Maintainable Software Solution Development Using Collaboration Between Architecture and Requirements in Heterogeneous IoT Paradigm (Short Paper)

Wajid Rafique[1,2], Maqbool Khan[1,2], and Wanchun Dou[1,2](✉)

[1] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, People's Republic of China
rafiqwajid@smail.nju.edu.cn, maqbool@163.com, douwc@nju.edu.cn

[2] The Department of Computer Science and Technology, Nanjing University, Nanjing, People's Republic of China

Abstract. Internet of Things (IoT) has been tremendously involved in the development of smart infrastructure. Software solutions in IoT have to consider lack of abstractions, heterogeneity, multiple stakeholders, scalability, and interoperability among the devices. The developers need to implement application logic on multiple hardware platforms to satisfy the fundamental business goals. Moreover, long-term maintenance issues due to the frequent introduction of new requirements and hardware platforms pose a vital challenge in IoT solution development.

Numerous techniques have been devised to satisfy the issues mentioned above for ubiquitous and smart infrastructure. However, these techniques lack in providing a comprehensive approach in dealing with the above challenges. In this paper, we argue that fundamentally, there is no difference between the architecturally significant requirements and the architectural design decisions in IoT solution development. The architecture revolves around the requirements gathered by the analyst at the requirements gathering phase. We stress that the requirements elicitation process must consider the software architectural assessment for maintainable software development. By adopting this perspective, we identify areas where both requirements and architecture communities collaborate to effectively increase the user acceptability, maintainability, and fulfill the heterogeneous needs of IoT solutions.

Keywords: Software architecture · IoT · Requirement Engineering · IoT solution development · Maintainability

1 Introduction

Rapid progress in IT has made the ubiquitousness a reality where individuals are widely using smart sensors, gesture control gadgets, and wearable devices in everyday life [1]. Nowadays, smart devices/things have the capability to connect with the internet and control the real-world infrastructure. These smart devices/things are denoted by the Internet of Things (IoT) and are widely being deployed in everyday life during the past few years. IoT provides advanced services by employing a global internet infrastructure including homes, manufacturing, aviation, agriculture, health-care, and many other areas of life. The things in IoT have a unique identity, connected by wireless network connections, and can be remotely controlled [1]. IoT has revolutionized the human lifestyle by enabling decision-making capabilities in the devices with very less human interaction.

Figure 1 illustrates an IoT scenario which uses traditional cloud and mobile edge computing for successful IoT implementation. The outermost layer corresponds to the actuators and sensors embedded in the IoT devices which use internet and gateways for the communication to offload the compute-tensive tasks to the edge infrastructure. The traditional cloud data center infrastructure is at the core of IoT which acts as the centralized data repository. A seamless IoT operation is governed by the software solutions in the IoT devices. A typical architecture of IoT consists of a mini processor, actuators, and sensors. IoT has been deployed in networked systems where smart devices collect data using sensors and pass it to the physical systems to control the real-world infrastructure. IoT uses IPv6 network addressing scheme which can accommodate a huge number of IoT devices with larger address space as compared to approximately 4.3 billion device space of IPv4 [2].

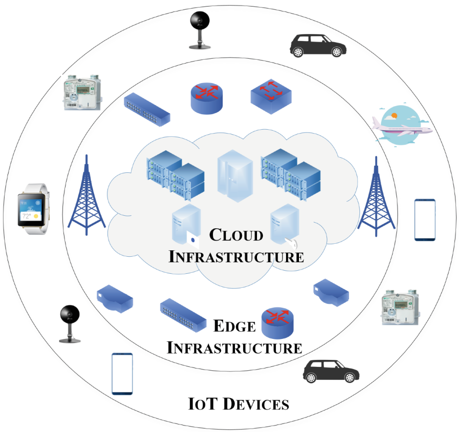

Fig. 1. An IoT service orchestration example using traditional cloud and mobile edge computing.

Software solutions in IoT are the fundamental component in handling the IoT services. They comprise up of architectural components used in a specific configuration while the interaction among these components depends on the underlying software architecture. These components denote the state of the system while the links between them describe the interaction pattern among them [3]. Moreover, the architectural configurations provide the underlying structure of the software design components. Furthermore, the architectural design explains the configuration of the software and hardware components and their interaction to orchestrate services from IoT. Software requirements provide the basis for the architectural development of an IoT software solution. These can be represented by the Architectural Design Decisions (ADDs) to illustrate the structure of a software solution.

Due to the gigantic proliferation of IoT, the need for standardized software architecture development has become a vital challenge [4]. Recently, many new software development life cycles aligned with the requirements of rapid application development and evolution have been introduced. Heterogeneity in IoT makes maintainable software development as a complex task. Traditional software architectural patterns suffer in IoT solution context which relies heavily on the functional requirements. However, ADDs in IoT depend mostly on the nonfunctional requirements of scalability, interoperability, programmability, and maintainability.

Therefore, the study of correlation among software architecture and the requirements in IoT is of vital importance. Both the solution architect and the requirements' analyst work on their specific part of the problem separately, where the architect concentrates on the development perspective, whereas the requirements' analyst apprehends the customers' view. The requirements translate the system in terms of "What" and "How," statements where "What" represents the functionality of the product to be developed and "How" corresponds to the design of the system. Hence, there is still a need to study areas where requirements and architecture are strongly associated [5].

Requirement Engineering (RE) in IoT concentrates on the elicitation of the intended user-goals from IoT that are specified in the form of functional requirements, quality attributes, and constraints. The underlying functions of requirements are distributed among agents like humans, available solutions, or the solutions that need to be developed [5]. The requirements comprise up of diverse characteristics because of the heterogeneity in the IoT infrastructure. They should satisfy the following goals.

- The problem domain must be represented in the form of precise formal statements.
- The problem statements should completely fulfill the intended functional requirement of the business.
- The relationship among different problem statements must be analyzed and documented.
- The stakeholders involved in the problem and the solution domain must be explicitly identified.

Figure 2 illustrates the characteristics of IoT including programmability, intelligence, unique identification, and internet access. The interoperability in IoT denotes its property to operate with multiple other platforms including

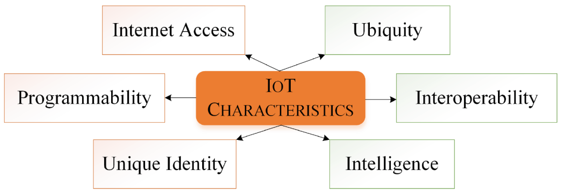

Fig. 2. Characteristics of IoT.

cloud infrastructure, physical devices, and other computing systems. The traditional architectural pattern selection tools analyze, make design decisions, and document them according to software requirements. However, these tools may not be equally effective in the IoT software development domain [5]. The analysis of the user requirements is performed from the top to bottom and further refined until the properties of the desired solution are established [6]. Recent research in the field of software engineering has changed the perspective of considering software architecture as only an abstract entity to a wider scope of architectural knowledge [5]. Additionally, new trends stress the importance of ADDS for developing maintainable software. In this research, we discuss different approaches which establish the collaboration among requirements and the architecture in the IoT paradigm. We present the Architecturally Significant Requirements (ASRs) that can be transformed to the ADDs and explain the architectural choices that are appropriate for effective software development in the IoT paradigm. Furthermore, we provide different models on the collaboration of requirements and the architecture. Based on the above discussion, we present the following contributions in this paper.

- We compare different perspectives of the software community on how software requirements and architecture contribute to the software development process in the heterogeneous IoT application domain.
- We propose that the requirements and architecture are the main building blocks for developing maintainable software and discuss how key characteristics are lost if software requirements and architecture are not considered concurrently.
- We present a systematic discussion on how different models can be used in the software requirements and architecture evolution and ascertain how a maintainable IoT solution can be developed using these models.

Table 1 illustrates the key terms and their description used in this research. Rest of the paper is organized as follows, Sect. 2 discusses the related work on different techniques used in the software architecture development for IoT. Section 3 provides the novel characteristics of IoT which pose challenges in maintainable architecture development. Section 4 explains insights into the approaches used in simultaneous software requirements and architecture development. Section 5 discusses different models to translate problems into corresponding solutions in IoT, whereas Sect. 6 includes the effect of requirements on ADDs. Furthermore,

Table 1. Key terms and their explanation.

Term	Explanation
ADD	Architectural Design Decision
CBSP	Component-Bus-System-Property
ASR	Architectural Significant Requirements
RE	Requirement Engineering
COTS	Commercial-of-the-shelf
IoT	Internet of Things
XP	Extreme Programming
JSF	JavaServer Faces
JMS	Java Message Service
MDD	Model Driven Development
KIWIS	I will Know it When I see it
ADRQ	Architectural Design Design and Requirement Repository

Sect. 7 provides a discussion on the cross-benefit analysis of software requirements and the architecture finally, Sect. 8 concludes the paper and provides some future insights.

2 Related Work

Software architecture is the collection of design decisions which presents a high-level structure of a software system including its components and their interaction [7]. The software architecture paradigm has been considered as structure-oriented in the past, however, with the development of smart infrastructure, it has been transformed towards the knowledge-oriented domain. The architecture is not a solution structure whereas, it is a collection of design decisions which tranced towards a structure [7]. Khan et al. discuss the similarities between the nonfunctional requirements and the architecture of the software. However, their approach lacks in presenting a solution for the heterogeneous IoT domain [8].

Michael Jackson devises optative and indicative problem frames and assigns frequently occurring requirements to a particular frame to document the available software specifications [9]. The indicative corresponds to the problem domain, and optative denotes the selected choices according to the underlying machine specifications. The problem domain is definite, which is represented by the indicative properties, e.g., the requirement of storing the temperature sensor's data by the IoT device. Whereas, the optative corresponds to the requirements, which are the explanations of what the client needs to be true in the problem side, e.g., the interval between consecutive data transfers of an IoT sensor. The machine specifications and the actual behavior of the machine at its interface are different. Limitations on the behavior of the hardware and specifications describe the difference between architecture, requirements, problem domain, and the solution.

Janson et al. propose an Archium model which maps the software architecture to a set of ADDs [10]. The COMPACT model proposed by Marquez et al. extracts components from the nonfunctional requirements [11]. However, COMPACT does not utilize any formal criteria to select design decisions. An improvement of COMPACT has been presented in [12], which utilizes a semantic recommendation system and provides architectural components from the business scenarios. It assists in architectural decision making by bridging the gap between software architecture and the requirements.

A web-based tool has been developed by [13], which assists in selecting architectural pattern and style from an architectural repository, however, it fails in addressing quality attributes in selecting architectural styles and patterns. Cai et al. [14] propose Model Driven Development (MDD) for mobile services in the cloud, they demonstrate that the current software service architectures start in a non-sequential manner and their limitations are identified during finalization of the architecture. This methodology often needs reconsideration, which increases cost and time. MDD for IoT has been proposed in [15] named as FARASAD framework. Similarly, an MDD methodology and SOA framework for architecture development and software artifacts generation have also been proposed in [3]. However, MDD tends to develop the solution at a higher level of abstraction, which makes it difficult to handle the low-level requirements of IoT solutions. Moreover, these technologies do not stress the need of ASRs for IoT solution development. Most of the times, the requirement analysis and the modeling teams include different individuals. Thus, it is difficult to translate the domain knowledge into the models.

These studies provide awareness of the relationship between software architecture and the requirements where the requirements represent the problem domain whereas the architecture illustrates the solution paradigm. As it has been discussed in the related work that different authors use ADD tools to select the appropriate architectural styles and patterns. However, a lack of holistic approach has been observed in the field of IoT architectural pattern selection from the requirements which poses novel challenges. The architectural selection problem in IoT depends on communication protocols, resource limitations, interoperability, scalability, and cloud support. Therefore, we discuss the architectural style and pattern selection from different perspectives.

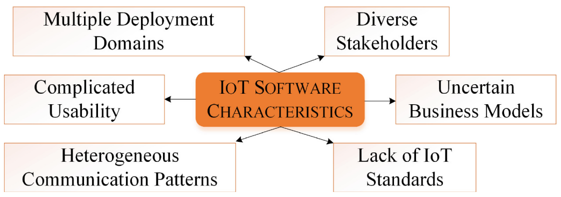
Multiple Deployment Domains
Diverse Stakeholders
Complicated Usability
IoT Software Characteristics
Uncertain Business Models
Heterogeneous Communication Patterns
Lack of IoT Standards

Fig. 3. IoT solution characteristics.

3 Characteristics of IoT Software Solution

Most of the IoT software solutions have been developed without the complete realization of the business and market because of a rapid IoT evolution during the past few years [1]. A common approach used by most of the IoT businesses is to start with a small-scale solution with the flexibility to innovate in order to cater to the huge business investment constraints. Large-scale development having entire features starts upon the acceptance of the product in the market. Figure 3 illustrates the characteristics of IoT solutions which have been discussed in the following.

3.1 Multiple Deployment Domains

IoT solutions need to be deployed on different distributed layers due to multiple IoT hardware constraints, including lower bandwidth, storage, and computation power. The resource limitation also instigates that the data security and privacy requirements are handled in a distributed way, hence, they are mostly provided on the network infrastructure. Moreover, different programming languages are used to develop IoT solutions without having a common code base. Nevertheless, with different build tools, frameworks, versatile platforms, and release cycles, everything must have to run smoothly. Similarly, IoT solutions need data management and storage capabilities to deal with the big data generated by huge IoT infrastructure. Sometimes, the data is classified and filtered before transferring to the cloud storage. In this context, NoSQL comes into play to efficiently manage time-series and high volumes of data.

3.2 Complicated Usability

Smart homes, health-care, and wearable IoT devices are extensively being employed to provide state-of-the-art services to the users. Most of the IoT applications are developed considering the business needs whereas, neglecting the users' perspective of the IoT. The categorization of the requirements according to the involved users and the IoT devices must be extensively performed to understand user needs from the IoT applications. In this regard, the IoT software solutions should be designed in a way that the end users encounter fewer usability problems.

3.3 Lack of IoT Standards

Conway's law in the IoT domain illustrates that the independent and parallel activities in different IoT business solutions provoke many functionally similar solutions which do not follow a standardized development environment [16]. Things work well in a limited paradigm, however, face problems while operating under a broader business context. There are a few IoT standards in the market because of the continuous evolution of the IoT infrastructure. Moreover, many competing IoT development activities are going on around the world, which makes it difficult to enforce standardized IoT solution development.

3.4 Diverse Stakeholders

A diverse range of stakeholders, including versatile domain experts, legal advisers, standardization bodies, third-parties, and infrastructure owners make the requirement gathering a cumbersome process in IoT. Consequently, numerous feasible solutions can be eliminated due to minimal reasons. Extensive technology knowledge even for a smaller solution is inevitable, e.g., development of smart temperature sensors in a heat treating furnace requires deep metallurgical knowledge of the heat treatment process to control the temperature in different parts of the furnace.

3.5 Uncertain Business Models

The business models in IoT change over time because the number of users, the connected devices, and the functionalities varies after deployment. A distributed runtime environment having a versatile set of communication patterns also pose uncertainty in the deployment models. The presence of multiple other IoT solutions increases complexity in designing a solution. Moreover, interaction among architects from other solution domains is also imminent, which poses a significant overhead in IoT solution development.

3.6 Heterogeneous Communication Patterns

Protocols are necessary for moderation when devices communicate with each other. IoT needs novel communication protocols because it has been predicted that the level of heterogeneity of IoT will be much higher than the internet [2]. There is a greater need for communication protocols that enable inter-device communication. Moreover, successful software solutions need to consider heterogeneous communication patterns in IoT.

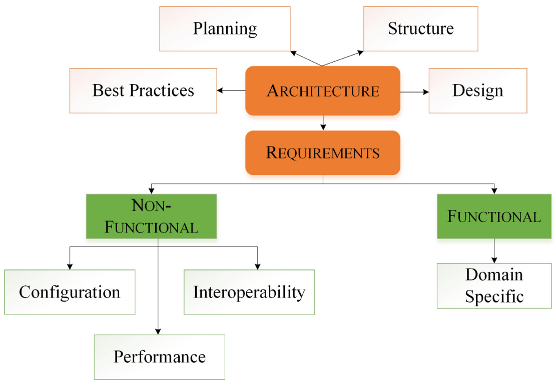

Fig. 4. The characteristics of IoT architecture and its dependence on requirements.

Most of the IoT solutions consider only a limited deployment paradigm which suffers in providing the solution for a global perspective. The above characteristics illustrate that the IoT software development undergoes numerous challenges due to the heterogeneity and lack of standardized development environments which must be addressed in developing a maintainable solution.

4 Software Requirements and Architecture in IoT

In the IoT domain, the requirements of IoT-big data, computational offloading, and resource limitation constraints need to be particularly addressed during the software development process [8]. Figure 4 illustrates the requirements and architecture modeling in an IoT solution development paradigm. It demonstrates that the functional requirements comprise up of domain-specific illustrations, whereas nonfunctional requirements depict configuration, interoperability, and performance. Furthermore, the architecture includes structure, design, and associated best practices in the software. The solution architecture must be aligned with the functional and nonfunctional requirements to enable the usability and maintainability of the IoT solutions. Many attempts have been put forward to evolve software architecture together with the requirements, including problem domains model [17], twin-peak model, and Component-Bus-System-Property (CBSP) approach [18].

The problem domain model relies on extracting individual domain models from the requirements. It represents the problem in terms of frames which increase the efficiency of the overall application development process and instigates the reuse of different frames, which reduces future development effort [17]. The domain modeling approach utilizes the object-oriented perspective for modeling the components of the problem and their relationship. The problem is specified in terms of multilevel abstract-layered representation, which illustrates the problem from multiple perspectives. These models are then developed as running software by the application developers.

Gunbacher et al. propose CBSP an architectural development approach which relates the requirements to the software architecture [18]. A hierarchical taxonomy is utilized to associate requirements with the architecture. A set of general requirements are provided to the CBSP method, which provides intermediate decisions about the architectural styles to be used.

Table 2. Relationship between RE and architecture design.

Requirement engineering	Architecture design
Goal-oriented RE	Pattern-based design
Use case-oriented RE	Architectural style-based design
Sociology and linguistics bases RE	Attribute-based design
Aspects-oriented RE	Component-based design
Sequence constrained business requirements	Product line-based design

IoT poses a versatile set of challenges which needs to be addressed in designing a solution, including the integration of software with heterogeneous platforms, big data needs, scalability, and versatile nature of communication device constraints. Moreover, the architecture is dependent on the software requirements, thus every statement in the requirement document must be represented in the architectural design of the IoT solution.

4.1 RE and Architectural Design

Requirements and the architecture are the integral components of the IoT solutions where requirements are considered as the analysis of the problem domain, whereas the architecture relates to the solution paradigm. IoT requirements consist of limited resources, interoperability among devices, scalability, and heterogeneity, which must be particularly addressed during the architectural development process. In designing an IoT solution, attention must be given to the ASRs which represent the essential design decision of the underlying solution. RE and software architecture play a pivotal role in developing the software structure as illustrated in Table 2. It represents the type of RE process used during the elicitation and the associated architectural design that can be employed.

Practically, both requirements and the architecture emerge separately however, there is a need to explore areas of collaboration among these two processes. Different architectural designs consist of many challenges which relate to the specific solution classes. It is always better to have an early understanding of the user requirements in the IoT solution development, consequently, it will be easier to achieve customer satisfaction towards the solution. Similarly, prior understanding of the architecture provides the basis to discover further constraints related to requirements and the architecture, it also helps to evaluate the system's deployment feasibility [6]. The waterfall process model creates the system architecture that confines the software team to do unavoidable changes in the requirements, which creates a bottleneck in updating the software architecture with evolving requirements. The spiral model was introduced to deal with these challenges, it resolved many deficiencies of the waterfall model and offered incremental software development which helps the developers to flexibly evaluate and change the requirements according to the project risks. The spiral process model concentrates on the need for the development of stable and maintainable software architecture. This model facilitates the developers in a way that they can work on requirements and architecture concurrently [7].

Twin-Peak Model. Figure 5 illustrates the twin-peak model, which enables requirements and architectural specifications in increments to fulfill the needs of the evolutionary software development in the IoT domain. The requirements can be associated with the architecture using a continuous evolution. They are translated to the software components in the architecture, whereas the interaction among the components is moderated by the requirement constraints. Requirements specification, component development, and configuration are carried out continuously until a final architecture is developed. A final assessment of the architecture is performed to ascertain that the architecture fulfills all the required specifications of the business logic. This model is an extension of the Stephen Mellor and Paul ward development model, which they proposed for real-time platforms [19]. Therefore, the twin-peak model has the ability to capture the needs of smart devices in the IoT paradigm. Change control in software development is one of the fundamental property which can efficiently be addressed by using the twin-peak model. Analysis and identification of core software requirements are extremely necessary for stable software architecture in a changing requirements scenario. Different processes are used to develop software systems in this context Commercial-of-the-Shelf (COTS) components can be utilized to re-using built-in products at an earlier stage of the requirements.

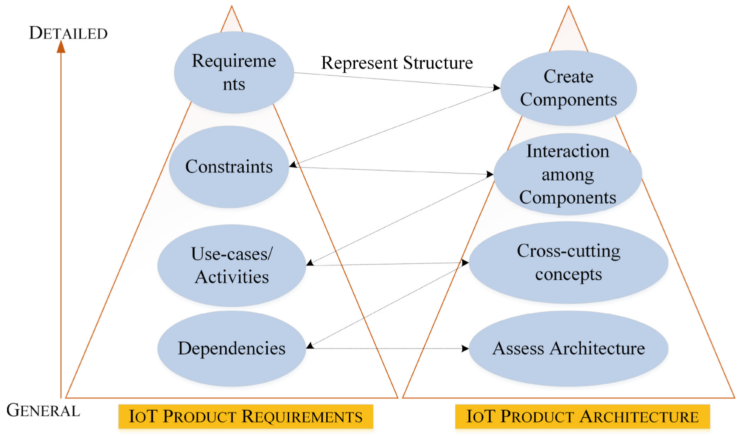

Fig. 5. Twin-peak model for translating IoT requirements to the architecture.

Twin-peak model has been widely used by software development organizations to deal with requirements specification and design issues concurrently. Independent consideration of software requirements and the architecture introduces many challenges which pose restrictions on the developers and provoke repeated software modifications. Agile software development model has the ability to deal with the changing software requirements. The underlying mechanism in agile is also based on the twin-peak model, which emphasizes strong interaction among architecture and the requirements domain. According to Barry Boehm, the twin-peak model has the following management concerns [20].

- **I will know it when I see it (KIWIS):** The requirements tend to change in the process of software development based on the user feedback on the releases. The twin-peak model can detect the changes at an early stage by using incremental modeling.
- **COTS:** Twin-peak model stresses the reuse policy by modifying the available COTS packages which reduce extra effort and cost.

– **Rapid change:** Twin-peak model employs highly adaptive and iterative modeling technique, moreover, it facilitates in incorporating the changes provided by the user during the development life-cycle.

The above discussion demonstrates that the twin-peak model facilitates efficient IoT solution development. It enforces reuse by employing multiple available design patterns and architectural styles which can be customized to adjust in the required context.

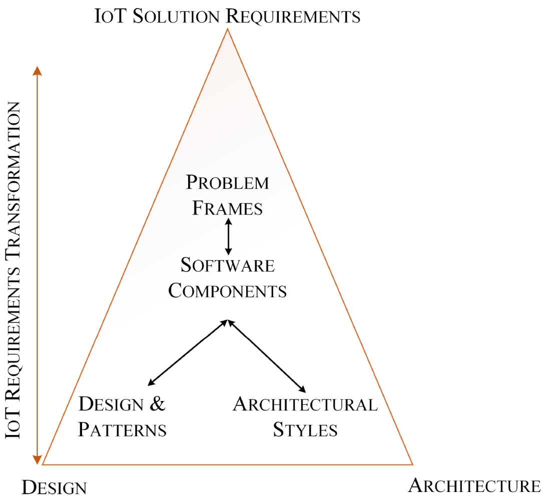

Fig. 6. Selection of solution design and architecture from IoT requirements.

Figure 6 illustrates how requirements can be translated to the design and architecture in IoT solution development. Initially, the requirements are represented by the problem frames, which further translates to the software components. Architectural styles and patterns are adopted by the developers according to the underlying software components. A predetermined architecture poses limitations on the underlying problem, alternatively, rigid requirements pose limitations on the architecture and design choices.

4.2 Weaving the Development Process

Extreme Programming (XP) approach has been used to stress the need for exploring possible implementations of a given problem iteratively [21]. This approach focuses more on front-end software development activities, architectures, and requirements. Therefore, large-scale projects can be efficiently managed if the requirements are comprehended at an early stage, and the choice of architecture is aligned with the requirements. The XP focuses on the production of code, whether it is at the expense of requirements or the architecture. Alternatively, a separate focus on requirements or architecture imposes scalability issues which can harm iterative development and modularity in IoT. Integrating the twin-peak model with the tested and derived components from reliable prototypes can also help in the development of large-scale applications in increments.

Comprehensive problem analysis helps in the reduction of time to market, high-quality, and cost-effective solutions. An efficient development life-cycle allows the concurrent evolution of the requirements and the architecture in order to produce the desired product. A concise consideration of the software architecture provides a clear understanding of the problem from the developers perspective. Moreover, the resultant architecture provides an accurate representation of the user requirements. IoT solutions need to consider the following questions.

- What are the stable requirements and how they can be selected in the context of rapidly changing requirements?
- What type of changes can be expected in the software architecture?
- How can the architecture and requirements be managed to minimize the evolving change impact because of the heterogeneous nature of the IoT devices?

The twin-peak model follows the characteristics of evolutionary software development. Thus, an answer to the above questions will help in identifying ASRs and further maintainable architecture development. There is still a need for a rapid software development process which ensures fast and incremental delivery in the IoT paradigm.

5 Translating Problems into Solutions

A challenging task in software engineering is how to devise a solution that satisfies the present customer's demands and addresses the needs for further evolution using a maintainable architecture. RE and architecture development are the most important activities in the software development life-cycle. The core objectives of an IoT solution can be ascertained during the requirements gathering process which ensures the unambiguity, correctness, and consistency of the requirements so that they provide a baseline for further development, validation, and evolution. The software architecture is explicitly defined, and a baseline is prepared on which subsequent development activities can be planned.

5.1 Problem Exploration Using RE

Problem exploration is concerned with the elicitation of the goals a user needs to accomplish from an underlying IoT device. When developing an IoT solution, an interplay exists between the problem domain and the solution because of the trade-off between the implementation of certain requirements over the others. However, the architecture depends on both the implementable and non-implementable requirements. The RE process explores the problem domain iteratively as further subproblems are identified during the implementation.

IoT solutions depend on design decisions during the architectural development process. These design decisions invoke multiple other design decisions based on the product requirements, which can be characterized in the following.

1. **Existence decision:** These decisions are represented in the implementation of the software system. Moreover, they show up as a quality of the system, e.g., "The temperature sensor solution will consist of three layers."
2. **Property decision:** These decisions have a high impact on the architecture, and they act as the design guidelines for the underlying system. These design decisions can be replaced by new design decisions, e.g., "IoT data will be offloaded to the cloud after 2 ms intervals."
3. **Executive decision:** These design decisions are not directly represented in the design of the solution. However, they are related to environmental factors including political, personal, financial, cultural, and technological constraints, e.g., "The IoT solution will be deployed by using SOA architecture."

ADDs have a high impact on developing a maintainable software solution for IoT. The relation between design decisions plays a pivotal role in the evolution and maintenance of the underlying IoT solution [22]. For instance, if we plan to develop a Java application and decide to use JavaServer Faces for implementation, it limits the use of Java Management Servlets and constrains the use of JavaServer Pages. Similarly, if we use publish-subscribe architectural style, it limits the decision to use peer-to-peer style and constraints with the decision of choosing the client-server architecture.

5.2 Discussion on Problems and Solutions in IoT

The software architect segregates the elicited requirements and identifies those which are not playing any role in the architecture. For example, the requirement to use a matrix-based display to illustrate the speed of a vehicle does not play any role in the software architecture development hence, we eliminate this requirement in the architecture level discussion. Therefore, architecture development only involves ASRs. Conflicting requirements may often appear in the IoT solution development, which needs special attention as IoT involves a versatile set of devices having different configurations. Heterogeneous nature of IoT, big data needs, and mobile platforms often provoke conflicting requirements. Seemingly, performance constraints in a particular situation employ an immense impact on the architecture of a software system. Architectural style selection directly affects the problem as the use of a style provoke new requirements which need additional design decisions.

In this discussion, we conclude that ASRs and ADDs have a strong relationship thus, a maintainable solution must consider both of them concurrently.

6 ADDs and Requirements in IoT

The problem space constitutes the specifications of the solution to develop, its structure, and the domain. ADDs use problem rather than the solution domain which allows the owner who has the core knowledge of the problem to assist in the requirements elicitation phase.

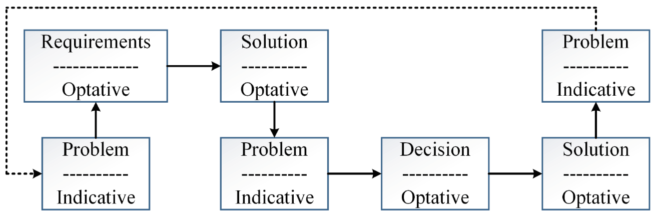

Fig. 7. Indicative and optative properties [23].

6.1 Indicative and Optative Statements

ASRs and ADDs can be represented by indicative and optative statements. They put constraints over the design decisions, and sometimes themselves are constraints over the other design decisions. The description of the problem domain involves indicative specifications. Figure 7 illustrates the process of opting requirements from the problem domain to the problem indicative, and subsequently, the problem indicative can further help in gathering effective requirements for the further phases. In this figure, requirements are extracted from the problem domain, and a loop is formed for the evolutionary software development, which adds experience with every delivery of the product.

Considering an example of using IoT motion tracker, if the hardware has already been selected and its properties are defined then we only need to build its architecture whereas the constraints are only applicable to the requirements. Alternatively, if we have not yet selected the hardware, the ADDs need to consider the hardware as well as the software architecture. In the first situation, the properties of the hardware were given as a part of the problem (indicative), therefore, it constrains the use of software architecture. Whereas, in the second scenario, the properties of the intended hardware need to be chosen (optative) which is part of the solution. Both ASRs and ADDs strongly affect the software system including the preferences for the desired implementation and elimination of the features that are not desirable.

6.2 Architectural Decision Loop

The decision loop illustrates the relationship between ADDs, as shown in Fig. 8 that has been extended from the [23]. It represents that a design decision introduces additional design decisions, which also depends on the previous design decisions hence, creating a decision loop [24]. By using the architectural decision loop ADDs introduce new requirements, and for those requirements, new ADDs need to be considered. Taking an example of flood traffic analysis on an IoT network, multiple use cases can solve this problem, however, we use broadcasting of the alarm when a flooding packet is observed. This endorse further requirements

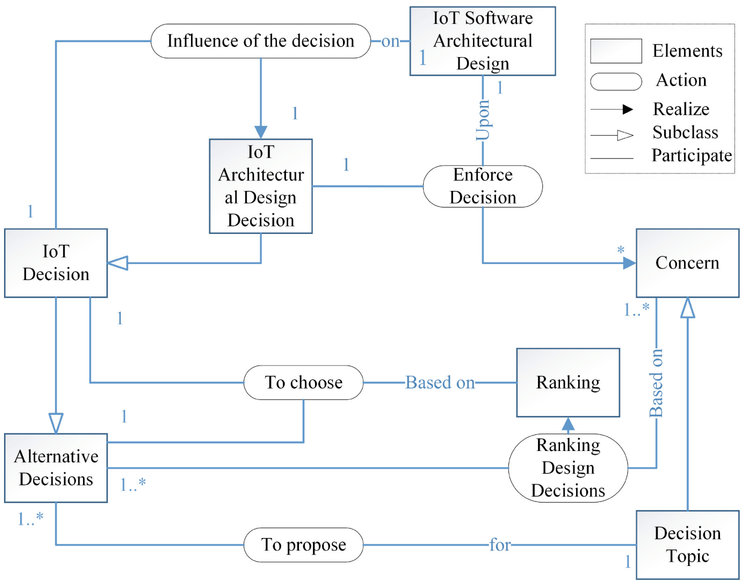

Fig. 8. Architectural decision loop for IoT solutions [23].

of flooding threshold definition, detection technique, and broadcast method to be used. Therefore, the threat broadcast on flood traffic becomes a design decision which has provoked various other design decisions. Another such instance is the storage of IoT generated data, as the IoT devices are limited in capacity, the system keeps track of the storage and transfers the data to the cloud when it exceeds the prescribed limit. Therefore, the decision of the cloud storage invoke new requirements of latency, bandwidth, data rate, and time interval will be provoked. In this regard, we consider this condition as an architectural design decision loop as new requirements will have to follow this design decision.

6.3 Repository of ADDs and Requirements

A solution to select the ADDs and ASRs has been to devise a repository denoted as ADD and Requirement (ADRQ) repository. User's intention dominates the choice of ASRs and ADDs while exploiting the repository. The relationship between the RE to architecture concerning the ADRQ repository has been elaborated in Table 3. Software architect and requirements analyst has domain-specific criteria to contemplate the ADRQ repository, as shown in Table 3. The requirements analyst stores the elicited requirements in the ADRQ in the same way, the solution architect stores the architectural statements in the repository. Requirements that are elicited as well as ADDs should be explicitly documented and stored in the repository to denote the specifications for implementation of the solution.

Table 3. The comparison of RE and architecture in an ADRQ repository.

Requirements domain	ADRQ repository	Architecture domain
Requirements elicitation	Formation of statements	Choice of architecture
Requirements negotiation	Cost-benefit analysis	Architectural exchange analysis
Requirements description	Documenting statements in the repository	Architectural design
Requirements validation	Relate repository statements with clarity	Architecture evaluation
Requirements document	Writing down the repository statements	Architectural explanation
Requirements administration	Organizing the repository	Knowledge management of the architecture

Both architectural requirements and ADDs can be documented using different techniques, including formal language specification, unified modeling language, entity-relationship, and sequence diagrams. The documentation process is extremely necessary to keep track of the software requirements and ADDs efficiently. Both requirements and architecture should analyze the quality of the content which can capture the relationship among ADDs and requirements.

7 Discussion

In this section, we provide a cross-benefit analysis of software requirements and the architecture.

7.1 Requirement Elicitation for Smart Devices

RE is concerned with the elicitation of the goals that a user wants to achieve from the software system. The RE process may involve focused groups, interviews, prototyping, and use cases development. Every requirement is given a relative weight by requirement negotiation process where the software architect selects a particular architecture using the trade-off analysis. In software architecture, the requirements are not processed as they are elicited, but they are less formally represented as they are elicited. Validation is an important component of RE and software architecture. The architecture community has devised various approaches for architectural assessment and their impact on software quality [25]. Moreover, multiple techniques are used for requirements validation, such as the informal technique of review and inspection. Usually, scenario-based methods are used both in architectural assessment and requirements validation.

7.2 Cross-fertilization

In this research, we discuss how software requirements and the architecture community can benefit from each other's experience for effective IoT solution development. The software architecture life-cycle stresses the need for constant interaction among stakeholders to understand their requirements. Business goals and stakeholders' requirements play a major role in architecture development. ADDs and related architectural knowledge plays an important role in architecture management. IoT application development suffers from many challenges, including modifiability, traceability, rationale, and evolution management. Knowledge frameworks can be developed for architectural knowledge management, which also corresponds to the requirements' management. Goal-oriented RE involves goals during the requirements management, whereas, architectural knowledge management includes areas such as traceability, conflicts discovery, and exploring new design variations.

Currently, both requirements and the architectural knowledge management are considered as different information paradigms, however, effective IoT solutions need to consider the similarities of both the fields. We realize that both areas have been addressing the same problem from different perspectives. This study finds that the requirements and architectural knowledge management for IoT solutions need further consideration because of the lack of standard development environments. Further exploration in this field will open new horizons for better requirement management for the IoT architecture where both communities can learn from each other's experience. The illustration of ADRQ repository elaborates that architecture development is not only the responsibility of the software architect, however, ASRs shape the architecture, which also involves the requirement managers concurrently. A maintainable architecture cannot be developed without the consideration of expertise from both the fields.

8 Conclusion

Internet of things utilizes smart infrastructure, big data, communication technologies, and heterogeneous platforms to enable ubiquitousness. The heterogeneity in IoT provokes IoT solutions to integrate with diverse software and hardware platforms. It becomes a critical challenge to develop a maintainable solution that satisfies the requirements of heterogeneous IoT needs. We provide a comprehensive analysis of how software requirements and architectural design decisions can help in developing a maintainable software system. We argue that the architectural design decisions and architecturally significant requirements are on an equal level of significance. This paper plays a key role in characterizing the relationship between architectural design decisions and architecturally significant requirements and recognize them as significant for IoT solution development.

This research opens new horizons towards tighter collaboration between these two paradigms to satisfy the heterogeneous needs of IoT solutions. The analysis of the elicited requirements should be performed by explicitly considering the architecture of the software. Consequently, we can extract architecturally significant requirements which can further be used to develop the architecture. Architectural design decisions play a key role in software evolution and maintenance. Therefore, we should use a proper blend of architecturally significant requirements and architectural design decision to develop maintainable solutions to satisfy heterogeneous IoT demands.

8.1 Future Research

New research directions in the field of IoT architecture modeling can be explored by identifying the significant requirements towards the interdependencies of heterogeneous hardware and software systems. Moreover, these dependencies can be defined in terms of big data needs, communication latency, and bandwidth requirements for IoT solutions.

We are further extending this research by facilitating IoT solution developers by automatic code generation, which reduces cost and time in software development. In this regard, a layered RESTful framework can be employed, which provides platform-specific APIs. Different layers in the RESTful architecture interact to deploy a hardware-specific solution.

Acknowledgment. This research is supported by the National Science Foundation of China under Grant No. 61672276 and 61702277 and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University.

References

1. Salman, O., Elhajji, I., Chehab, A., Kayssi, A.: Iot survey: an SDN and fog computing perspective. Comput. Netw. (2018)
2. Zanella, A., Bui, N., Castellani, A., Vangelista, L., Zorzi, M.: Internet of things for smart cities. IEEE Internet Things J. **1**(1), 22–32 (2014)
3. Sosa-Reyna, C.M., Tello-Leal, E., Lara-Alabazares, D.: Methodology for the model-driven development of service oriented iot applications. J. Syst. Architect. **90**, 15–22 (2018)
4. Swaminathan, J.M.: Big data analytics for rapid, impactful, sustained, and efficient (RISE) humanitarian operations. Prod. Oper. Manag. **27**(9), 1696–1700 (2018)
5. Venters, C.C., et al.: Software sustainability: research and practice from a software architecture viewpoint. J. Syst. Softw. **138**, 174–188 (2018)
6. Uikey, N., Suman, U.: A lifecycle model for web-based application development: incorporating agile and plan-driven methodology. Int. J. Comput. Appl. **117**(19), 28–36 (2015)
7. Sarker, I.H., Faruque, F., Hossen, U., Rahman, A.: A survey of software development process models in software engineering. Int. J. Softw. Eng. Appl. **9**(11), 55–70 (2015)
8. Özdemir, V., Hekim, N.: Birth of industry 5.0: making sense of big data with artificial intelligence "the internet of things" and next-generation technology policy. OMICS J. Integr. Biol. **22**(1), 65–76 (2018)
9. Jackson, M.: Problem frames and software engineering. Inf. Softw. Technol. **47**(14), 903–912 (2005)

10. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture(WICSA), Pittsburgh, Pennsylvania, pp. 109–120 (2005)
11. Márquez, G., Astudillo, H.: Selecting components assemblies from non-functional requirements through tactics and scenarios. In: Proceedings IEEE 35th International Conference of the Chilean Computer Science Society (SCCC), Valparaíso, Chile, pp. 1–11 (2016)
12. Marquez, G., Astudillo, H.: Selection of software components from business objectives scenarios through architectural tactics. In: Proceedings 39th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, pp. 441–444 (2017)
13. Capilla, R., Nava, F., Pérez, S., Dueñas, J.C.: A web-based tool for managing architectural design decisions. ACM SIGSOFT Softw. Eng. Notes **31**(5), 4 (2006)
14. Cai, H., Gu, Y., Vasilakos, A.V., Xu, B., Zhou, J.: Model-driven development patterns for mobile services in cloud of things. IEEE Trans. Cloud Comput. **6**(3), 771–784 (2018)
15. Nguyen, X.T., Tran, H.T., Baraki, H., Geihs, K.: FRASAD: a framework for model-driven IoT application development. In: Proceedings IEEE 2nd World Forum on Internet of Things (WF-IoT), pp. 387–392 (2015)
16. Kwan, I., Cataldo, M., Damian, D.: Conway's law revisited: the evidence for a task-based perspective. IEEE Softw. **29**(1), 90–93 (2012)
17. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Proceedings Future of Software Engineering, pp. 37–54. IEEE Computer Society (2007)
18. Grunbacher, P., Egyed, A., Medvidovic, N.: Reconciling software requirements and architectures: the CBSP approach. In: Proceedings 5th IEEE International Symposium on Requirements Engineering, pp. 202–211 (2001)
19. Mohammadi, N.G., Heisel, M.: A framework for systematic analysis and modeling of trustworthiness requirements using i* and BPMN. In: Katsikas, S., Lambrinoudakis, C., Furnell, S. (eds.) TrustBus 2016. LNCS, vol. 9830, pp. 3–18. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44341-6_1
20. Rosa, W., Madachy, R., Clark, B., Boehm, B.: Agile software development cost modeling for the US DoD. In: SEI Software and Cyber Solutions Symposium, pp. 1–29. Software Engineering Institute (2018)
21. Beck, K., Boehm, B.: Agility through discipline: a debate. Computer **36**(6), 44–46 (2003)
22. Shahbazian, A., Lee, Y.K., Le, D., Brun, Y., Medvidovic, N.: Recovering architectural design decisions. In: Proceedings IEEE International Conference on Software Architecture (ICSA), pp. 95–9509 (2018)
23. De Boer, R.C., Van Vliet, H.: On the similarity between requirements and architecture. J. Syst. Softw. **82**(3), 544–550 (2009)
24. Fitzgerald, B., Stol, K.-J.: Continuous software engineering: a roadmap and agenda. J. Syst. Softw. **123**, 176–189 (2017)
25. Barnes, J.M., Garlan, D., Schmerl, B.: Evolution styles: foundations and models for software architecture evolution. Softw. Syst. Model. **13**(2), 649–678 (2014)