application, the Monitoring component would update the knowledge base accordingly for monitoring and reasoning adaptation plan. As discussed in previous section, these adaptations are also transfered under RDF triples and processed by the ApplicationMonitor in oder to reconfigure the application by modifying its rules. As an example, Listing 4.17 described the Fire Alarm application after an ServiceSubstitute adaptation. Specifically, instead of using `SpeakerSetService` of the `Speaker1` as in Listing 4.15, the application changes to use service provide by the `LampActuator1` which also supports the same `NotificationContext`.

Listing 4.17: Fire Alarming Application after reconfiguration.

```
1  /*****************************************************************
2   *  Rule to observe the air quality value measured by the SmokeSensor1
3   *  and control the lamp actuator accordingly (e.g. turn on the lamp)
4   *  whenever the observed value exceeds the threshold.
5   *****************************************************************/
6  begin
7     rule "fire_alarming_operation"
8     use
9        sensor smoke_sensor@SmokeSensor1;
10       actuator lamp_act@LampActuator1 for NotificationContext;
11    declare
12       var alarming_thereshold;
13       var lamp_power;
14    action
15       alarming_thereshold = smoke_sensor.SmokeSensor1Threshold;
16       lamp_power = lamp_act.LampAct1PowerMaxValue;
17
18       if(smoke_sensor.SmokeSensor1GetService() > alarming_thereshold )
19       {
20          lamp_act.LampAct1SetService(lamp_power);
21       }
22 end
```

At the moment, the prototype support relatively limited features in describing logics for application. It should be enhanced along with the Application Model in the ontology in oder to support complicated adaptation situation. We will discuss this direction further more in Chapter 7.

Chapter 5

The Smart Home Case Study

In previous chapter, we have explained the conceptual architecture and briefed the reader on technical aspects of the platform, which are also required to understand the case study and experiments to be described in this chapter. Specifically, the chapter explained how the main three components of the platform – the SAI ontology, the Autonomic Manger, and the Application Model – are implemented. In this chapter, we will explain in more detail how the autonomic mechanism is actually carried out within the platform by demonstrating a case study focusing on the smart home situations. Specifically, this chapter considers a case study, which involves multiple applications usually found in a smart home, which reconfigure their services according to changes in the operating environment. The use case scenarios are described from two sides of the architecture - namely, the platform and the application perspectives. With the use case scenario explained, we will be able to describe each of the autonomic mechanism carried out in each component of the platform. Accordingly, the use case demonstrates how raw data is first transformed into RDF, then queried with C-SPARQL queries, reasoned by the SWRL Reasoning Engine, and eventually sent to applications for reconfiguration.

Overall, the case study involves a house consisting of two rooms where all of the devices are located. Figure 5.1 presents the architecture of this house which we use to illustrate the case study. In *Room 1*, there are a *Smoke Sensor*, a *Speaker* and a *Lamp* connected to a switch which could be controlled remotely. *Room 2* has a *Temperature Sensor*, a *Heater*, a *Luminous Sensor* which measures the ambient light intensity and a *Bulb* connected to a dimmer. In this case study, we assume that all gadgets provide their accesses through web services hosted on the two gateways which have connection with a server running our SAI platform. There are also smart devices on which the applications are deployed. Specifically, there are five applications running on a computer and a smart phone:

- The *Lamp Controlling Application* which is deployed on the smart phone, is used to control the lamp in Room 1. Listing 5.1 describes the logic of this application. Specifically, the application switches the lamp on/off whenever receiving a button pressed signal.

- The *Fire Monitoring Application* in Listing 5.2 simply reads value observed by the smoke sensor and outputs to the console.

- The *Light Controlling Application* which is running on the computer, is described in Listing 5.3. This application uses the luminous sensor and bulb in Room 2 in order to

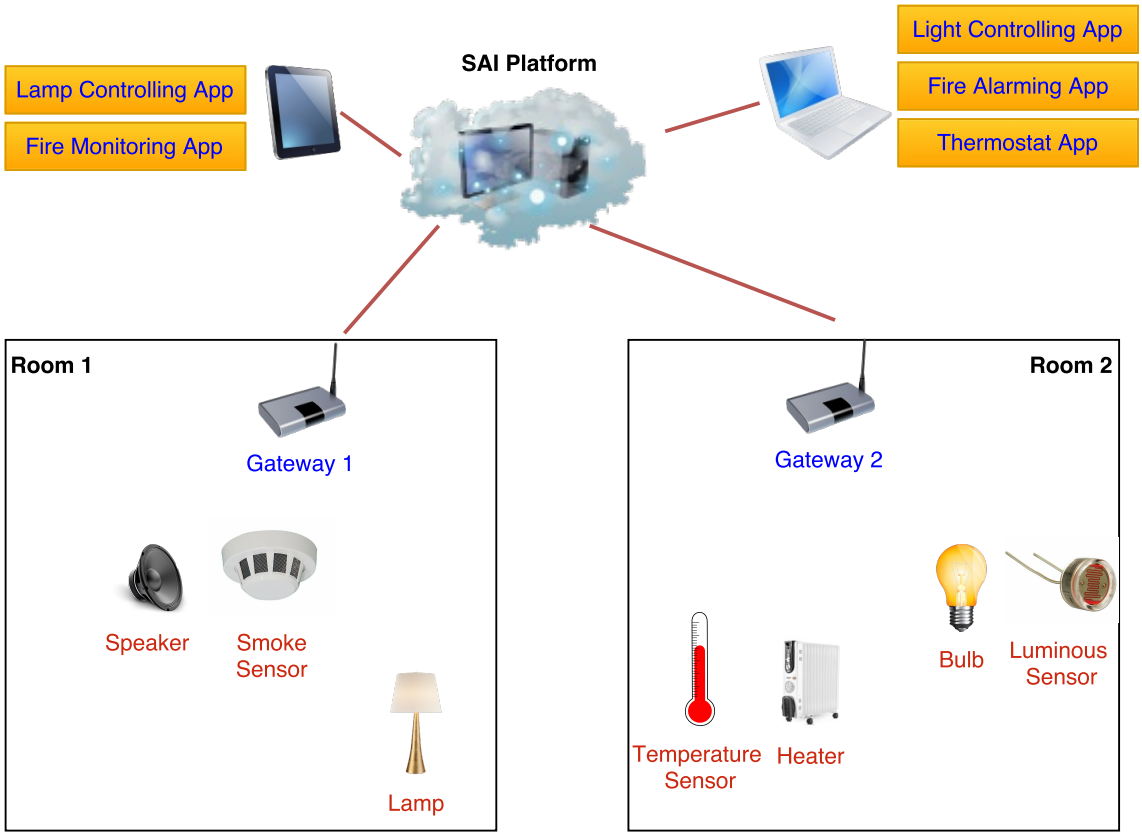

Figure 5.1: Overview of physical arrangement in the Smart Home Case Study.

maintain the light intensity around desired value. Specifically, the application keeps comparing the sensor value with the threshold in order to decide whether to increase or decrease power of dimmer connected to bulb.

- The *Thermostat Application* operates in a similar way as the Light Controlling Application. It measures the temperature of Room 2 and controls the heater accordingly. The rule describing this application was already presented from previous chapter in Listing 4.1.

- The *Fire Alarm Application* in Listing 4.15 observes air quality measured by the smoke sensor and turn the speaker on whenever encountering value greater than a threshold.

Listing 5.1: Lamp Controlling Application.

```
1 /*************************************************************************
2  *  Rule to detect button pressed and turn the lamp actuator
3  *  accordingly.
4  *************************************************************************/
5 begin
6    rule "lamp_control_operation"
7    use
```

```
8         sensor button@Button1;
9         actuator lamp_act@LampActuator1;
10    declare
11        var lamp_power;
12    action
13
14        //turn off the light
15        lamp_power = lamp_act.LampAct1PowerMinValue;
16        lamp_act.LampAct1SetService(lamp_power);
17
18        if(button.Button1GetService() == "button_pressed")
19        {
20            //Switching the actuator
21            if(lamp_power == lamp_act.LampAct1PowerMinValue)
22            {
23                lamp_power = lamp_act.LampAct1PowerMaxValue;
24            }
25            else
26            {
27                lamp_power = lamp_act.LampAct1PowerMinValue;
28            }
29            lamp_act.LampAct1SetService(lamp_power);
30        }
31 end
```

Listing 5.2: Fire Monitoring Application.

```
1  /*********************************************************************
2   *  Rule to read input from smoke sensor, output the value to console.
3   *********************************************************************/
4  begin
5     rule "fire_monitoring_operation"
6     use
7        sensor smoke_sensor@SmokeSensor1;
8     declare
9        var sensor_output;
10    action
11       sensor_output = smoke_sensor.SmokeSensor1GetService();
12       print sensor_output;
13 end
```

Listing 5.3: Light Controlling Application.

```
1  /*********************************************************************
2   *  Rule to detect the intensity of the ambient light and adjust the
3   * luminous actuator accordingly.
4   *********************************************************************/
5  begin
6     rule "lam_regulating_operation"
7     use
8        sensor luminous_sensor@LuxSensor1;
9        actuator lumious_act@LuxActuator1;
10    declare
11       var dire_lux_thershold;
12       var lux_input;
13       var step;
14    action
```

```
15        dire_lux_thershold = 600;
16        lux_input = 600;
17        step = 2;
18
19        if(luminous_sensor.LuxSensorGetService() < dire_lux_thershold)
20        {
21           lux_input = lux_input + step;
22        }
23        else
24        {
25           lux_input = lux_input + step;
26        }
27
28        luminous_act.LuxActatorSetService (lux_input);
29  end
```

It is necessary to explain that this case study was tested in a simulating manner, which means we did not work with the real gadgets but simulated them instead. Specifically, for each device presented above, we created corresponding RESTful APIs which return our desired values when being called. By doing so, we could not only reduce effort of collecting raw data at the first stage since the APIs were designed to return their values in format conforming to the ontology but also simplify the process of making changes to the operating environment observed by the Autonomic Manager, thus critical situations could be simulated easily. In particular, we produced the device outputs in a way to generate the three presented adaptation types, which will be discussed in more details in following scenarios:

- **Upgrading to new luminous sensor:**

  This scenario involves the *Light Controlling Application* in Listing 5.3 which controls the light intensity of Room 2. Specifically, the application was designed to use the luminous sensor with id `LuxSensor1` from the beginning. In order to to generate a `ServiceUpgrade` adaptation, we integrated a new luminous sensor with higher resolution. This could be done by manually inserting new instance of sensor into the ontology, which means we created new sensor that is acting on the same luminosity property, locating in Room 2 and having higher range of sensing capabilities. Listing 5.4 presents the SWRL which is used for reasoning about this situation. Specifically, the rule searches for an application `app` which is currently using service provided by a sensor `device1` which has upper limit of sensing capability less than another sensor `device2` which has same type `property` and location `loc`.

Listing 5.4: An example of SWRL rule indicating a ServiceUpgrade.

```
Application(?app), hasServiceUsage(?app, ?usg), usesService(?usg, ?ser1),

Sensing_device(?device1), hasLocation(?device1, ?loc),
'has service'(?device1, ?ser1), observes(?device1, ?property),
hasSensingCapability(?device1, ?cap1), hasSensingProperty(?cap1, ?prop1),
hasRegion(?prop1, ?reg1), hasMaxValue(?reg1, ?max1)

Sensing_device(?device2), hasLocation(?device2, ?loc),
'has service'(?device2, ?ser2), observes(?device2, ?property),
hasSensingCapability(?device2, ?cap2), hasSensingProperty(?cap2, ?prop2)
```

```
hasRegion(?prop2, ?reg2), hasMaxValue(?reg2, ?max2),

swrlb:lessThan(?max1, ?max2)
-> ServiceUpgrade(?app), UpgradeTo(?usg, ?ser2)
```

Whenever the rule was matched, the SWRL Reasoning Engine would indicate a ServiceUpgrade plan to the corresponding application for reconfiguration as discussed in previous section. As an example, Figure 5.2 presents an output from the platform whenever an application needs adapting.

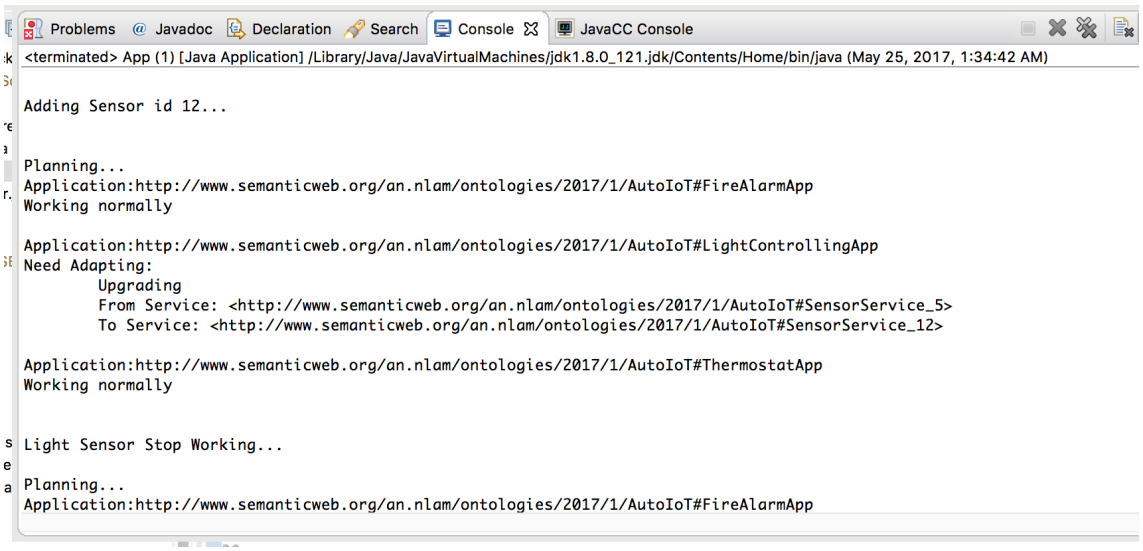

Figure 5.2: A snapshot of output from the reasoning.

- **Substituting lamp for speaker:**

  This scenario presents an example of approximating services by using SWRL rule. Particularly, the platform tries to use the lamp in Room 1 as an alternative method for alarming in case of fire whenever the speaker stops working. As discussed in last chapter when we explained details of the SAI application, the *Fire Alarming Application* in Listing 4.15 uses the service provided by `Speaker1` for `NotificationContext`. Whenever, the speaker stops working, the platform keeps searching for other alternatives which support the same context. In order to simulate a case of losing control of speaker, we provided values indicating a failure whenever the platform called services provided by the speaker for checking its status. As described earlier, in this case, the RDF Standardization Engine would generating RDF triples presenting the actuator with `OfflineState`. As an example, Figure 5.3 presents sample RDF outputs from this engine.

  As presented, these RDF triples will be served as input to the SPARQL Query Engine for filtering critical situations. Listing 5.5 described the C-SPARQL used for selecting triples which containing information about an offline actuator. Specifically, the query keep analyzing over the RDF stream and selecting only actuators which have state `OfflineState`. Upon receiving result from this query, the corresponding

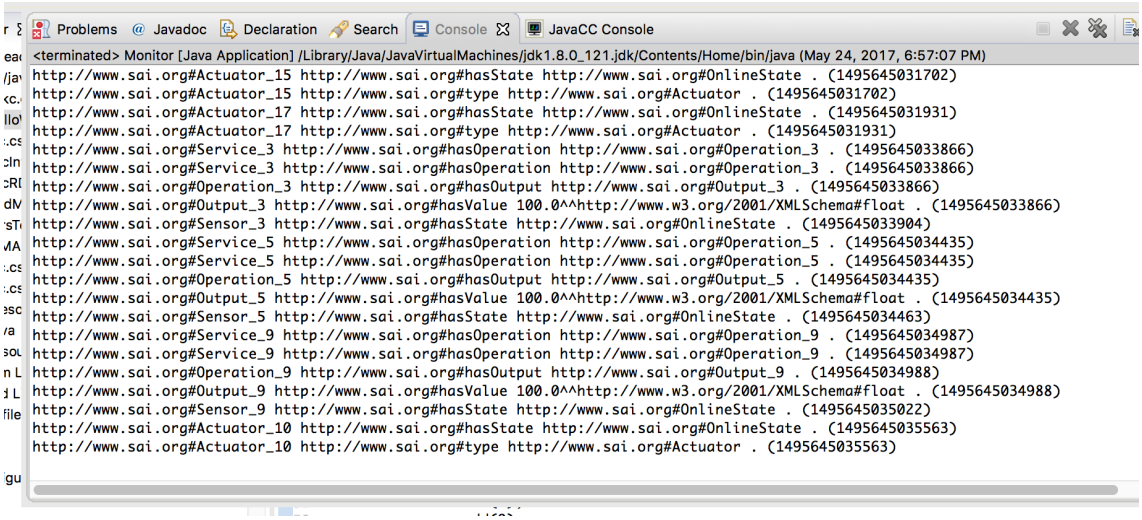

Figure 5.3: A snapshot of output from the monitoring.

status update to the respective instances of actuators will be carried out by calling functions presented in Listing 4.10.

Listing 5.5: C-SPARQL query for offline actuator.

```
REGISTER QUERY OfflineActuator AS
PREFIX sai: <http://www.sai.org#>
SELECT ?actuator "
FROM STREAM <http://www.sai.org/invalidvaluestream> [RANGE 5s STEP 1s]
WHERE {
      ?actuator sai:type sai:Actuator .
      ?actuator sai:hasState sai:OfflineState .
}
```

Once the actuator status was updated into the ontology, the SWRL Reasoning Engine would be able to find a new equivalent service. Listing 5.6 describes the SWRL rule for reasoning `ServiceSubstitution` adaptation. In particular, whenever an actuator stops working, the rule keeps searching for another actuator which provides service with the same context, and indicates the application for substituting to the new service.

Listing 5.6: An example of SWRL rule indicating a ServiceSubstitution.

```
Application(?app), hasServiceUsage(?app, ?usg), usesService(?usg, ?ser),
hasContext(?usg, ?context),

Actuator(?dev1), 'has service'(?dev1, ?ser), hasLocation(?dev1, ?loc)
hasState(?dev1, ?state),  SameAs (?state, OfflineState),

Actuator(?dev2), 'has service'(?dev2, ?ser2), hasLocation(?dev2, ?loc)
usedInContext(?ser2, ?context)
```

```
-> ServiceSubstitution(?app), SubstitutionTo(?usage, ?ser2)
```

Accordingly, the reconfiguration process would be starting at application upon receiving the adaption plan from the platform. Figure 5.4 shows an example of output from the application after reconfiguring.

Figure 5.4: A snapshot of output from an application.

- **Temperature sensor malfunctioning:**

  This scenario presents a situation when the platform could not find any suitable service for application to replace whenever the currently used service disrupted. To illustrate the scenario, we simulate a temperature sensor which provides invalid value. Specifically, we provided relatively large value to the calls to retrieve value observed by the temperature sensor in Room 2. As discussed, this case should be detected as a critical situation by the Analysis component. To do that, we inserted a query that detects invalid observed value from the sensor to the SPARQL Query Engine, as illustrated in Listing 5.7. Specifically, the query keeps analyzing over the RDF stream, focuses only on value greater than 60 degree, and finally outputs the triple of sensor, service and value for further investigation. Figure 5.5 present an example of output of this query from the engine for this case.

Listing 5.7: C-SPARQL query for invalid sensor value.

```
REGISTER QUERY InvalidSensorValue AS
PREFIX sai: <http://www.sai.org#>
SELECT ?sensor ?service ?value
FROM STREAM <http://www.sai.org/invalidvaluestream> [RANGE 5s STEP 1s]
WHERE {
      ?sensor sai:type sai:Sensor .
      ?sensor sai:hasService ?service .
      ?service sai:hasOperation ?op .
      ?op sai:hasOutput ?out .
      ?out sai:hasValue ?value .
      FILTER (?value > 60)
}
```

Figure 5.5: A snapshot of output from the analysis

Upon receiving the query results, the Analysis would update the ontology accordingly. Specifically, the same action as in previous scenario is carried out, which is updating the sensor status to `OfflineState`. Since then, the SWRL Query Engine should be able to detect a `ServiceDisruption` situation. As explained earlier, this adaptation is supported by the SWRL rule described in Listing 4.7. Particularly, the rule checks whether a service which is currently occupied by an application is provided by an offline device. If so, the application would be classified into `ServiceDisruption` class. Accordingly, the Execute component would send this information to the corresponding application which is the *Thermostat Application* in this case for further action. At the moment, upon receiving this kind of adaptation, the application would do nothing rather than stop switching the heater on/off in oder to protect it from damage due to overheating and wait for alternative plan from the platform.

To summarize, in this chapter, we have demonstrated how the SAI platform operates by presenting a case study of a Smart Home. This case study focuses on the five applications which were deployed on two smart devices (a laptop and a smart phone) and connected to different services provided by the gadgets through RESTful APIs. The main goal of the considered use case scenarios was to demonstrate the analysis and reasoning capabilities of the platform as well as the reconfiguration mechanisms carried out at the applications. Additionally, by using code snippets, the chapter demonstrated how the main components function, and how monitored data is first transformed and then flows within whole system. These scenarios will also be used for setting up experiments for further evaluation of our approach. Specifically, in the next chapter, we will continue with the presented scenarios in order to evaluate the performance of the system from perspectives of the two main stake-holders: the SAI platform and the SAI applications.

Chapter 6

Evaluation and Discussion

In this chapter, we summarize the main aspects of the presented approach in a more structured manner. First, we evaluate and discuss our approach under the perspective of performance capabilities. Next, we continue the discussion with a detailed explanation about the proposed platform with respect to the problem statements presented in the introductory chapter.

6.1 Evaluating performance of the approach

In this section, we evaluate the performance of our proposed approach under two perspectives: the SAI platform and SAI application. Accordingly, we conducted several experiments aiming at demonstrating the performance of the whole system with respect to several configurable parameters. As mentioned earlier, our experiments were conducted based on the Smart Home Case Study presented in Chapter 5. Specifically, we changed the physical environment of Smart Home by simulating the value observed by the sensors and actuators in order to trigger the reconfiguration in each scenario and measured several interesting metrics accordingly in order to assess the performance of both the platform and the underlying process of applications. Based on the results of each experiment, we discuss the benefits of employing our solution for achieving autonomic capabilities into IoT platform. Furthermore, we also present the threats to validation as well as potential limitations of the approach.

6.1.1 Experimental Setup

All the experiments were performed on a workstation with a CPU Intel Core I5 2.70 GHz processor and 8 GB memory running Sun J2RE 1.8. We set the JVM maximum allocation pool to 2 GB, so that virtual memory activity has no influence on the results. As explained earlier, in order to trigger the adaptation from each scenario in the Smart Home Use Case, we simulated the physical devices by hosting our own RESTful APIs on a local computer, thus we were able to take control over the values of sensors or actuators that were read from the Monitor component and the applications. It is worth explaining that in the experiments, the Monitor component monitored each physical device periodically, thus the RDF triples were generated at relatively stable rate. Particularly, throughput of the RDF stream fed to the SPARQL Query Engine is approximately 20 triples/sec. Accordingly, we injected critical values into the stream in order to trigger each scenario sequentially

and measured metrics interesting to the performance evaluations at two sides: the SAI platform and the applications.

6.1.2 Performance of the SAI platform

The SAI platform was evaluated under the aspect of scalability, which is testing processing performance of platform on system which involves vast number of devices. Specifically, for each scenario of the use case, we introduced additional devices into the system by creating corresponding instances related to them into the knowledge base. Accordingly, the following metrics were measured for this evaluation:

- **Memory Utilization** is the maximum RAM memory occupied by the platform during the runtime of the scenarios. This value was measured by using the YourKit Java Profiler [1] tool.
- **Detection Time** of the scenario is the duration from the moment critical values are injected to the Monitor to the time when adaptation plans are ready to be sent to the applications.

The experiment was designed as follows. Every scenario was reproduced ten times and the average value from all runs were used in order to increase the reliability of the measured data. Additionally, for each scenario, apart from the active gadgets, we generated $(10^x | x = 0..4)$ different devices and measured the metrics of interest from each step.

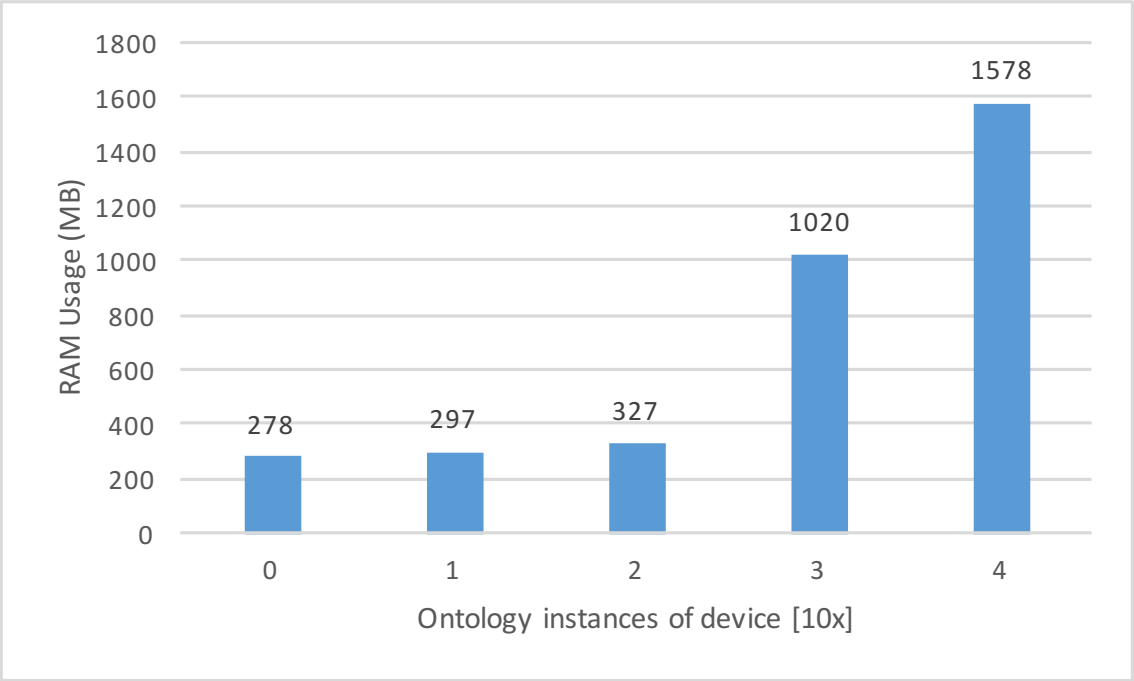

Figure 6.1: Memory utilization of the SAI Platform with additional instances.

Figure 6.1 presents the results of memory utilization from the experiment. As can be seen from this figure, the memory allocation tends to increase slower than linear trend.

[1] https://www.yourkit.com

Specifically, the number is around 300MB when there are up to 100 devices existing in the system. This figure is approximately tripped when the number of devices increases ten times and raises up to 1.5GB with 10^4 devices. It is worth explaining that in order to introduce a device into the system, multiples type of instances should be created in the knowledge base in accordance with the presented SAI ontology (e.g. the services or properties related to that particular device). Therefore, for one device, probably more than ten individuals of ontology class are created. Additionally, the numbers presented in Figure 6.1 do not present the memory allocation for only knowledge base but the entire autonomic platform. Since our approach applies in-memory stream processing and reasoning, this metric could be critical to the platform whenever the system grows larger. Therefore, these numbers illustrate that the presented use case can be arguably compared to real-world IoT systems consisting of thousands of sensors and actuators, thus demonstrating the efficiency of applying the presented ontology for knowledge representation in term of physical resource consumption.

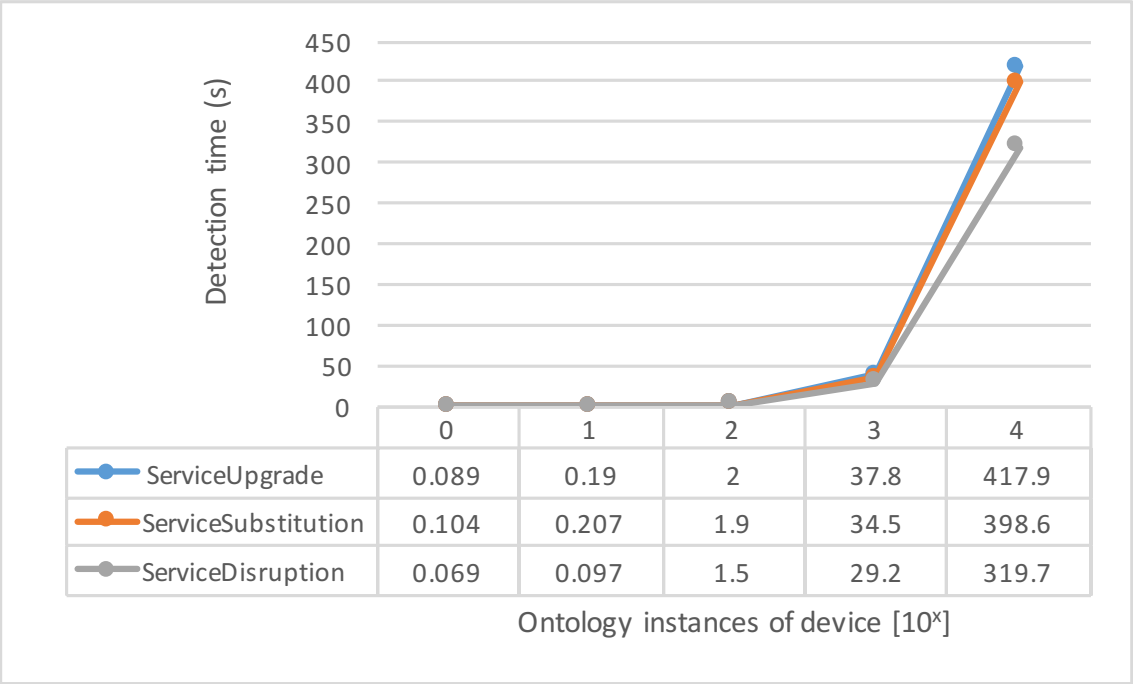

Figure 6.2: Autonomic performance of the SAI Platform with additional instances.

Regarding the detection time metric, as can be seen from Figure 6.2 which presents the measured values from three different types of adaptation, we can conclude with the following observation - as the workload is increasing, the platform is facing considerable challenges, which stem from the volume and variety of the collected data. Specifically, the numbers in three scenarios have the same increasing tendency, which shows that the platform was able to detect each critical situation within two seconds when there are up to 100 devices. The reasoning slows down the platform significantly if there are more than 10^3 devices existing. Additionally, the numbers from *ServiceDisruption* are relatively smaller than the other two adaptation plans. This could be intuitively explained by the fact that the corresponding SWRL rule describing this situation has less predicates than the others,

thus taking the engine smaller time for matching the rule. Furthermore, the evaluation of the *ServiceUpgrade* took an average of 37.8 seconds for 10^3 and 417.9 seconds for 10^4 devices, which are also the biggest number observed from the experiment.

It is important to explain that Figure 6.2 does not illustrate the efficiency of the platform in term of providing services to the consumers. Thus, our proposed platform is still comparative to other IoT platforms with regard to support for real-time applications. Specifically, as explained in Chapter 4, once deployed into the system, SAI applications work directly with the devices through web services, thus, the network latency is the primary factor in that case. In this experiment, this figure presents performance of the autonomic mechanism provided by the platform, that is how frequently the platform produces an adaptation plan to its client, which could be considered as an additional intelligent service provided by the platform. As discussed from the beginning, autonomic computing is introduced to support system evolution scenarios which required significant human effort. In practice, each of the presented adaptations can be carried out by the platform in minutes compared to hours of working from the system operators. In addition, human effort required could be increasing with the number of adaptations or applications while they could be executed by the platform in parallel manner. Therefore, with this perspective, the performance presented in Figure 6.2 could be acceptable since the autonomic capabilities could be considered as an extra feature supported by the platform. Furthermore, in real-world context of IoT system (e.g. smart home, smart city), critical situations may rarely occur but take longer time to be fixed by the administrator (e.g. user need to report about a broken fire alarm and wait some days for replacement), thus our autonomic adaptation could be consider as an efficient alternative in the mean time.

It is worth explaining that the reasoning process of in the SWRL Reasoning Engine is implemented in a sequential approach. That is, the engine queries over the applications sequentially, matching rules for each type of adaptations is also done in the same way, which could affect the detection performance of the engine. Therefore, appropriate actions have to be taken to address this issue. In fact, this could be done simultaneously for each application. We will discuss the possibilities of this improvement in more details in next chapter.

6.1.3 Reconfiguration at the SAI application

As IoT applications could be deployed on different frameworks which have different physical specifications, it is important to evaluate the SAI applications under the perspective of resource consumption. In addition, the cost of reconfiguration is also critical to the performance of the applications. Specifically, the applications are expected to quickly adapt to changes from their operating environment. Accordingly, under perspective of the consumer, we assessed the performance of underlying autonomic mechanism of the SAI application with these two metrics:

- **Memory Utilization** is the maximum RAM memory occupied by the application during the runtime of the scenarios. This value was also measured with the YourKit Java Profiler tool.
- **Reconfiguration Time** is the duration from receiving adaptation plan to reconfiguring completed.

Table 6.1 presents the results of this experiment. As can be seen from the table, every

Application	Memory Utilization (MB)	Reconfiguration Time (ms)
Light Controlling	61	108
Lamp Controlling	59	-
Thermostat	54	86
Fire Alarming	51	117
Fire Monitoring	57	-

Table 6.1: Cost of reconfiguration in each application.

application allocated relatively low RAM, which is around 60MB. Regarding reconfiguration time, the *Light Controlling* and *FireAlarming* executed adaptations in approximately 100 ms, while the respective figure for the *Thermostat* is about 86 ms. This could be explained by the fact that the *Thermostat* simply stops running upon receiving *ServiceDisrupted* plan. In general, these numbers are still within the range acceptable under the context of IoT application.

It is necessary to remind that our solution to the SAI application is still immature and does not cover every functionalities required in IoT context, and that currently implementation of SAI applications is only for demonstrating the presented use case. Therefore, these numbers may vary when the system fully supports real-world requirements. However, this figure can still be used to demonstrate the potential of our approach toward modeling IoT applications that support autonomic behaviors. Additionally, the testing also shows a smooth cooperation between the platform and the application, which demonstrates the effectiveness of our method. In next chapter, we will discuss more about improving application modeling toward increasing semantic understanding in order to support complicated adaptation.

6.1.4 Threats to validity

In this section, we discuss possible shortcomings of the process of validation presented above. Subsequently, we also explain how our results are still arguable representative for the presented validations as well as discuss possible directions for improvements.

The number of C-SPARQL and SWRL queries is not significant
In the presented use case, we primarily focus on the workload putting on autonomic engine in form of ontology instances, while there are also other impact factors of the reasoning process. As an example, additional number of C-SPARQL queries as well as the stream throughput could slow down the process of analysis. At the moment, we stabilize rate of the input stream, which could possibly delay the process of detecting potentially critical situation. In fact, this deficiency in validation was already covered in [18], which mainly tested the performance of the C-SPARQL engine under high throughput. The experiments in this work showed positive results on the performance of their approach, which generated the output within 7 seconds with a stream of 10000 triples/second. Therefore, it is possible to integrate the engine into our platform with minimum effect.

Regarding the SWRL rules, the use case consists of six SWRL rules corresponding to the three adaptation plans. In real-world system, this number would be increasing gradually. Furthermore, SWRL rules could be even more complicated with enormous predicates. In this case, scalability of the platform would be challenged by the emerging Big Data,

and appropriate actions have to be taken to address this issue. As mentioned above, this could be consider as a trade-off between human effort and autonomic support. In fact, optimization in the implementation of SWRL Reasoning Engine could be employed in order to improve the performance of platform. We will discuss this issue as a future direction in the next chapter.

Immature implementation of the applications
As discussed in last section, the implementation of underlying autonomic mechanism in SAI application is simplified to support demonstration of the use case as a proof of concept. Therefore, the presented figure in this experiment could be changed but expected to not vary considerably when the system fully satisfies real-world contexts. Regarding this issue, there are also possible improvements of application modeling toward reflecting real-world requirements, which will be discussed shortly in the next chapter.

Additionally, it is also necessary to explain that the knowledge base stored centrally in the platform should be populated to the application side so that developers could choose the correct service call signature or property name while programing applications. It is also important to the process of reconfiguration. The reason was already explained earlier in Chapter 4, which is the interpreter directly uses ids of the instances of services/properties in the ontology for naming the corresponding services/properties used in the rules; thus, whenever encountering a service call or property access, it simply retrieves data value stored in the corresponding instances with the same name in the knowledge base. At the moment, each application in the use case stores a copy of ontology file which is being used by the platform. Since the use case involves not many entities, this solution does not affect the figure found in Table 6.1. However, possible improvements (e.g. storing only necessary information for each application) could be employed for optimization purpose. We will discuss this direction later in the next chapter.

6.2 Discussing problem statements

In the presented work we raised the issue of evolution in IoT system, which is becoming of utmost importance, as IoT platforms compete, striving to deliver even wider selection of services and accommodate even more user applications. It is our belief that platform providers should enable their platforms with self-management capabilities since these systems require enormous human effort for such manual managements as they grow into large scale. In the first instance, such self-governance capabilities are expected to enable platforms with more control over their constantly growing software ecosystems to support platform stability and optimal resource consumption. Furthermore, these capabilities are intended to exempt operators from implementing this functionality themselves, and provide more intelligence into how the applications behave and perform. In this light, in the introductory chapter of this thesis we raised several research questions we aimed to address. We answered these questions with our proposed approach of the Service-oriented Autonomic IoT (SAI) Platform. Specifically, by addressing the research questions, our approach contributes to the IoT in particular to enabling IoT platform with autonomic capabilities. Main contributions include a software engineering for achieving autonomic computing in IoT platforms, an extension of existing IoT ontology toward supporting modeling application for autonomic behaviors and the realization of the SAI platform as well as application development. The following subsections will discuss these questions in more details.

Research Question 1. *How to design an autonomic IoT platform addressing the system evolution challenges?*
The initial step to answer this question is to conduct a study about challenges related to IoT system evolution as well as existing solutions to address them. Accordingly, with the evolution of IoT systems, these following factors could challenge IoT platform development: *heterogeneity* indicates the diversity of devices involved into the system, *scalability* represents enormous amount of devices continually generating data, and *interoperability* signifies abilities of seamless interaction among different entities in the system. As a result, there are various approaches to tackle these problems as presented in Chapter 2, which, in general, employed a service-oriented architecture that separates the physical integration from application development and provide access to physical layer with services in order to increase the flexibility from application perspective, thus hiding heterogeneity of devices from higher layer and improving platform support toward scalability. In addition, semantic representation of the system could be introduced in order to support interoperability. There are also works focusing on developing self-managed operations for those system, which are either ad-hoc or specific to a particular problem. Considering this limitation as a technological gap in the area, we aim to enhance IoT platforms with autonomic abilities with respect to three aforementioned challenges. As suggested by IBM, in order to achieve self-governance at high level, it is necessary to employ a reference model for autonomic control loops [30], which is sometimes called the MAPE-K loop. This model is being used more and more to communicate the architectural aspects of autonomic systems [27]. In this thesis, we also employ the model for achieving autonomic computing in our IoT platform. Particularly, the model is used as a fundamental architecture for the Autonomic Manager which is the key component for developing autonomic behaviors of the platform. The Autonomic Manager is introduced into the platform under the modular concept which enables the reusability of this component for existing IoT platforms. Furthermore, the platform also follows service-oriented paradigm to support heterogeneous physical devices. OWL ontologies are used as the core knowledge representation of internal architecture of the whole system, thus improving the semantic understanding of the system, which is necessary for enable interoperability among the system entities. Finally, autonomic behaviors are designed under the form of SWRL rules with respect to the system administrator's policies in order to provide the platform with self-managed capabilities with regard to scalability problem.

As a comparison with other related works, our proposed architecture is one of the initiatives which concretely apply Autonomic Computing concept into IoT systems. Compared with the DIAT architecture [39, 43] presented in Chapter 3, this work explains in more details the internal mechanisms of each component of the reference loop as well as knowledge representation and policy aspects. In addition, while DIAT architecture supported relatively limited self-governance capabilities by using simple context representation, our proposed platform provides autonomic behaviors with higher flexibility and more general purposes with the use of OWL ontologies and SWRL rules. Compared with the MoRE engine [27], although our platform employs the same MAPE-K model, the approaches differ in the representation of reconfiguration policies. Particularly, MoRE engine used Variabilities Models for generating new reconfiguration plans which are, in our approach, specified by SWRL rules. Using Variability Models at design time could limit the variety of autonomic behaviors since not every system has a design model completely reflecting

its actual implementation. Additionally, our platform also employs various techniques in Semantic Web Technology in order to support filtering monitoring data and detecting critical contexts, which is not addressed in MoRE Engine where these contexts are assumed to be provided by the administrator.

Research Question 2. *How to model the internal architecture of IoT applications in order to facilitate adaptation process?*
As explained earlier, the knowledge base is the core component which facilitates information retrieval and communication among the other components of the MAPE-K model. An efficient knowledge representation could not only increases the flexibility and intelligence level of autonomic behaviors but also improves the reusability of the autonomic solution. In this regard, we propose using OWL ontology for representing IoT aspects of our platform. Firstly, OWL language is a standardized Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things; which is highly suitable for the usage of our case. By using OWL ontologies, we can quickly reuse existing tools that support tasks related to knowledge processing of the wide Semantic Web Community. In fact, as presented in previous chapter, we also exploit other facilities (SPARQL query, SWRL rule) of the Semantic Web Technology along with the OWL ontologies to implement different mechanisms related to analysis and reasoning. In addition, main characteristics of OWL are their public openness and extensibility. Once designed and published, the ontologies can be imported and immediately reused or extended with necessary domain-specific concepts by any third parties. This reduces the time and effort to develop an ontology from scratch, and exempts ontology engineers from 'reinventing the wheel' [18]. Furthermore, by using OWL ontologies, we separate the knowledge base from the programing code of the platform. As a result, the knowledge base can be modified or updated seamlessly to the whole platform. In practice, the process of knowledge modification could be assigned to third-party add-on providers who are expected to to be more familiar with critical situations related to the system domain.

Regarding IoT area, there have been a lot of work focusing on defining ontologies for different domains in IoT. The ultimate goal is improving semantic understanding of IoT systems, thus helping to improve interoperability of those systems. Since these ontologies provide sufficient knowledge representation for IoT concepts, we propose to reuse these ontologies to model the internal architecture of our system in order to quickly obtain an efficient ontology for our autonomic platform. Specifically, we extend the IoT-O ontology [10] with IoT application-related concepts in order to support providing autonomic behaviors of IoT applications. As a result, we present in this work the SAI ontology which is used as the common vocabulary for defining knowledge, policies and rules which are used throughout the whole autonomic process. Accordingly, we also propose the approach to model the SAI application (as a set of rules) in accordance with the SAI ontology to support semantically analyzing those applications in order to facilitate the process of autonomic planning.

Research Question 3. *How to realize the proposed design for autonomic computing into executable implementation satisfying reasonable IoT performance requirement?*
Regarding this question, we propose to exploit different techniques of the Semantic Web Technology for implementing the mechanisms of the other components in MAPE-K loop. Specifically, as present in Chapter 4, the C-SPARQL Query Engine is used for analyzing

real-time streams of monitoring data and filtering only critical situations in respect of the SPARQL queries specified in the knowledge base. Accordingly, the SWRL Reasoning Engine takes those query results as the input and reasons over the knowledge base for any adaptations needed by the applications. Output of this reasoning engine will then be used to construct the new configuration which is sent to the corresponding application for reconfiguration. Since the performances of these engines were already evaluated under scalability perspective, our platform, once employing these technologies, can also handle large-scaled systems involving enormous number of entities. In fact, in oder to assess processing performance of the proposed platform, we also present a Smart Home Case Study and conduct detailed experiments with respect to IoT requirements. As discussed from above section, the results from these experiments also show the potential of our solution toward scalability challenge.

Compared to the EXCLAIM framework [18] presented in Chapter 3, which also exploited these techniques for the process of monitoring and analysis, our proposed approach take a further step toward planning and executing adaptation plan. Furthermore, while EXCLAIM framework address self-governance in cloud based platform, our platform provides autonomic behaviors for general-purpose IoT systems which could be deployed distributedly. In next chapter, we will explain more in details about this benefit of our proposed solution.

Chapter 7

Conclusion

In this thesis, we aimed to address issues related to IoT system evolution by enhancing IoT platforms with autonomic capabilities. Accordingly, we introduced three problem statements regarding this direction and step-by-step answered these questions by introducing the three main elements of our approach - the conceptual design of an autonomic platform called SAI, an extension of IoT ontology toward application modeling and the realization of the SAI platform, which are also the primary contributions of this work. This chapter will summarize these main aspects of the thesis in a more structured manner. Specifically, the chapter starts with a discussion of our contributions. Next, we list possible benefits of the proposed approach, followed by potential limitations of the platform. The chapter concludes with a discussion about the future improvements in connection with the presented limitations.

7.1 Contributions

We now summarize the main contributions associated with the presented SAI platform. These are our findings in accordance with the presented problem statements, and can be potentially re-used by the wider research community.

1. **A software engineering approach for achieving autonomic computing in IoT platforms**

   As explained at the very beginning, when IoT systems grow into large scale, challenges related to heterogeneity, scalability and interoperability could become critical to the platform development. Furthermore, existing IoT platforms provide various approaches to those challenges, in which autonomic solution is still immature and worth exploring. In this light, we proposed our conceptual design of the SAI platform which is:

   - A service-oriented platform, which means its interaction with external entities is based on services. Particularly, all devices in the system are abstracted to web services which could be hosted in different location. This paradigm standardizes the communication with smart devices from the perspective of applications and support seamless integration of new devices, thus addressing the heterogeneity problem.

- An autonomic platform that follows the established MAPE-K reference model for self-adaptations. Specifically, the fundamental underpinning of this approach is interpretation of IoT entities (e.g., applications, services) as 'software sensors' which continually emitting raw heterogeneous data to be monitored and analyzed to support run-time situation assessment. This enabled us to reuse existing solutions developed by Semantic Web Technology and Big Data processing which effectively support large-scale systems.
- An ontology based autonomic manager which employs OWL ontology from Semantic Web for representation internal architecture of the system. Accordingly, every governance-relevant aspect of IoT entities could be semantically represented by using a common ontological vocabulary, thus enable the interoperability among them.

Following these ideas, we presented *a software engineering approach for achieving autonomic computing in IoT platform* - namely, SAI platform. This platform hides the heterogeneity of hardware, software, data formats and communication protocols by standardizing communication into web services. In addition, the platform is also equipped with autonomic capabilities to support self-adaptability for large-scale system.

2. **The SAI ontology**

   The ontology acts as the core element of the underlying knowledge base, used throughout the whole process of the autonomic manager from monitoring changes to the environment to executing respective adaptation. Therefore, the SAI platform needs an ontology covering almost every IoT concepts necessary for the process. Designing such ontology from scratch requires enormous effort. Instead, we proposed a *SAI ontology* which extends existing ontologies in the fields of IoT and sensor network, and design our own application model to support different adaptation plans from application side. The SAI ontology, in fact, could be extended appropriately to model a particular IoT application which supports even more complicated adaptation. Additionally, the ontology also plays an important part in enable interoperability among internal components of platform as well as external entities (e.g., applications, devices).

3. **The SAI platform**

   We have presented *implementation details of the SAI platform* which employs different existing technologies in Semantic Web. In particular, inspired by the Semantic Sensor Network, we express the heterogeneous sensor values into RDF triples using the common ontological vocabulary, which later are input to the SPARQL Querying Engine acting as a critical situations detector and incoming data filter in order to avoid overwhelming amount of RDF instances into the next stage. For reasoning, the platform represents adaptation policies in form of SWRL rules in order to exploit existing SWRL Reasoning Engine for generating reconfigurations which eventually will be sent to the application as a guideline for self-adapting.

   The design and implementation of the platform was done with scalability in mind. Additionally, it is also validated under different perspectives regarding IoT requirement. Specifically, the majority of the specified functionality has been implemented

and testing from two sides: processing performance of the SAI platform and cost of reconfiguration of the SAI applications. The results of the experiments also demonstrate the efficiency of the platform toward supporting autonomic adaptation for applications. Although the platform still suffers from scalability issue, it is argued to be acceptable when compared to manual operation, and that appropriate optimizations can be employed to improve the performance when the system grows into large scale.

7.2 Potential benefits of the approach

Apart from enabling IoT systems with self-governance capabilities, the proposed approach also have following benefits:

Increased opportunity for reuse
The SAI platform was designed in a similar way to existing service-oriented architecture, which makes our proposal easily to be integrated to existing approach as an extension supporting autonomic capabilities. In fact, our architecture can act as a high-level conceptual model for creating autonomic IoT platform, which is independent of the underlying technology and implementation (e.g. how the communication between the devices and the monitoring component takes place). Furthermore, the platform is designed to be a modular architecture in which each element performs its functionality in an independent manner, thus increasing the possibility of plugging or replacing new module serving equivalent functions.

Possibility of distributed deployment
Since the platform consists of different modules working independently from each other, which have data transfered under standardized form (e.g., RDF stream, OWL instances), it could be implemented distributedly on different computers. As an example, parts of the architecture could be running in the cloud offering high-performance services (e.g., the SWRL Reasoning Engine), while parts interacting locally with the devices could run even on embedded systems located at the device layer. As a result, it not only improves performance of the platform on autonomic support but also increases the reliability of provided services to the application.

Loose coupling between knowledge and platform
Our proposed architecture applies the modularity principle of separating the knowledge base from the main programing code. Such separation remarkably simplifies modification of the knowledge base. As a result, the platform provides us with the possibility to declaratively define the knowledge base and update it if needed dynamically during run-time, without recompiling and restarting the whole system. Additionally, critical queries and adaptation rules can also be added into the platform 'on the fly' with no effect on the operational stability of the platform.

7.3 Potential limitations of the approach

In this section, we summarize associate limitations to the presented approach as well as the implementations.

Performance issues associated with scalability
In this work, we employed the Pellet Reasoner integrated with the OWL API for reasoning about the new configuration. This is an in-memory reasoner which shows its high performance toward small knowledge base. However, when the knowledge is growing in size and complexity, it may result in a considerably heavy-weight ontology (e.g., more than 10^5 devices existing in the system), and therefore can slow down the reasoning process. One possible solution to this issue is that breaking down the knowledge base in to several parts, and enable the engine to pick only the necessary elements for reasoning in oder to avoid keeping the whole heavy-weight ontology in memory [14]. Another approach provided in [18] is that transferring all reasoning tasks associated with the problem detection into the streaming step, thus exploiting existing big data streaming tool for processing input stream and not requiring any static reasoning component at all. Unfortunately, at the moment, existing RDF streaming support is not sufficient for representing diagnosis and adaptation policies with continuous SPARQL queries, thus static SWRL reasoning is still in need.

Immature application modeling
This issue was already mentioned several times in previous sections, which is the current proposed ontology is not enough for adequately representing all aspects of IoT applications. Therefore, the possibilities of more complex adaptation plans could be limited from the platform. However, since the platform is designed to be separated from the knowledge base, this limitation could be addressed by introducing more concepts of IoT application into the ontology without affecting operation of the whole system.

7.4 Future Work

In this section, we outline several directions for future activities which we found potential to improve efficiency of the platform significantly.

Employing machine learning technique for policy inference
In this work, we assume that the knowledge base is manually populated by the platform administrator. That is, it is required human effort for defining new rules or queries in accordance with their polices. The same manner applied to the reverse process - once old services are retired and do not need to be monitored or analyzed, the corresponding policies should be manually removed from the knowledge base. These processes can be done automatically with the help of machine learning techniques. Accordingly, since the data generated by large scale system could be considered sufficient for referring meaningful information, it is worth exploring possible machine learning techniques into these data in oder to enable the knowledge base with self-training capabilities so that new policies would be added and existing ones would be modified with respect to changing observations.

Enriching the ontology with more concepts toward adaptation
The proposed ontology could be improved with multiple concepts beneficial to developing autonomic behaviors. First of all, as explained earlier, current semantic representation of the application do not cover all aspects regarding IoT context. At the moment, the ontology is designed to present only information about services and sensors being used by the application, thus putting limitation on the variety of adaptation. In fact, improving

the semantic understanding of application logics would be beneficial to developing autonomic behaviors of the whole system. Therefore, it is worth putting effort on enhancing the ontology toward this direction. Another concepts missing from the ontology is complex service which is a combination of multiple services in order to support more complicated functionalities. In fact, such service composition could be considered as a special type of application, which if existed, would simplifies application development as well as improve the 'quality' of adaptation plan. As an example, a smoke sensor could be considered to be substituted by a combination of a temperature sensor and an air quality sensor, thus improving the precision of service approximation process.

Bibliography

[1] Everyware. https://www.eurotech.com.

[2] EvryThng. https://evrythng.com.

[3] IFTTT. https://www.ifttt.com.

[4] SensorClould. https://www.sensorcloud.com.

[5] ThingSquare. https://www.thingsquare.com.

[6] ThingWorx. https://www.thingworx.com.

[7] Xively. https://www.xively.com.

[8] Adnan Aijaz and A Hamid Aghvami. Cognitive machine-to-machine communications for internet-of-things: a protocol stack perspective. *IEEE Internet of Things Journal*, 2(2):103–112, 2015.

[9] Atif Alamri, Wasai Shadab Ansari, Mohammad Mehedi Hassan, M Shamim Hossain, Abdulhameed Alelaiwi, and M Anwar Hossain. A survey on sensor-cloud: architecture, applications, and approaches. *International Journal of Distributed Sensor Networks*, 2013, 2013.

[10] Mahdi Ben Alaya, Samir Medjiah, Thierry Monteil, and Khalil Drira. Toward semantic interoperability in onem2m architecture. *IEEE Communications Magazine*, 53(12):35–41, 2015.

[11] Qazi Mamoon Ashraf and Mohamed Hadi Habaebi. Autonomic schemes for threat mitigation in internet of things. *Journal of Network and Computer Applications*, 49:112–127, 2015.

[12] Qazi Mamoon Ashraf, Mohamed Hadi Habaebi, Gopinath Rao Sinniah, Musse Mohamud Ahmed, Sheroz Khan, and Shihab Hameed. Autonomic protocol and architecture for devices in internet of things. In *2014 IEEE Innovative Smart Grid Technologies-Asia (ISGT ASIA)*, pages 737–742. IEEE, 2014.

[13] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[14] Carliss Young Baldwin and Kim B Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[15] Victoria Bellotti and Keith Edwards. Intelligibility and accountability: human considerations in context-aware systems. *Human–Computer Interaction*, 16(2-4):193–212, 2001.

[16] Gordon S Blair, Geoff Coulson, and Paul Grace. Research directions in reflective middleware: the lancaster experience. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 262–267. ACM, 2004.

[17] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[18] Rustem Dautov. *EXCLAIM framework: a monitoring and analysis framework to support self-governance in Cloud Application Platforms*. PhD thesis, University of Sheffield, 2015.

[19] Rustem Dautov, Dimitrios Kourtesis, Iraklis Paraskakis, and Mike Stannett. Addressing self-management in cloud platforms: a semantic sensor web approach. In *Proceedings of the 2013 international workshop on Hot topics in cloud services*, pages 11–18. ACM, 2013.

[20] Rustem Dautov, Iraklis Paraskakis, and Dimitrios Kourtesis. An ontology-driven approach to self-management in cloud application platforms. In *Proceedings of the 7th South East European Doctoral Student Conference (DSC 2012)*, pages 539–550, 2012.

[21] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1(2):223–259, 2006.

[22] Michael Dunbar. Plug-and-play sensors in wireless networks. *IEEE Instrumentation & Measurement Magazine*, 4(1):19–23, 2001.

[23] Carson P Edwards, David G Leeper, Robert I Foster, Ray O Waddoups, and Sam Mordachai Daniel. Self-organizing network with decision engine and method, March 22 2005. US Patent 6,870,816.

[24] Jens Ehlers, André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 197–200. ACM, 2011.

[25] Mahmoud Elkhodr, Seyed Shahrestani, and Hon Cheung. A smart home application based on the internet of things management platform. In *2015 IEEE International Conference on Data Science and Data Intensive Systems*, pages 491–496. IEEE, 2015.

[26] Wilfried Elmenreich, Raissa D'Souza, Christian Bettstetter, and Hermann de Meer. A survey of models and design methods for self-organizing networked systems. In *International Workshop on Self-Organizing Systems*, pages 37–49. Springer, 2009.

[27] Carlos Cetina Englada. *Achieving autonomic computing through the use of variability models at run-time*. PhD thesis, Universidad Politecnica de Valencia, 2010.

[28] Michael Fahrmair, Bernd Spanfelner, Wassiou Sitou, Klaus-Dieter Althoff, and M Schaaf. Unwanted behavior and its impact on adaptive systems in ubiquitous computing. In *LWA*, pages 36–41, 2006.

[29] Alan G Ganek and Thomas A Corbi. The dawning of the autonomic computing era. *IBM systems Journal*, 42(1):5–18, 2003.

[30] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. 2001.

[31] Markus C Huebscher and Julie A McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7, 2008.

[32] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[33] Jeffrey O Kephart and William E Walsh. An artificial intelligence perspective on autonomic computing policies. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3–12. IEEE, 2004.

[34] Jeffrey King, Raja Bose, Hen-I Yang, Steven Pickles, and Abdelsalam Helal. Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces. In *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pages 630–638. IEEE, 2006.

[35] Philippe Lalanda, Julie A McCann, and Ada Diaconescu. Future of autonomic computing and conclusions. In *Autonomic Computing*, pages 263–278. Springer, 2013.

[36] Shancang Li, Li Da Xu, and Shanshan Zhao. The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259, 2015.

[37] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[38] Rohan Narayana Murty, Geoffrey Mainland, Ian Rose, Atanu Roy Chowdhury, Abhimanyu Gosain, Josh Bers, and Matt Welsh. Citysense: An urban-scale wireless sensor network and testbed. In *Technologies for Homeland Security, 2008 IEEE Conference on*, pages 583–588. IEEE, 2008.

[39] SN Akshay Uttama Nambi, Chayan Sarkar, R Venkatesha Prasad, and Abdur Rahim. A unified semantic knowledge base for iot. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 575–580. IEEE, 2014.

[40] Mohammad Reza Nami and Koen Bertels. A survey of autonomic computing systems. In *Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on*, pages 26–26. IEEE, 2007.

[41] Apostolos Papageorgiou, Manuel Zahn, and Ernö Kovacs. Auto-configuration system and algorithms for big data-enabled internet-of-things platforms. In *2014 IEEE International Congress on Big Data*, pages 490–497. IEEE, 2014.

[42] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[43] Chayan Sarkar, SN Akshay Uttama Nambi, R Venkatesha Prasad, and Abdur Rahim. A scalable distributed architecture towards unifying iot applications. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 508–513. IEEE, 2014.

[44] Amit Sheth, Cory Henson, and Satya S Sahoo. Semantic sensor web. *IEEE Internet computing*, 12(4), 2008.

[45] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, Luciana Moreira Sá De Souza, and Vlad Trifa. Soa-based integration of the internet of things in enterprise services. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 968–975. IEEE, 2009.

[46] John A Stankovic. Research directions for the internet of things. *IEEE Internet of Things Journal*, 1(1):3–9, 2014.

[47] Richard Tynan, Gregory MP O'Hare, and Antonio Ruzzelli. Autonomic wireless sensor network topology control. In *2007 IEEE International Conference on Networking, Sensing and Control*, pages 7–13. IEEE, 2007.

[48] Pal Varga, Fredrik Blomstedt, Luis Lino Ferreira, Jens Eliasson, Mats Johansson, Jerker Delsing, and Iker Martínez de Soria. Making system of systems interoperable–the core components of the arrowhead framework. *Journal of Network and Computer Applications*, 2016.

[49] Yi Xu and Abdelsalam Helal. Scalable cloud–sensor architecture for the internet of things. *IEEE Internet of Things Journal*, 3(3):285–298, 2016.