Requirements Development for IoT Systems with UCM4IoT

Paul Boutot[a], Mirza Rehenuma Tabassum[a], Abdul Abedin[a], Sadaf Mustafiz[a,*]

[a]*Department of Computer Science, Toronto Metropolitan University, 245 Church Street, Toronto, M5B 1Z4, ON, Canada*

Abstract

The engineering of IoT (Internet of Things) systems brings about various challenges due to the inherent complexities associated with such adaptive systems. Addressing the adaptive nature of IoT systems in the early stages of the development life cycle is essential for developing a complete and precise system specification. In this paper, we propose a use case-based modelling language, UCM4IoT, to support requirements elicitation and specification of IoT systems. UCM4IoT takes into account the heterogeneity of IoT systems and provides domain-specific language constructs to model the different facets of IoT systems. The language also incorporates the notion of exceptional situations and adaptive system behaviour. Our language is supported with a textual modelling environment to assist modellers in writing use cases. The environment supports syntax-directed editing, validation of use case models, and requirements analysis. The proposed language and tool is demonstrated and evaluated with two case studies: smart store system and smart fire alarm system.

Keywords: Requirements development, Domain-specific modelling languages, Use case modelling, Model-driven engineering, Internet of Things

---

[*]Corresponding author

Email addresses: `pboutot@ryerson.ca` (Paul Boutot), `mirza.tabassum@ryerson.ca` (Mirza Rehenuma Tabassum), `abdul.abedin@ryerson.ca` (Abdul Abedin), `sadaf.mustafiz@ryerson.ca` (Sadaf Mustafiz)

1. Introduction

The Internet of Things (IoT) is bringing about a rapid evolution in the engineering of software systems. IoT is a heterogeneous set of interconnected things (machines, objects, people, animals), middleware, and software (smart) systems. IoT is revolutionizing application domains from consumer and commercial to industrial and infrastructure [1, 2].

The *things* constituting an IoT system have common goals and need to interact and cooperate to fulfill the functionalities of a smart system. The need to communicate with and control physical devices assigns new characteristics to such systems. The many facets of IoT - software, hardware, network, and environment - have to be taken into consideration when specifying the system interactions. Moreover, the adaptive nature of IoT systems introduces complexities and challenges that need to be addressed at the early stages of development. Discovering and documenting the complex interactions between the different participants or entities in such a system is critical for successfully developing the system. Potential exceptional situations need to be identified and possible adaptive behaviour also need to be explored. Deviations from the expected behaviour that are not identified during requirements elicitation might eventually lead to an incomplete or ambiguous system specification during analysis, and ultimately to an implementation that lacks certain functionality, or even behaves in an unreliable way. While the complexity of the requirements is recognized, there is still no established means of capturing the various elements and describing the complex interactions between the different dimensions of such systems.

In requirements engineering, use cases are an established means of discovering and detailing the system requirements. However as reported in [3], "structural defects can occur when use cases are written without following guidelines". A maximally constrained language can help designers define precise and complete requirements.

Previously, we have proposed a domain-specific requirements elicitation environment for IoT systems. We have taken inspiration from the exceptional use cases approach presented in [4] and adapted it for IoT. Our language, UCM4IoT (**U**se **C**ase **M**odelling for **I**nternet **o**f **T**hings) provides a template as well as explicit guidelines for developing textual use case models. It facilitates discovery and specification of exceptional scenarios and adaptive mechanisms. It extends traditional use case models with IoT-specific elements. The language is supported by a modelling environment that en-

ables modellers to write and validate their UCM4IoT models. The use of an extended use case diagram is proposed for summarizing the discovered requirements.

This paper is an extension of [5]. It introduces new features in UCM4IoT along with an additional IoT case study, smart fire alarm system. We present an evaluation of our work using two case studies. We have extended the language and approach with the following concepts and constructs: 1)services and modes of operation, 2) multiplicities of actors, 3) global exceptions, 4) specification of internal system processing steps and further categorization of the internal steps, and 5) explicit definition of exceptions in order to differentiate between the definition and occurrence of an exception. The modelling environment has been extended with new syntax and validation checks to support all new language extensions. We have also added tool support for generation of mode summary tables, handler summary tables as well as a new view of the exception summary table (use case view, in addition to the previous global view). Moreover, we provide more details on the process, approach, and tooling, and have also expanded the related work section with a detailed comparison table.

This paper is structured as follows: Section 2 provides essential background. Section 3 presents our modelling language and Section 4 describes a process to support use of the language. Section 5 discusses the modelling environment. Section 6 and Section 7 demonstrates the application of our language on two IoT case studies. Section 8 evaluates our approach using the case studies. Section 9 discusses potential extensions of the current work. Section 10 includes related work and Section 11 concludes the paper.

2. Background

2.1. Use Cases

Use case modelling is an established means for elicitation and specification of system requirements [6, 7, 8]. Use cases are written in plain text, hence making it well-suited for communication with stakeholders. Use case modelling helps in the discovery process by bringing hidden requirements to the surface. A use case represents a scenario or a description of system behaviour that is required to satisfy a user goal. A goal can be defined at different levels of abstraction: summary, user goal and sub-functional. Use case descriptions (often referred to as textual use cases) includes a *main success scenario* for describing the system interactions leading to a successful

outcome (of a goal) and an *extensions block* for specifying alternate scenarios. Every goal is associated with a set of actors: primary (actor with goal on system), secondary (actors with which the system has a goal and creates value for other actors), or facilitator (actor used by a primary or secondary actor to communicate with the system). Each use case is identified with a name, scope (context), intention (of the primary actor), level, and multiplicity. Use cases are scalable, since they can be decomposed into other use cases. Textual use cases are not part of the Unified Modelling Language (UML) standard. UML use case diagrams give a summarized view of the use cases (scenarios and system goals), relationships among the goals, and associations with external actors (systems, components, or human agents interacting with the system). These diagrams are usually complemented with textual use cases that contain essential information on the interactions. Such use case descriptions are much more appropriate for requirements elicitation over more formal diagrams such as UML activity or state diagrams. In our work, we use the well-defined textual use case template proposed by Fondue [9], a UML-based software development method for reactive systems. A use case model in Fondue comprises of a set of textual use cases and a use case diagram. In comparison to Cockburn's loosely-defined use cases, the strict guidelines and template offered by Fondue make it easier and less confusing for novice users to write complete and precise use cases.

While failure scenarios have been informally discussed in Cockburn's use cases, the notion of exception handling in use cases was introduced in the work on exceptional use cases [4]. Exceptions in use cases refer to exceptional situations that interrupt the normal flow of interaction and may require special handling to be carried out. It should be noted that requirements-level exceptions are not the same as design-level exceptions [10]. Exceptions internal to the system would prevent the system from fulfilling the user goal. Exceptions may also occur in the environment and change the context of the interaction. In both cases, the exceptions need to be detected and addressed by defining handling mechanisms. A special type of use case, referred to as a handler use case, is used to specify the exceptional interactions required to handle an exception. A UML profile for use case diagrams is provided to support creation of diagrams with exceptions and handlers.

The use case model is a representation of the problem space, not the solution space. UML sequence diagrams are also used for interaction modelling, however they are more appropriate for requirements analysis and specification, but not for elicitation. It should be noted that use cases are not the

4

same as user stories used in agile methods. It is difficult to assess if the stories reflect the true reality and whether all essential requirements have been identified. For complex systems (such as, IoT systems), use cases, which give a detailed, clear and unambiguous description of the system behaviour are more appropriate for requirements development.

2.2. IoT Architectural Reference Model

The IoT Architectural Reference Model (ARM) [11] was established within the European research project, IoT-A, as a step towards standardizing a reference model for the IoT domain. In addition to introducing several views (context, functional, information, etc.), a domain model is provided which encompasses the common concepts of IoT systems along with the associations among the entities. The metamodel includes two kinds of users, *human users* and *digital artefacts* (software applications, agents, or services). A user invokes *services* and interacts with *physical entities*. A physical entity may be a type of *device* (sensor, actuator, or tag) along with associated software (network or on-device resources). Notions of virtual entity and augmented entity (composition of a virtual and physical entity) are also part of the domain model.

3. Use Case Modelling Language

There are many facets of IoT systems: software, hardware, network, and the environment in context. *Software* is the core entity behind smart systems. The software controls the system and facilitates the interactions between the system and the environment, hardware and network. *Hardware* includes physical entities or devices that are part of an IoT system and have to be detailed out at the requirements stage to ensure that all interactions with the hardware are considered. The *environment* brings in a lot of complexity as well as uncertainty. An IoT system has to be designed to be aware of the uncertain nature of the environment in which the system is operating. The environment is made up of human users, physical entities, as well as all smart systems/objects/devices that may interact with the system via the Internet. The *network* is the glue that brings together the environment and the system as well as facilitates communication between the various *things* interacting with the system under development.

Unlike traditional software applications, IoT systems should be designed and developed by taking the different facets of IoT into consideration from the

early stages of development. Eliciting IoT requirements is a challenge since typically requirements development techniques do not take into account these aspects. In this section, we propose a domain-specific requirements elicitation language for IoT systems, UCM4IoT. UCM4IoT aims to facilitate the elicitation process by bringing hidden requirements to the surface with the use of IoT-specific constructs. UCM4IoT allows specification of the requirements with a set of textual use cases (a UCM4IoT model). A use case diagram language with IoT-specific extensions is also proposed for summarizing the requirements.

As part of the requirements development phase, a domain model (a structural view of the system) should be developed along with the use case model (outside the scope of this paper).

3.1. Actors in UCM4IoT

Use cases elaborate on the interactions of the system with the participating primary and secondary actors. The possible types of actors that are part of an IoT system are discussed here. The definitions of the types are according to the IoT Architectural Reference Model [11].

Human User: The user of the system is typically a human user. This is a type of primary actor that initiates or drives system interactions. Human users may also be secondary actors, i.e. actors participating in fulfilling the goal of a primary actor.

Physical entity: These entities are part of the physical environment and play a role in satisfying a user's goal. A physical entity can be any object in the environment, for instance, living entities (humans, animals), moving objects (vehicles), immobile objects (buildings, stores, factories), electronics (appliances, computer, mobile devices), equipment, personal items (clothes, shoes), edible items (food). A physical entity has one or more associated devices.

Leaf Device: This is a special kind of IoT element associated with a physical object. The devices form a core part of a smart system and can be of three basic types: sensors, actuators, and tags. We refer to a leaf device as a *device* in this paper. *Sensors* are used to monitor physical entities and provide information or data on the entity. Special types of sensors, known as *readers*, can also be used to identify an entity. *Actuators* are used to change the physical state of an entity (e.g. switching on a robotic vacuum or moving the direction of the vacuum). *Tags* are attached to entities to enable identification of the object (e.g., a barcode label or a RFID tag). The identification

is carried out with the use of *readers*. A device may be embedded within a physical entity (e.g., an actuator in a robotic vacuum, a sensor within a human body), attached to an object (e.g., a barcode tag on a store item), placed close to an object (e.g., a temperature sensor placed near a plant), or placed in the operating environment (e.g., a sensor placed in a room to detect humidity levels, a face-recognition enabled camera).

Software: Typically, communication with IoT devices is carried out with the use of a software application. Such an application is usually mediated by a human user. However, it might be running automatically to monitor a smart environment by gathering and analyzing data (e.g., monitoring temperature in a smart farm) or to control an environment (e.g., to automatically start a sprinkler system at a given time).

Physical entities in the IoT world typically need a virtual entity. The physical along with the virtual entity represents the "thing" in the Internet of Things. At the use case level, the goal is to elicit requirements based on the problem space or the application domain. Concerns related to the solution space is not taken into account at this stage. Hence, the virtual aspect is not considered in our use cases. A physical entity identified during requirements elicitation will need to be associated with a corresponding virtual entity in the design phase.

Having specific types of actors constrains the modeller to IoT-specific concepts when developing the use cases. The system can interact with actors of specific types and the interactions in the use cases can be validated to ensure that only permitted types of actors are part of the scenarios.

3.2. Exceptions and Handlers in UCM4IoT

Abnormal or exceptional situations are identified with defined *exceptions* in the use case model. Such exceptions identify situations that occur unexpectedly, interrupt the normal flow of interaction, and require the system to tolerate and possibly adapt to the new circumstances. Unless these exceptions are discovered and documented, no special handling will be incorporated in the system, leading the system to have missing functionality and to behave in unexpected ways in such situations.

In this work, we classify exceptions that can occur in an IoT system to be of the following types: hardware, software, network, and environment. A *hardware exception* denotes abnormal behaviour originating in a physical entity or device in the system (e.g., sensor failure, actuator failure). A *software exception* represents unexpected behaviour of a software system or subsystem

participating in fulfilling a user's goal (e.g., credit card system down). This does not refer to exceptions in the software system under development, which come up later in the design and implementation phase. At the requirements development stage, the focus is on "what" the system should be doing, not on "how" it should be done. A *network exception* occurs when a system or *thing* is unreachable, potentially due to network issues (e.g., fire station alarm unreachable, robotic vacuum unreachable). An *environment exception* represents an unexpected situation occurring in the external environment in which the system is operating. Such an exception would typically interrupt and suspend or terminate a user's goal (e.g., a fire hazard in a smart store).

An exception can occur in more than one use case. Each occurrence of an exception is regarded as a different instance of the same exception - the origin or source of the exceptional situation is not the same. However, some exceptions (for instance, a fire hazard) can occur anytime and impact many use cases. In UCM4IoT, these exceptions are classified as global exceptions. While an environment exception typically would be identified as a global exception, other exception types can also be identified as global. As an example, a network exception (no internet connectivity) in the smart fire alarm system, would impact several user goals at the same time.

The environment is not under our control. Unaddressed uncertainties arising in the environment can lead to incomplete requirements and eventually missing functionalities in the target system. In the case of hardware, all hardware are prone to failure. Depending on the quality of service requirements of a system, fault tolerance mechanisms are incorporated at the design and implementation levels to deal with hardware failures.

In an adaptive system, an exceptional situation identified during the execution of a user's goal would require the system to handle and attempt to recover from the situation in the most feasible way. The adaptive behaviour of the system needs to be detailed by specifying the system interactions in such situations. In our language, we use a special type of use case, a handler use case, to specify the steps to be completed to handle a certain exception. When an exceptional situation occurs, the base behaviour is put on hold and the interactions in the handler are initiated. A handler can temporarily take over the system interaction, perform some compensation tasks, and then return to the normal interaction scenario. In cases when the handler cannot help recover from the exception, the exception would cause the goal that was interrupted to fail. At times, a new actor, referred to as an *exceptional actor*, may participate in the handling actions. These actors are introduced

in handler use cases.

3.3. Services and Modes

As defined by The Open Group, a service is a logical representation of a set of activities that has specified outcomes, is self-contained, may be composed of other services, and is a "black-box" to consumers of the service [12]. Services are reusable components of a system that are independent and loosely coupled [13].

While developing a system, designers often try to reuse existing services that can be used to fulfill some goals of the system. After the requirements elicitation phase, designers identify existing services from service repositories to meet business requirements, and additionally develop required services for the system. However, often services are reused that do not meet the business requirements fully. Reusing a service that partially matches the requirements results in an inaccurate system [14]. Services must be aligned with the requirements, not vice versa.

Use cases describe actors and system's interaction without explaining the internal technical details of that system. As use cases lay out goals of the system, and services are developed to meet user goals, the use cases are used as a tool for describing operational activities in service oriented architecture [15]. Goals are specified from the perspective of the users of the system, while services are specified from the system's perspective.

A system can operate in different modes in different circumstances. A mode is defined by the set of services that is exposed by the system while operating in a certain mode[16]. On the event of an exceptional situation, a system may not be able to provide all services, or it may only be able to provide exceptional or emergency services. In such cases when future service provision may be impacted, the system would switch from a normal mode of operation to a degraded, restricted, or emergency mode of operation [17].

UCM4IoT accommodates different modes of operation in the use cases. The categorization of modes in UCM4IoT is based on the definition of modes proposed in [17]. A system operates its day to day business in the `normal` mode. In case of an emergency, the system may need suspend normal operation and offer some special services to address and handle the scenario by switching to an `emergency` mode. There may be special circumstances when the system can only offer limited services (a subset of the normal services) as well as required emergency services, putting the system in a `restricted`

mode. System can also operate in a `degraded` mode when certain services are unavailable or offered with reduced quality-of-service.

There is an implicit connection between goals, services, and modes. Services are offered by the system to achieve the goals of actors, and different modes offer different sets of services. Hence, meeting goals of the system also depends on which mode the system is in at that time. In UCM4IoT, the default mode is the `normal` mode, hence all use cases, except handler use cases start in a `normal` mode. However, a mode switch can occur during the execution of a use case or handler use case. A step in the alternate or exceptional block may also trigger a mode switch. Multiple mode switches are also possible within a use case in UCM4IoT. Moreover, it may also be possible for a system to offer more than one normal mode of operation.

3.4. Representing Exceptions, Handlers, and Modes in Use Cases

Similar to a standard use case, a use cases in UCM4IoT includes a header with details on the name, scope, level (summary/user-goal/sub-function), intention, and multiplicity. Each use case has a set of associated actors categorized as primary, secondary, or facilitator. The syntax for specifying actors is Type::UniqueName (e.g., `Human::Customer`, `Sensor::Weight`). The body of the use case is composed of the main scenario (outlining the normal system interactions leading to a success of the user goal) and an extensions block (outlining the alternative paths and/or exceptional cases). A step in an use case can be of the following types: interaction (send or receive requests between the system and actors), invocation (calls another use case), condition (an interaction occurring in the environment), internal (internal system processing on which the following steps are dependent or a timeout specification), or control flow (a step dictating the flow of interactions). The main success scenario as well as each extension block have an associated outcome: success (denoting a successful execution of a goal), failure (denoting a failed execution of a goal), degraded (denoting a partially successful execution of a goal), or abandoned (denoting voluntary termination of a goal by a user).

Exceptions are defined in the extensions block with the type and name of exception (e.g., `EnvironmentException::FireHazard`). In case of global exceptions, `::global` is added to the exception name. The UCM4IoT use case template is shown with the use of an example in Fig. 8. Exceptions can be associated with single steps or a block of steps. If a step (or a block) includes an invocation, the exception can also be raised during the execution of the invoked use case (see Fig 7). Every handler is associated with the

context (use case in which the exception appears) along with the exceptions, which is explicitly stated in the header with the `contexts & exceptions` clause (see example in Fig. 10).

Modes are specified in use cases with the `mode switch` keyword. If a use case in normal mode switches to restricted mode after an exception is raised, it can be specified using the syntax `mode switch: Restricted`.

An exception can trigger a mode switch, then the handler for the exception operates in that switched mode.

The relationship between handlers and use cases can be one of two types: 1) *interrupt & continue*: in this case, the user goal (including all sub-goals) is put on hold due to an exceptional situation - the use case execution resumes following the handling; or 2) *interrupt & fail*: when the user goal (including all sub-goals) fail due to an exceptional situation - the use case execution is terminated. The UCM4IoT metamodel is presented in Fig. 1.

Figure 1: UCM4IoT metamodel.

For summarizing the outcome of the requirements elicitation process, we have adapted and used the exception-aware use case diagram proposed in [4]. Standard UML use case diagrams have been extended with exceptions, handler use cases (a subclass of Use Case) and interrupt relationships (similar to Extend and Include relationships but specifically uses handler use cases as the source). The relationship between handlers and use cases are tagged with `<<interrupt & continue>>` or `<<interrupt & fail>>`. These dependencies are depicted in the extended use case diagram. We have used this UML profile with the IoT-specific elements and syntax introduced in UCM4IoT for specifying actors and exceptions (see example in Fig. 11). The profile is not included here due to space constraints.

11

4. UCM4IoT Process

While the UCM4IoT language can be used for requirements elicitation according to the modeller's need, we provide here some recommended guidelines.

4.1. Inception

The requirements development process starts off with identifying business needs and understanding the problem and stakeholders. A problem statement followed by an initial set of hardware and software requirements (an informal natural language specification) is to be developed by gathering domain knowledge and analyzing the stakeholders and existing systems or documentation.

4.2. Elicitation

The use case-based requirements elicitation process begins by identifying the major stakeholders of the system under development (referred to as `System` in the use cases) and by identifying their goals with the system. Based on the problem domain and communication with stakeholders, establish goals and services that the system needs to fulfill. The primary actor is typically a human user, however in IoT systems, software and hardware systems may also initiate the system interactions. Based on the UCM4IoT actor types, the secondary and facilitator actors (humans, physical entities, software, devices) participating in a goal are identified. With the help of the UCM4IoT process guidelines, identify scenarios that lead to the success of the goals. The normal behaviour of the system in order to fulfill the goal is described in the main success scenario of the use case. Next establish all alternative behaviour that may arise in the system.

The language constructs should be used to identify different types of actors (human, physical entity, leaf device, and software). In this phase, exceptional behaviour also needs to be discovered based on the UCM4IoT language. Having the different types of actors make it easier to define granular interactions ensuring that every interaction between the system and the associated devices, physical entities, external software systems, or other human users are present in the use case. Relevant internal context steps or timeouts associated with each goal should also be specified.

The system provides various services that fulfill one or more user goals. The services to be offered by the system need to be derived from the use cases established.

In addition to discovering actors and goals, the possible modes of operation supported by the system needs to be established. This includes normal modes as well as special modes (restricted, degraded, emergency) that the system may need to switch to in order to provide special or limited services.

4.3. Elaboration

This activity focuses on analyzing and elaborating the requirements. Once the normal behaviour is defined, each interaction in the main scenario needs to be revisited to explore possible exceptional situations that may arise. The following questions can be used to identify exceptional situations: *Can this interaction step fail in any way? What might be possible sources of failure? Are there any exceptional situations that may arise due to problems with the hardware, software, or network? Can any external event originating in the operating environment change or fail the user's goal?* Having explicitly defined types for the exceptions, makes modellers aware of the different aspects associated with a given interaction and assists them in identifying potential hardware, software, network, or environment exceptions that would otherwise not be discovered using standard use cases.

As the next step, handling mechanisms for each identified exception should be explored. The following probes can be used to elicit handlers: *How should the system react and adapt in such an exceptional situation? What additional system functionality is required to enable the system to continue fulfilling the user's goal in a such situation? Does the system need the help of an (exceptional) actor to handle the situation?* In the case of hardware exceptions, a possible handling mechanism would be to call a service person for repairing or replacing the device. Software exceptions are due to issues with an external system, and hence in such scenarios, there should be some compensation activity carried out by the system so that the system does not fail to fulfill the user's goal. Network exceptions can be handled by periodically attempting to reconnect or by trying to communicate with the actor using some alternate means. To address environment exceptions, there may be a need to support such exceptional situations with added system functionality that was not part of the initial problem statement and desired system features.

The handlers should also be analyzed for exceptional behaviour. The handlers need to be documented. Designers should ensure that all outcomes are clearly defined in the model. UCM4IoT gives warnings and errors to assist in the completion of the use case model.

4.4. Negotiation/Prioritization

Static analysis with UCM4IoT: The UCM4IoT tool should be used to generate the mode table, exception summary table, and handler summary table. Based on the use cases, UCM4IoT generates a summary of paths that result in an exception being thrown. The tool also generates summary information on handlers and their associated use cases. The criticality of use cases, exceptions and handlers can be determined based on the generated information and higher weight or priority can be attached to the corresponding functionality when design decisions are taken later on expected quality-of-service and fault tolerance measures to be used.

4.5. Specification

The UCM4IoT model (textual uses cases and the use case diagram), the accompanying summary tables and the IoT ARM domain model are all part of the requirements specification of the system.

4.6. Validation

Requirements validation involves examining the requirements model for inconsistency, omissions, and ambiguity. The use cases model needs to be validated to ensure that there are no inconsistencies in the use cases, for instance, missing or mismatched actor types, missing use cases or handlers, ambiguities in the definition or use of actors and exceptions.

The UCM4IoT modelling environment, described in Section 5, further assists developers in eliciting and defining use cases, exceptions, and handlers for IoT systems.

5. Tool Support

The UCM4IoT environment assists modellers in developing requirements for IoT systems with textual use cases. The environment was built in Eclipse utilizing its IDE Plug-in support and Xtext[1] which supports the development of textual modelling languages. Xtend[2] was used for building support for validation and summary information generation.

An UCM4IoT model consists of two types of elements: standard use cases and handler use cases. Both types require several clauses. Specifically, the

---

[1]https://www.eclipse.org/Xtext/
[2]https://www.eclipse.org/Xtend/

environment enforces that the use cases include the scope, level, intention, multiplicity, primary actor, and main success scenario clauses. It also supports optional clauses that the two types of use cases can utilize. These include the extensions, secondary actor, facilitator actor, precondition, and post-condition clauses. Additionally, the tool specifies that handler use cases require one other clause, namely, the *context and exceptions* clause. An UCM4IoT model also includes a header located at the top of the model. This header consists of a list of modes and exceptions used throughout the UCM4IoT model.

Our modelling environment provides support for syntax highlighting, refactoring, type-checking, cross-referencing, validation, and data generation. Specifically, the grammar specifies the keywords of the modelling language, the allowed types for actors and exceptions, and the linking of references to their respective declaration. Moreover, model validation and data generation are carried out in the backend. Figure 7 shows an example of syntax highlighting and type-checking. Notably, UCM4IoT highlights all of the clauses and types.

UCM4IoT limits all steps found in the main success scenario or the extensions to either an invocation, interaction, control-flow, condition, internal step, or exception. Extensions consist of extension blocks, which then contain steps or more extension blocks. Extension blocks represent alternative paths from a declared step in the use case. These blocks can be either exceptional or alternative blocks. The main difference is that exceptional blocks contain one step that throws an exception. The environment will throw an error if the modeller fails to conform to the provided grammar.

UCM4IoT also lets users define the different modes of the system along with the mode switches. The user lists the default mode and all other modes in the header of the UCM4IoT model. Afterwards, the user can reference these modes in the main success scenario or any extension block to specify when the system switches into these modes. The system can only switch into a mode at the beginning or end of the main scenario or an extension block.

The following outlines the features of our environment.

Step Ordering and Formatting: For the steps defined in the main success scenario and the extensions, UCM4IoT provides validation checks for their ordering and format. Any numbered step must follow the logical ordering of the previous step. Thus, this simplifies the creation of the main success scenario and the extensions as the environment notifies modellers when the sequence of steps is no longer correct. If a step does not follow the logical

15

ordering of its previous step, UCM4IoT will throw an error message along with a list of possible step numbers that can occur after the last numbered step. For invocation steps, our environment enforces that the modeller states the invoked use case of the step. Our environment cross-references the invoked use case, giving an error for use cases that are not defined. Likewise, exception steps require modellers to define the raised exception in the header (as seen in Fig. 3). Any exceptions that do not exist in the header result in the environment giving an error. Lastly, internal steps provide the modeller with the option to describe *timeouts* in the system, denoted with a keyword and the amount of time (see Fig. 2). The environment will give an error if the modeller incorrectly uses this feature.

```
extensions:
    exception for 2:
        2a. ^ timeout:60s "SprinklerSystem fails to acknowledge
```

Figure 2: UCM4IoT Syntax: Internal step specifying a *timeout.*

UCM4IoT also enforces that all interaction steps found in summary-level and user goal-level use cases include the *system* and an actor from the list of actors defined in the use case. Lastly, UCM4IoT also features the ability to cross-reference steps. Any step mentioned in an outcome of a use case or the header of an extension block will be linked to an existing step found in the use case.

Type Checking: The tool checks to see if all actors in a use case match one of the possible valid types (human, software, device, or physical entity). Moreover, an actor can also be typed as a sensor, actuator, tag, or reader by the grammar; however, these types are considered as device actors by UCM4IoT. The environment provides an error for any actor or exception that forgets to include a type, for example `EntryGate` instead of `PhysicalEntity::EntryGate`. This clearly shows the type of the participating actor. UCM4IoT also enforces that all exceptions have their associated type defined.

Multiplicity: UCM4IoT allows for modellers to specify the multiplicity of actors. The environment also checks the format of the multiplicity. Its lower bound must be lower than or equal to the upper bound. Moreover, the modeller may denote the range as "*" to indicate any amount. Syntax checks are carried out by the environment.

Exception Handling: For use case handlers, UCM4IoT checks if the de-

16

fined contexts and exceptions are valid. If the exception does not appear in the use case context, the environment notifies the modeller about this issue (see Fig. 4: bottom). Furthermore, UCM4IoT checks that exceptions that appear in a use case are actually handled in some use case handler. Otherwise, a warning is displayed (see Fig. 4: middle). Thus, the environment prevents the modeller from forgetting any unhandled exceptions.

UCM4IoT also requires modellers to define exceptions separately from their occurrence within use cases. An exception is defined in the header, and the modeller can choose to reference that exception within an exception step. It is necessary to distinguish between the definition and the occurrence of an exception so that multiple exception steps may refer to the same exception; otherwise, if two exception steps include the same exception, they would be referring to two unique definitions of exceptions that happen to have the same name. This difference is significant within the environment since the latter option prevents UCM4IoT from properly identifying handled exceptions. Since both exceptions are considered to be distinct, a handler use case attempting to refer to a raised exception would not be able to handle all occurrences of that exception. By requiring modellers to define exceptions in the header and raising them in use cases, UCM4IoT can link handled exceptions to their occurrence as all occurrences of an exception will be the same exception.

```
!List Exceptions:
    {HARDWARE_EXCEPTION::PressureSensorUndetected},
    {HARDWARE_EXCEPTION::WeightSensorUnavailable},
    {HARDWARE_EXCEPTION::TagUnavailable},
    {HARDWARE_EXCEPTION::EntryFailure},
    {HARDWARE_EXCEPTION::ExitFailure},
    {HARDWARE_EXCEPTION::CameraFailure},
    {SOFTWARE_EXCEPTION::ImageUnmatched},
    {NETWORK_EXCEPTION::CustomerUnreachable},
    {ENVIRONMENT_EXCEPTION::EnvironmentExceptionName},
    {SOFTWARE_EXCEPTION::PaymentServiceDown},
    {NETWORK_EXCEPTION::CriminalAttack::global},
    {ENVIRONMENT_EXCEPTION::FireHazard::global}
```

Figure 3: UCM4IoT Syntax: Exceptions defined in header.

Outcomes: UCM4IoT enforces that all main scenarios, exceptional blocks, and alternative blocks end in some outcome. Outcomes must either end in success, failure, degraded, abandoned, or continue with some other step. The main success scenario only has one outcome that always ends in success. In contrast, the extensions clauses can have many alternate steps, meaning that they can have many outcomes. This feature prevents the modeller from

forgetting the outcome of any scenario. The tool also provides validation checks for the outcome of an exception block. Specifically, if there is no handler for a raised exception when an exception block continues to another step, the environment would display an error message stating that the use case cannot continue as the exception is never handled.

Figure 4: Sample validation checks.

Table Generation:

UCM4IoT provides the generation of exception summary tables for each use case within a UCM4IoT model. Any exception that could occur in a use case (including those found in any invoked use case) will be included along with the source use case of the exception, the handler that handles the exception, the list of possible situations that result in the exception occurring, and the actors that participated in the event that caused the exception to be thrown. A slice of an exception summary table is shown in Fig. 5. It is possible to generate two different views: a global view and a use case view. A global view contains information about all of the exceptions within a UCM4IoT model. Comparatively, a relative view contains information about exceptions that can be raised when looking at a particular use case. This generated information can then be used for making inferences and carrying out application-specific analysis (discussed further in Sections 6 and 7). Future iterations will allow users to import use cases from other UCM4IoT files, allowing support for larger-scale projects.

18

UCM4IoT also supports the generation of handler tables for every handler within a UCM4IoT model. The table describes which use cases depend on the handler, the exceptions that it handles, and any exceptional actors that appear (denoted with a '*'). A snippet of a handler table is shown in Fig. 6. Moreover, the tool can generate a mode table listing names of all use cases along with mode switches. Fig. 12 shows the smart store system's mode switches.

A demo video of our environment is available at
`https://www.cs.ryerson.ca/~pboutot/ucm4iot-new.html`.
Interoperability:

Moreover, the environment generates an exportable .xmi version of UCM4IoT to enable UCM4IoT models to be ported to other modelling tools.

Exception Type	Exception Name	Exception Source	Associated Actors	Handler	Interrupt Type	Path
ENVIRONMENT_EXCEPTION	CriminalAttack	Shopping ExitStore MaintainStore CheckOut	PHYSICAL_ENTITY::MobileDevice	{AlertOnAttack}	Interrupt and Continue (2) Interrupt and Fail (2)	Shopping >> {AlertOnAttack} !I&F GLOBAL::EnterStore >> {AlertOnAttack} !I&F GLOBAL::ScanMobileDeviceOnEntry >> {AlertOnAttack} !I&F GLOBAL::AddToCart >> {AlertOnAttack} !I&F GLOBAL::IdentifyItem >> {AlertOnAttack} !I&F GLOBAL::RecognizeCustomer >> {AlertOnAttack} !I&F GLOBAL::RemoveItem >> {AlertOnAttack} !I&F GLOBAL::ExitStore >> {AlertOnAttack} !I&F GLOBAL::ScanMobileDeviceOnExit >> {AlertOnAttack} !I&F GLOBAL::PayBill >> {AlertOnAttack} !I&F GLOBAL::ReturnItem >> {AlertOnAttack} !I&F ExitStore >> {AlertOnAttack} !I&C GLOBAL::ScanMobileDeviceOnExit >> {AlertOnAttack} !I&C GLOBAL::PayBill >> {AlertOnAttack} !I&C GLOBAL::RemoveItem >> {AlertOnAttack} !I&C GLOBAL::IdentifyItem >> {AlertOnAttack} !I&C GLOBAL::RecognizeCustomer >> {AlertOnAttack} !I&C MaintainStore >> {AlertOnAttack} !I&F GLOBAL::CheckIn >> {AlertOnAttack} !I&F GLOBAL::ScanMobileDeviceOnEntry >> {AlertOnAttack} !I&F GLOBAL::RestockShelves >> {AlertOnAttack} !I&F GLOBAL::IdentifyItem >> {AlertOnAttack} !I&F GLOBAL::RecognizeStaff >> {AlertOnAttack} !I&F GLOBAL::CheckOut >> {AlertOnAttack} !I&F CheckOut >> {AlertOnAttack} !I&C
	FireHazard	Shopping ExitStore MaintainStore CheckOut	PHYSICAL_ENTITY::MobileDevice	{HandleFireHazard}	Interrupt and Continue (2) Interrupt and Fail (2)	Shopping >> {HandleFireHazard} !I&F GLOBAL::EnterStore >> {HandleFireHazard} !I&F GLOBAL::ScanMobileDeviceOnEntry >> {HandleFireHazard} !I&F GLOBAL::AddToCart >> {HandleFireHazard} !I&F GLOBAL::IdentifyItem >> {HandleFireHazard} !I&F GLOBAL::RecognizeCustomer >> {HandleFireHazard} !I&F GLOBAL::RemoveItem >> {HandleFireHazard} !I&F GLOBAL::ExitStore >> {HandleFireHazard} !I&F GLOBAL::ScanMobileDeviceOnExit >> {HandleFireHazard} !I&F GLOBAL::PayBill >> {HandleFireHazard} !I&F GLOBAL::ReturnItem >> {HandleFireHazard} !I&F ExitStore >> {HandleFireHazard} !I&C GLOBAL::ScanMobileDeviceOnExit >> {HandleFireHazard} !I&C GLOBAL::PayBill >> {HandleFireHazard} !I&C GLOBAL::RemoveItem >> {HandleFireHazard} !I&C GLOBAL::IdentifyItem >> {HandleFireHazard} !I&C GLOBAL::RecognizeCustomer >> {HandleFireHazard} !I&C MaintainStore >> {HandleFireHazard} !I&F GLOBAL::CheckIn >> {HandleFireHazard} !I&F GLOBAL::ScanMobileDeviceOnEntry >> {HandleFireHazard} !I&F GLOBAL::RestockShelves >> {HandleFireHazard} !I&F GLOBAL::IdentifyItem >> {HandleFireHazard} !I&F GLOBAL::RecognizeStaff >> {HandleFireHazard} !I&F GLOBAL::CheckOut >> {HandleFireHazard} !I&F CheckOut >> {HandleFireHazard} !I&C
HARDWARE_EXCEPTION	PressureSensorUndetected	IdentifyItem	DEVICE::PressureSensor	{ServiceSensor}	Interrupt and Continue	UseSmartStore >> Shopping >> AddToCart >> IdentifyItem >> {ServiceSensor} !I&C UseSmartStore >> Shopping >> AddToCart >> RemoveItem >> IdentifyItem >> {ServiceSensor} !I&C UseSmartStore >> Shopping >> ExitStore >> ScanMobileDeviceOnExit >> PayBill >> RemoveItem >> IdentifyItem >> {ServiceSensor} !I&C HoldPayment >> PayBill >> RemoveItem >> IdentifyItem >> {ServiceSensor} !I&C WorkAtSmartStore >> MaintainStore >> RestockShelves >> IdentifyItem >> {ServiceSensor} !I&C
	WeightSensorUnavailable	IdentifyItem	DEVICE::WeightSensor	{ServiceSensor}	Interrupt and Continue	UseSmartStore >> Shopping >> AddToCart >> IdentifyItem >> {ServiceSensor} !I&C UseSmartStore >> Shopping >> AddToCart >> RemoveItem >> IdentifyItem >> {ServiceSensor} !I&C UseSmartStore >> Shopping >> ExitStore >> ScanMobileDeviceOnExit >> PayBill >> RemoveItem >> IdentifyItem >> {ServiceSensor} !I&C HoldPayment >> PayBill >> RemoveItem >> IdentifyItem >> {ServiceSensor} !I&C WorkAtSmartStore >> MaintainStore >> RestockShelves >> IdentifyItem >> {ServiceSensor} !I&C
	TagUnavailable	IdentifyItem	DEVICE::TagReader	{ServiceSensor}	Interrupt and Continue	UseSmartStore >> Shopping >> AddToCart >> IdentifyItem >> {ServiceSensor} !I&C UseSmartStore >> Shopping >> AddToCart >> RemoveItem >> IdentifyItem >> {ServiceSensor} !I&C UseSmartStore >> Shopping >> ExitStore >> ScanMobileDeviceOnExit >> PayBill >> RemoveItem >> IdentifyItem >> {ServiceSensor} !I&C HoldPayment >> PayBill >> RemoveItem >> IdentifyItem >> {ServiceSensor} !I&C WorkAtSmartStore >> MaintainStore >> RestockShelves >> IdentifyItem >> {ServiceSensor} !I&C
	EntryFailure	EnterStore CheckIn	PHYSICAL_ENTITY::EntryGate	{ServiceGate}	Interrupt and Continue	UseSmartStore >> Shopping >> EnterStore >> {ServiceGate} !I&C WorkAtSmartStore >> MaintainStore >> CheckIn >> {ServiceGate} !I&C
	ExitFailure	ExitStore CheckOut	PHYSICAL_ENTITY::ExitGate	{ServiceGate}	Interrupt and Continue	UseSmartStore >> Shopping >> ExitStore >> {ServiceGate} !I&C WorkAtSmartStore >> MaintainStore >> CheckOut >> {ServiceGate} !I&C
	CameraFailure	EnterStore CheckIn	DEVICE::Camera	{RequestUser}	Interrupt and Continue	UseSmartStore >> Shopping >> EnterStore >> {RequestUser} !I&C WorkAtSmartStore >> MaintainStore >> CheckIn >> {RequestUser} !I&C
NETWORK_EXCEPTION	CustomerUnreachable	ScanMobileDeviceOnExit	HUMAN::Customer	{GetResponse}	Interrupt and Continue	UseSmartStore >> Shopping >> ExitStore >> ScanMobileDeviceOnExit >> {GetResponse} !I&C
SOFTWARE_EXCEPTION	ImageUnmatched	RemoveItem RecognizeCustomer RecognizeStaff RestockShelves	DEVICE::Camera	{RequestCamera}	Interrupt and Continue	UseSmartStore >> Shopping >> AddToCart >> RemoveItem >> {RequestCamera} !I&C UseSmartStore >> Shopping >> ExitStore >> ScanMobileDeviceOnExit >> PayBill >> RemoveItem >> {RequestCamera} !I&C HoldPayment >> PayBill >> RemoveItem >> {RequestCamera} !I&C UseSmartStore >> Shopping >> AddToCart >> RecognizeCustomer >> {RequestCamera} !I&C UseSmartStore >> Shopping >> AddToCart >> RemoveItem >> RecognizeCustomer >> {RequestCamera} !I&C UseSmartStore >> Shopping >> ExitStore >> ScanMobileDeviceOnExit >> PayBill >> RemoveItem >> RecognizeCustomer >> {RequestCamera} !I&C HoldPayment >> PayBill >> RemoveItem >> RecognizeCustomer >> {RequestCamera} !I&C WorkAtSmartStore >> MaintainStore >> RestockShelves >> RecognizeStaff >> {RequestCamera} !I&C WorkAtSmartStore >> MaintainStore >> RestockShelves >> {RequestCamera} !I&C
	PaymentServiceDown	PayBill	SOFTWARE::PaymentService	{HoldPayment}	Interrupt and Fail	UseSmartStore >> Shopping >> ExitStore >> ScanMobileDeviceOnExit >> PayBill >> {HoldPayment} !I&F HoldPayment >> PayBill >> {HoldPayment} !I&F

Figure 5: Sample exception summary table: global view.

19

```
+----------------+------------------+--------------------------------------+----------------------------------------+
| Handlers       | Dependent Use Cases | Handled Exceptions                 | Actors                                 |
+----------------+------------------+--------------------------------------+----------------------------------------+
| HandleFireHazard| Shopping         | ENVIRONMENT_EXCEPTION::FireHazard    | HUMAN::Customer                        |
|                |                  |                                      | SOFTWARE::FireDetectionSystem          |
|                |                  |                                      | HUMAN::Staff                           |
|                +------------------+--------------------------------------+----------------------------------------+
|                | MaintainStore    | ENVIRONMENT_EXCEPTION::FireHazard    | SOFTWARE::Staff                        |
|                |                  |                                      | SOFTWARE::FireDetectionSystem          |
|                |                  |                                      | HUMAN::Staff                           |
|                +------------------+--------------------------------------+----------------------------------------+
|                | ExitStore        | ENVIRONMENT_EXCEPTION::FireHazard    | HUMAN::Customer                        |
|                |                  |                                      | PHYSICAL_ENTITY::ExitGate              |
|                |                  |                                      | SOFTWARE::FireDetectionSystem          |
|                |                  |                                      | HUMAN::Staff                           |
|                +------------------+--------------------------------------+----------------------------------------+
|                | CheckOut         | ENVIRONMENT_EXCEPTION::FireHazard    | HUMAN::Staff                           |
|                |                  |                                      | PHYSICAL_ENTITY::ExitGate              |
|                |                  |                                      | PHYSICAL_ENTITY::MobileDevice          |
|                |                  |                                      | SOFTWARE::FireDetectionSystem          |
+----------------+------------------+--------------------------------------+----------------------------------------+
| AlertOnAttack  | Shopping         | ENVIRONMENT_EXCEPTION::CriminalAttack| HUMAN::Customer                        |
|                |                  |                                      | HUMAN::Staff                           |
|                |                  |                                      | PHYSICAL_ENTITY::EmergencyExit         |
|                |                  |                                      | PHYSICAL_ENTITY::ExitGate              |
|                |                  |                                      | PHYSICAL_ENTITY::EntryGate             |
|                |                  |                                      | PHYSICAL_ENTITY::PoliceStation         |
|                |                  |                                      | PHYSICAL_ENTITY::AttackAlertSwitch     |
+----------------+------------------+--------------------------------------+----------------------------------------+
```

Figure 6: Sample handler summary table (slice).

6. Case Study 1: Smart Store System

In this section, the use of the UCM4IoT language is demonstrated with a case study, namely the smart store system.

6.1. Smart Store System Overview

The retail industry is being revolutionized in the IoT era with intelligent shopping, smart carts, and smart stores [18, 19].

A smart store system is a brick and mortar, checkout-free, "walkout" store. *Just walk out* technology (https://justwalkout.com), introduced by Amazon, allows customers to pick up items from the store and walk out without going through the usual checkout process. The system automatically processes payments by identifying items taken by the customers. Implementing this system requires automatic detection of various items in the store as well as recognizing customers purchasing the items.

Users are required to complete the customer registration process and install the smart store mobile application on their mobile device prior to shopping at the smart store. Customers need to scan their mobile device (or alternatively, a pre-registered credit card) at the entry gate. Cameras placed at the entry location take images of customers, and the system associates those images with respective customers. These images are used by the system later to recognize a customer inside the store. There are various sensors attached to the items' shelves to identify items. Cameras installed inside

the store take images of customers while they are shopping and associate each customer with the items they took from the shelves. Customers scan their mobile devices or credit cards at the exit gate while leaving the store. Payment service deducts the bill, and then the system opens the exit gate. Customers can set up their preferred payment service beforehand and allow the system to process the payment automatically on exit. There are staff on premises for maintenance, assistance, and restocking purposes.

A set of informal hardware and software requirements (not included here due to space reasons) for the smart store system is initially derived based on the problem statement and domain knowledge. We then used the UCM4IoT approach to elicit and specify the system requirements.

6.2. Requirements Development with UCM4IoT

The use case model for the smart store system has been developed using the guidelines elaborated in Section 4.

6.2.1. Actors and Goals

The smart store system has two primary actors, Customer and Staff, both of type Human User (`Human::Customer` and `Human::Staff`). The Customer's goal with the system is `Shopping` for items, and the Staff's is `Maintain Store`.

The system also interacts with external software applications to help it achieve its goals, such as `Software::PaymentService` for processing payments and `Software::FireDetectionSystem` for detecting fire hazards.

For the customer's primary goal to have a successful outcome, a customer has to enter the store, purchase or return items, and then leave the store. Customers may be unable to shop in case of a fire or other hazards (e.g., criminal attack), leading to a failed outcome. Participating actors have been categorized as per the UCM4IoT actor types. Figure 7 presents the summary level use case of the smart store system modelled with the UCM4IoT environment. As depicted in this use case, the primary actor, customer, has three subgoals: `Enter Store`, `Add To Cart`, and `Exit Store`. The `Enter Store` use case requires other actors to participate in fulfilling the customer's (`Human::Customer`) goal: `Sensor::Camera` and `PhysicalEntity::EntryGate` as secondary actors; and `PhysicalEntity::MobileDevice` and `PhysicalEntity::CrediCard` as facilitator actors. This use case invokes `Scan Mobile Device on Entry`, a sub-function level use case that includes the interaction steps associated with a customer scanning the mobile device or

```
use case: Shopping
scope: SmartStoreSystem
level: USER_GOAL
intention: "The customer intends to shop for items in
the smart store."
multiplicity: "Multiple customers can shop at the same
 time."
primary actor: HUMAN::Customer::1..*
main success scenario:
    1. [EnterStore] "The customer enters the store"
    "Step 2 can be repeated according to the intent of
    the Customer."
    2. [AddToCart] "The customer adds an item to the cart."
    3. [ExitStore] "The customer exits the store."
    use case ends in: SUCCESS
extensions:
    exception for (1-3):
        (1-3)a. {ENVIRONMENT_EXCEPTION::FireHazard} #Fire
         hazard in the store. Customer needs to leave the stor
         immediately, and no new customer can enter.#
         use case ends in: FAILURE
    exception for (1-3):
        (1-3)b. {ENVIRONMENT_EXCEPTION::CriminalAttack} #Store
         under attack. No new customer can enter.#
         use case ends in: FAILURE
    alternative for 2:
        2a. [ReturnItem] "Customer returns an item."
        use case continues at step: 3
    alternative for 2:
        "Customer does not add anything to the cart."
        use case continues at step: 3
```

Figure 7: Smart store: Shopping use case.

```
use case: ExitStore
scope: SmartStoreSystem
level: SUB_FUNCTION
intention: "The customer intends to exit the smart store."
multiplicity: "Only one customer can exit through an exit gate
              at a time."
primary actor: HUMAN::Customer::1..*
secondary actor: PHYSICAL_ENTITY::ExitGate::1..1
main success scenario:
    1. [ScanMobileDeviceOnExit] "Customer scans mobile device
        or credit card at the exit gate."
    2. "System instructs the ExitGate to open."
    "Customer passes through the ExitGate."
    3. "System checks out the Customer."
    4. "System closes the ExitGate after 10 seconds"
    use case ends in: SUCCESS
extensions:
    exception for 1:
        1a.{ENVIRONMENT_EXCEPTION::FireHazard}
          #Customer does not need to scan mobile device.#
            use case continues at step: 2
    exception for 1:
        1b. {ENVIRONMENT_EXCEPTION::CriminalAttack} #Customer does not
            need to scan mobile device.#
            use case continues at step: 2
    exception for 2:
        2a. {HARDWARE_EXCEPTION::ExitFailure} #Exit gate failed
            to open. Customer cannot exit.#
        use case continues at step: 1
    alternative for 4:
        "Customer cannot exit the store before the ExitGate is closed."
        use case continues at step: 1
```

Figure 8: Smart store: Exit store use case.

a credit card at the entry gate. The entry gate (`PhysicalEntity::EntryGate`) and the exit gate (`PhysicalEntity::ExitGate`) are built with readers and actuators. System reads user's device or credit card with the reader attached to these gates, and the actuator opens and closes the gates based on the system's instruction. If the store's maximum allowed capacity is reached (for instance, during a pandemic), entry is deactivated. A customer may change his mind while entering the store, and leave the store just after scanning his device at the `PhysicalEntity::EntryGate`, leading the use case to end in an *abandoned* outcome. `Add To Cart` use case starts with the customer picking an item from the shelf and ends in success after the system identifies the item (`Identify Item`) and associates the customer via image recognition (`Recognize User`).

`Identify Item` and `Recognize User` are two core use case scenarios that make the store a "smart store" by automatically detecting who (customer) took what (item). The secondary actors, `Sensor::PressureSensor`, `Sensor::WeightSensor`, and `Reader::TagReader`, participate in `Identify Item` to fulfill the system's goal of identifying any item on the shelves. System uses images received from `Sensor::Camera` to `Recognize User`. The interactions part of the exit scenario at the end of a shopping trip is described in the use case `Exit Store` (see Fig. 8). The `Human::Customer` scans his/her mobile device (or credit card) at the gate invoking the interactions part of

Scan Mobile Device on Exit (see Fig. 9). The system processes the payment via an external Software::Payment Service at this time. Due to space constraints, only a subset of the use cases are discussed and presented here.

```
use case: ScanMobileDeviceOnExit
scope: SmartStoreSystem
level: SUB_FUNCTION
intention:"System intends to read mobile device at the
          exit gate."
multiplicity: "Only one customer can scan the mobile
              device at one exit gate at a time."
primary actor: None
secondary actor:SOFTWARE::PaymentService::1..*,
                HUMAN::Customer::1..1
facilitator actor: PHYSICAL_ENTITY::MobileDevice::1..1,
                   PHYSICAL_ENTITY::CreditCard::1..1
main success scenario:
    1. "System reads MobileDevice at the exit gate."
    2. "System asks Customer to confirm completion of
       shopping."
    3. "System receives confirmation from the Customer."
    4. [PayBill]  "System connects to a PaymentService
       to process the payment."
    5. "System checks out the Customer from the store."
    use case ends in: SUCCESS
 extensions:
    alternative for 1:
        1a. "System reads CreditCard at the exit gate."
        1a.1."System receives balance deduction confirmati
             from the PaymentService."
        1a.2. "System checks out the Customer from the sto
        use case ends in: DEGRADED
    exception for 2:
        2a. {NETWORK_EXCEPTION::CustomerUnreachable} #Syst
            communicate with the customer due to network
            connectivity issue.#
        use case continues at step: 3
```

Figure 9: Smart store: Scan mobile device on exit use case.

```
handler use case: HandleFireHazard
scope: SmartStoreSystem
level: SUB_FUNCTION
intention: "The system locks all entry gates and unlocks all exit gates
           for the customers to ensure safety from fire hazard."
multiplicity: "System locks multiple entry gates and unlocks multiple exit
              gates at once."
primary actor: N/A
secondary actor: SOFTWARE::FireDetectionSystem::1..1,HUMAN::Staff::1..*
facilitator actor: N/A
contexts and exceptions: Shopping {ENVIRONMENT_EXCEPTION::FireHazard},
                         MaintainStore{ENVIRONMENT_EXCEPTION::FireHazard},
                             ExitStore {ENVIRONMENT_EXCEPTION::FireHazard},
                               CheckOut{ENVIRONMENT_EXCEPTION::FireHazard}
main success scenario:
    mode switch: FireEmergency
    "The FireDetectionSystem detects fire in the store."
    1. "System receives an alert from the FireDetectionSystem."
    "Step 2 will be repeated for all EntryGates in the store."
    2. [RestrictEntry] "System locks EntryGate."
    "Step 3 will be repeated for all ExitGates in the store."
    3. "System unlocks ExitGate."
    "Step 4 will be repeated for all EmergencyExits."
    4. "System opens EmergencyExit."
    "Step 5 will be repeated for all Customers and Staff in the store."
    5. "System sends alert to the Customer and Staff inside the store
       to leave store."
    "The fire hazard situation is being resolved."
    6. "System receives a resume message from the FireDetectionSystem."
    "Step 7 will be repeated for all EntryGates in the store."
    7. "System unlocks EntryGate."
    "Step 8 will be repeated for all ExitGates in the store."
    8. "System locks ExitGate."
    "Step 9 will be repeated for all EmergencyExits."
    9. "System closes the emergency exit."
    mode switch: Normal
    use case ends in: SUCCESS
```

Figure 10: Smart store: Send fire alert handler.

While developing the use case scenarios for the smart store system with UCM4IoT, we have discovered several hardware, software, network, and environment exceptions.

6.2.2. Exceptions and Handlers

Environment Exception: We identified environment exceptions by considering possible abnormal scenarios occurring in the environment in which the smart store is operating that may prevent a customer or staff from fulfilling their primary goal.

In the event of a fire hazard, the system receives input from the fire detection system (Software::FireDetectionSystem), an external system used to detect fire, and raises an exception (EnvironmentException::FireHazard, see Fig. 7). In this exceptional situation, no new customers are allowed to enter the store, and all remaining customers and staff need to leave the store

immediately. To handle this exception, the system will temporarily lock the entry (`PhysicalEntity::EntryGate`) for new customers while the fire fighters try to mitigate the fire hazard.

The adaptive behaviour of the system in such a scenario is defined in the `Handle Fire Hazard` handler use case (see Fig. 10). Another environmental exception identified is any kind of criminal attack, such as robbery or a terrorist attack (`EnvironmentException::CriminalAttack`). The handling mechanism, `Alert On Attack`, is triggered when a customer or staff observes such a situation and hits one of the emergency switches (`Device::AttackAlertSwitch`) installed in the store. System notifies the nearest police station, locks the entry gates, and opens the emergency exits.

Hardware Exception: With the numerous devices that are part of a smart store, hardware-related exceptions are very common for this system. As `Sensor::PressureSensor`, `Sensor::WeightSensor`, and `Reader::TagReader` send item data to the system at a time, if the system detects any of these three inputs are missing, it raises one or more of the following exceptions: `HardwareException::PressureUndetected`, `HardwareException::WeightUndetected`, and `HardwareException::TagUnavailable`). If the exit or entry gates stop working due to a hardware failure, `HardwareException::EntryFailure` and `HardwareException::ExitFailure` are identified. These exceptions require repair or replacement of the damaged device. In this case, a service person (`Human::ServicePerson`), is an exceptional actor (of type human user) that the system needs to handle such exceptional scenarios. The interactions of this exceptional actor with the system are detailed in the handler use cases, `Service Gate` and `Service Sensor`. The system will request the customer to use another entry gate (during check-in) or exit gate (during check-out). For item-related hardware exceptions, when the system does not receive a response from the sensor or tag reader, it sends a request along with the device location to the staff to manually scan items for the customers.

Cameras installed at the entry location may be unable to capture the necessary features of a customer (`HardwareException::CameraFailure`), causing the system to not recognize that customer. System requests customer to wait while cameras attempt to capture images (described in the `Request Customer` handler use case).

Network Exception: IoT systems are typically connected with various external networks. Although in the requirements elicitation phase we are assuming reliable communication between components of the smart store system under development, the system needs to communicate with exter-

nal systems through a network (e.g. a customer pays via external payment services, system sends cart updates to a customer's mobile device). If the customer does not have network connectivity, the system will be unable to communicate with the customer to get invoice confirmation, and hence will be unable to process payment on exit. In such a scenario, a network exception, `NetworkException::CustomerUnreachable`, should be raised. This is handled by the system sending request to the customer to confirm payment every five seconds and waiting for the response. When the customer gets connected, he/she receives the request from the system, confirms bill payment, and exits the store.

Software Exception: The smart store system needs to interact with several external software applications (e.g. payment service, fire detection system, etc.) to provide necessary system functionality. If any payment service is down, the system will not be able to process payment from customers, which ultimately disallows customers to exit the store with the items. System checks payment status at the exit gate when someone scans the mobile device, and payment failure will cause the system to not instruct the exit gate to open by default (`SoftwareException::PaymentServiceDown`). To handle this scenario, a special system feature is required such that the system updates the payment status to *on hold* and lets the customer leave the smart store. The handler `Hold Payment` describes the system's adaptive behaviour.

`SoftwareException::PhotoUnmatched` is another example of a software exception raised in the `Recognize User` use case.

Figure 11 presents the use case diagram of the smart store system. To model the structural view of the system, an IoT ARM domain model was also developed. The UCM4IoT model in conjunction with this domain model form the basis for the subsequent analysis and design.

6.2.3. Services and Modes

Services are defined after completing smart store system's requirements elicitation with UCM4IoT. Services were identified to meet the requirements of this system. The services with the associated goals are listed below.

1. Cart processor: `Add To Cart, Return Item, Remove Item`
2. Bill payer: `Pay Bill, Hold Payment`
3. User recognizer: `Recognize Customer, Recognize Staff, Check In, Request User, Request Camera`
4. Entry operator: `Scan Mobile Device On Entry, Check In, Service Gate`
5. Exit operator: `Scan Mobile Device On Exit, Check Out, Service Gate, Get Response`
6. Entry restriction manager:`Restrict Entry`
7. Access manager: `Staff, Staff Login, Customer Login, Register Customer`
8. Inventory manager: `Restock Shelves`
9. Police station notifier: `Alert On Attack`

25

Figure 11: Smart store system: Extended use case diagram.

10. Fire hazard manager: `Handle Fire Hazard`

Cart processor uses *User recognizer* and *Item identifier* services to process customers' cart. It matches customer with their picked or returned items and updates cart accordingly. It then sends update to the *Inventory manager* service. *Bill payer* service receives updated cart information from the *Cart processor* and connects to the payment services to process the payment.

Item identifier service identifies items in the shelf with sensor data. *User recognizer* service recognize customers and staff with the information sent by cameras. *Entry operator* service controls all entry, and *Exit operator* controls exit gates. *Access manager* service manages access to the smart store system.

Fire hazard manager service operates in case of a fire detected by an external `FireDetectionSystem`. `HandleFireHazard` handler invokes a change in behaviour of the *Bill payer* service. *Police station notifier* service triggered by actor `AttackAlertSwitch` communicates with the nearest police station. `AlertOnAttack` handler changes the behaviour of the *Cart processor* and *Exit operator* services. In case of a fire hazard or criminal attack, all exit gates unlock automatically and remain open, temporarily suspending the requirement for scanning customer devices on exit. All emergency gates also remain open in these two emergency cases. However, entry gates stay locked (*Entry restriction manager* service) in these situations to restrict any new customer to enter the store.

26

```
+----------------------+------------------------------------+
| Use Case             | Mode Switch                        |
+----------------------+------------------------------------+
|RestrictEntry         |Normal --> RestrictedEntry          |
|RestrictEntry         |RestrictedEntry --> Normal          |
+----------------------+------------------------------------+
|HandleFireHazard      |Normal--> FireEmergency             |
|HandleFireHazard      |FireEmergency --> Normal            |
|AlertOnAttack         |Normal--> ExternalAttackEmergency   |
|AlertOnAttack         |ExternalAttackEmergency --> Normal  |
+----------------------+------------------------------------+
```

Figure 12: Smart store system: Generated mode switch table.

Table 1: Smart Store System: Mode Summary Table

Modes	Type	Available Services
Normal	Normal	All services excluding *Entry Restriction Manager*, *Fire Hazard Manager*, and *Police Station Notifier*
Restricted Entry	Restricted	*Entry Restriction Manager* and all other services excluding *Entry operator*, *Fire Hazard Manager* and *Police Station Notifier*
Fire Emergency	Emergency	*Entry Restriction Manager*, *Fire Hazard Manager*
External Attack Emergency	Emergency	*Entry Restriction Manager*, *Police Station Notifier*

Different services are activated in different modes. The following section discusses smart store system's operational modes and the services.

Normal Mode: All services operate in `Normal` mode except `Entry restriction manager`, `Fire hazard manager`, and `Police station notifier`. Smart store system has only one `Normal` mode for regular operations. A mode switch from `Normal` to some other mode occurs in an alternate step (e.g., switching from `Normal` to `RestrictedEntry)`), and in other exceptional situations (`EnvironmentException::FireHazard`). All mode switches of the smart store system is shown in Fig. 12.

Emergency Mode: There are two emergency modes in the smart store system. The environment exceptions (`EnvironmentException::FireHazard` and `EnvironmentException::CriminalAttack`) impact the services offered by the system, hence triggering a mode change to the `Fire Emergency` and `External Attack Emergency` mode respectively (see Fig. 12). Handling mechanisms are invoked to recover from the emergency situation (`Handler Fire Hazard`, `Alert On Attack`). On recovery, the system resumes operation in the `Normal` mode.

Restricted Mode: In a situation where customer entry needs to be restricted, the system can still offer the checkout service but no new customer can be allowed into the store. In such a case, the system moves to a `Restricted Entry` mode offering restricted services. This situation may arise when there is the store needs to operate with a limited capacity, such

as in a pandemic. Table 1 shows the services available in a particular mode for smart store system.

6.3. Exception and Handler Summary Generation and Analysis

Figure 4 presents the generated exception summary table for the smart store system. Based on this information, it is possible to quickly identify the possible paths (sequence of use cases) that result in an exception being thrown. Modellers can identify exceptions that are raised in many possible sequences. They can then determine how critical that use case, exception, and any related handlers are to the rest of the system.

As an example, the `TagUnavailable`, `PressureSensorUndetected`, and `WeightSensorUnavailable` exceptions can occur in three possible sequences of use cases each: *UseSmartStore→ Shopping → AddToCart → IdentifyItem*, *UseSmartStore→Shopping → AddToCart → RemoveItem → IdentifyItem*, and *UseSmartStore→Shopping → ExitStore → ScanMobileDeviceOnExit → PayBill → RemoveItem → IdentifyItem*. Likewise, the `Service Sensor` handler handles each occurrence of any of those three exceptions. As each exception can be raised in three possible sequences each, the handler can be invoked in a total of nine different paths. Hence, we can deduce the significance of this handler in this system and attach a higher weight or priority to it.

The mode summary table (Table 1) shows that if there is a fire emergency, customers and staff will not be able to avail any other services apart from exiting the store. The generated mode switch information, shown in Figure 12, explicitly shows the use cases in which the mode of operation switches back to normal. Hence, the success of these goals can be inferred to be critical to the normal operation of the system, requiring designers to ensure that the level of quality-of-service provided by this service is sufficient to continue with normal operation.

7. Case Study 2: Smart Fire Alarm System

In this section, UCM4IoT is demonstrated with the smart fire alarm system case study.

7.1. Smart Fire Alarm System Overview

Over the last decade, IoT devices have revolutionized many industries, from retail and healthcare to home appliances. However, most residential

and commercial buildings still use traditional fire detection systems. Smart fire alarm systems are becoming more common as they are safer than conventional systems as they automatically alert the fire department and the user when a fire is detected. A smart fire alarm system is a system that utilizes internet connectivity to automatically alert the fire department and user when a fire is detected. These smart systems are safer than traditional systems as quick notification increases fire response time and reduces deaths.

The system can also communicate with other systems, such as a sprinkler system, and automatically trigger it. Implementing this system requires various sensors such as heat, smoke, and carbon monoxide. It also requires an initial setup. Users are required to input their contact information and the contact information of their local fire department.

If any of the sensors are triggered, the system automatically alerts the user(s) and the fire department. The heat sensor has a connection with the sprinkler system, and if it detects a high level of heat, it will communicate with the sprinkler system and request it to turn on the sprinkler nearest to the detected heat source. After the initial setup, the system does not require any human intervention unless a hardware failure requires maintenance.

Hardware and software requirements for the smart fire alarm system are initially derived based on domain knowledge and informal requirements. Next, the UCM4IoT language is used to specify the system requirements and further elicit any applicable hardware, software, network, and environment exceptions.

7.2. Requirements Development with UCM4IoT

7.2.1. Actors and Goals

The Smart Fire Alarm System has four primary actors, User (`Human::User`), HeatSensor (`SENSOR::HeatSensor`), SmokeSensor (`SENSOR::SmokeSensor`) and CarbonMonoxideSensor (`SENSOR::CarbonMonoxideSensor`).

One of the summary level use cases for the smart fire alarm system that was written using the UCMIoT tool is presented in Fig. 13. The primary actor, the user, has three sub-goals: System Maintenance, Test Components, and Turn Off Alarm. The System Maintenance use case does not require any other actor and does not need to invoke other use cases to fulfill the user goal. In this scenario, the user first authenticates with the system by providing a password and then updates the system settings. Next, the system refreshes the software to synchronize any new changes. If the authentication fails, the use case ends in failure.

The user's goal with the system is to perform system maintenance and turn off the fire alarm. To achieve this goal, the user enters a password to authenticate with the system and then replaces the faulty sensors/battery. The user then requests the system to refresh the software to ensure the new component is synced correctly. The user's second goal is to turn off the alarm; this is done by pressing the turn off button on the alarm, which makes the system request the alarm to be turned off. The user's goal will be successful if they are able to perform system maintenance by replacing a component and also if they are able to turn off the alarm manually. The user's goal will have a failed outcome if the user cannot authenticate with the system or if the turn-off button is not functioning. Participating actors have been categorized as per the UCM4IoT actor types.

The goal of the heat sensor is to sense heat and inform the system so it can trigger the alarm. Fig. 14 illustrates the `Sound Heat Alarm` use case. The sensor will then invoke the fire response use case. Similarly, the smoke sensor's goal is to sense smoke levels, and the carbon monoxide sensor's goal is to sense carbon monoxide levels. The sensor's primary goal will be successful if the alarm is triggered and the fire response use case is invoked. The alarm is triggered when abnormal amounts of heat, smoke, or carbon monoxide are detected. These trigger the `Alert Fire Department` use case. The fire response use case is invoked if heat, smoke or carbon monoxide are sensed for more than 2 minutes (the timeout may vary for different systems). The sensor's goal will have a failed outcome if there is a hardware failure with the sensors.

The secondary actors, `Sensor::HeatSensor`, `Sensor::SmokeSensor`, and `Sensor::CarbonMonoxideSensor`, `PhysicalEntity::Battery`, are required to complete the Test Components goal. The use case starts with the system checking the battery level and then notifying the User if it is below a certain threshold through their mobile device. Next, the system runs diagnostic tests on all the various sensors, and if it finds a failing sensor, it will notify the User through the mobile device. However, if the Diagnostic test shows all the sensors are working correctly, the use case will end in success.

The `Fire Response` use case will invoke all the other required use cases to handle the fire. The first use case invoked is `Notify Sprinkler System`, in which the system communicates with the Sprinkler System to reduce the temperature threshold. The second use case invoked is `Alert Fire Department`, in which the system notifies the fire department about the fire. Then, it will invoke the `Alert User` use case in which the system alerts the User about

the fire. It will also invoke the `Change Display Colour` use case in which the system communicates with the Display to change the colour to red. Finally, it will invoke the `Open Emergency Door` use case in which the system requests the emergency door to unlock.

Numerous software, hardware, network, and environment exceptions were discovered during the creation of use cases for the smart fire alarm system using the UCMIoT language.

Figure 13: Smart fire alarm: Use smart fire alarm system use case.

Figure 14: Smart fire alarm: Sound heat alarm use case.

Figure 15: Smart fire alarm: Alert Fire Department use case.

Figure 16: Smart fire alarm: Reconnect To Network handler use case.

7.2.2. Exceptions and Handlers

Environment Exception: In the event that a power outage occurs, the system will raise an exception `EnvironmentException::PowerOutage`, see Fig. 4). When this exception occurs, the network connectivity will be lost, so the system will be unable to alert the User and fire department if a fire is detected. The system will also not be able to communicate with the sprinkler system. To handle this exception, the system first tries to reconnect to the network a maximum of three times. If the issue is still not resolved, the system will switch to a `NoInternet` mode and alert the User about the network issue by sending an SMS. Once the issue is resolved, it will switch back to the normal mode. The adaptive behaviour of the system in such a scenario is defined in the `ReconnectToNetwork` handler use case. Another environmental exception that might occur is a cyberattack `EnvironmentException::CyberAttack`. The cyberattack can occur at any time, and it will compromise the system by taking down the internet connection. This exception is handled the same way as the `PowerOutage` using the `ReconnetToNetwork` handler. Fig. 16 shows the use case to handle these exceptions.

Hardware Exception: The smart fire alarm system utilizes various sensors and hardware components to operate, and as a result, hardware exceptions can occur frequently. To detect the fire or carbon monoxide, the `Sensor::SmokeSensor`, `Sensor::HeatSensor`, and `Sensor::CarbonMonoxideSensor` are used. If a system test determines that any of these sensors are not working, it will raise one of these exceptions `HardwareException::SmokeSensorFailure`, `HardwareException::HeatSensorFailure`, or `HardwareException::CarbonMonoxideSensorFail` The physical button to turn off the Alarm and reset the sensors might not work, raising the `HardwareException::ButtonFailure`.

One of the essential hardware components is the Alarm; if one or more sensors are triggered but the Alarm is not sounding, then the `HardwareException::AlarmFailure` is called. The Display might have a hardware failure `HardwareException::DisplayColorChange` or `HardwareException::DisplayColorRevert`, resulting in disabled individuals not being alerted about the fire. If the automatic emergency door does not work, `HardwareException::DoorNotWorking` is raised. All of these hardware exceptions require repairing the damaged sensor or physical entity. The system will need the User `Human::User` to handle such exceptional scenarios. The interaction steps of this exceptional actor with the system are defined in the System Maintenance handler use cases.

Network Exception: If the system gets disconnected from the network, a network exception, `NetworkException::NoInternet`, will be raised. This exception is handled by the `ReconnectToNetwork` handler, where the system first tries to reconnect to the network three times. If it still cannot connect, the system will inform the User of the network issue by sending an SMS. After the User resolves the issue, they will press the reset button to let the system know. Then the system will try to reconnect; if it still does not work, the system will keep looping the above steps. Network exceptions can also be raised if the network issue comes from the external system. For example, if the sprinkler system has connection issues, the system will raise a `NetworkException::SprinklerSystemUnavailable`. Similarly, if the Central Monitoring Station or the User does not have a network connection, the following exceptions will be raised: `NetworkException::CenteralMonitoringStationUnavailabl` `NetworkException::UserUnavailable`. All three of these exceptions are also handled by the `ReconnectToNetwork handler` in a similar manner.

Software Exception: The smart fire alarm system communicates with various external software systems to provide users with essential functionality. For example, the system needs to communicate with the Central Monitoring Station software system to provide the `AlertFireDeparment` functionality. If this system is down, the `SoftwareException::CenteralMonitoringStationUnavailable` exception will be raised. Another example of a software exception is `SoftwareException::Sprinkl` where the system cannot provide the sprinkler notification functionality and send messages to the sprinkler system.

The use case diagram showcasing the actors, use cases, exceptions, and handlers derived from the UCM4IoT model is presented in Fig. 17. The UCM4IoT model forms the basis for the subsequent analysis.

7.2.3. Services and Modes

We discovered the following services are required to satisfy the goals of the smart fire alarm system.

Heat detector, *Smoke detector*, and *Carbon-monoxide detector* services detect and sound alarm for heat, smoke, and carbon-monoxide respectively. These services invoke the *Notifier* service to notify sprinkler system and alert fire department in case of a fire hazard. *Notifier* service also alerts and warns the system user.

Commercial users of this smart fire alarm system use *Display colour changer* service along with other services where the system display colour changed from green to red in case the system detects fire, smoke, or carbon-
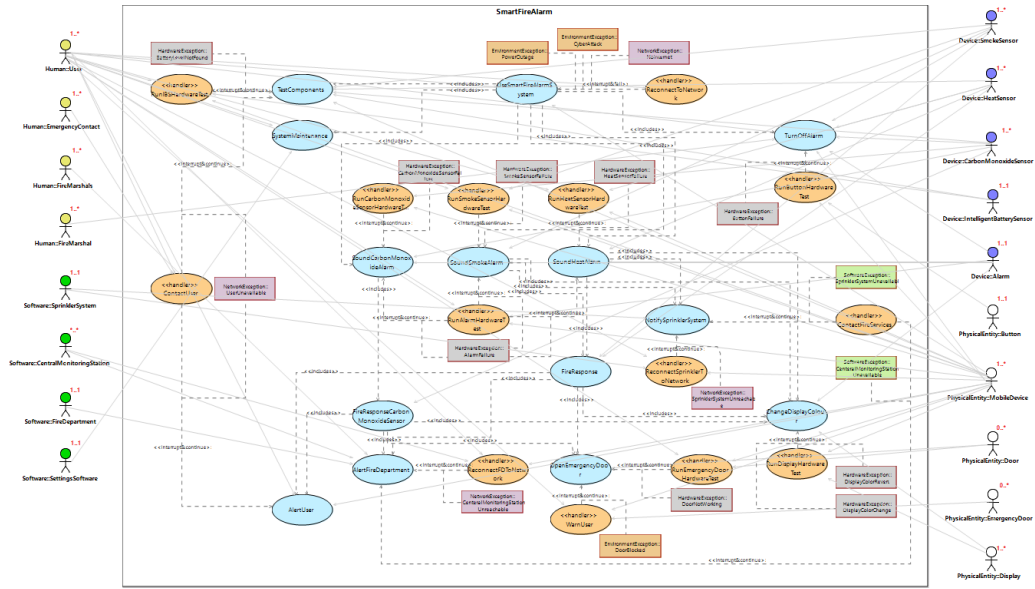
33

Figure 17: Smart fire alarm system: Use case diagram.

monoxide. The system needs regular maintenance, that is performed by the *System maintainer* service.

System needs to test its hardware components regularly to ensure maximum safety. *Component tester* service tests all hardware components of the system. In case of exceptions, system also need to run tests on these components. Thus, all the hardware and network exceptions are handled by this service. Users of this system can manually turn off the alarm by using the *Users control* service. *Emergency doors control* service controls the doors opening after receiving notification from the *Notifier* service. The following list shows services and the associated goals.

1. Heat detector: `Sound Heat Alarm`
2. Smoke detector: `Sound Smoke Alarm`
3. Carbon-monoxide detector: `Sound Carbon Monoxide Alarm`
4. Notifier: `Notify Sprinkler System, Alert Fire Department, Alert User, Reconnect To Network, Warn User`
5. Display colour changer: `Change Display Colour`
6. System maintainer: `System Maintenance`
7. Component tester: `Test Components, Run Smoke Sensor Hardware Test, Run Heat Sensor Hardware Test, Run Alarm Hardware Test, Run Emergency Door Hardware Test, Run Carbon Monoxide Sensor Hardware Test, Run Button Hardware Test, Run Display Hardware Test, Contact User, Reconnect Sprinkler to Network, Reconnect FD to Network, Run IBS Hardware Test`
8. Users control: `Turn Off Alarm`
9. Emergency doors control: `Open Emergency Door`

The smart fire alarm system has two operational modes: one normal mode for regular operations and two restricted modes.

Normal Mode: All services are available in the `Normal` mode except when there is an issue with the network that does not let the system reach the central monitoring system in case of power outage, cyberattack, and lack of internet connection.

Degraded Mode: When system cannot connect to the central station due to network exception, it switches to a degraded mode, `No Alert`. The system goes to the `No Internet` mode in case of `ENVIRONMENT_EXCEPTION::PowerOutage`, `ENVIRONMENT_EXCEPTION::CyberAttack`, and `NETWORK_EXCEPTION::NoInternet`. In `No Internet` mode, all services except *Component tester* and *System maintainer* service will remain operational.

Table 2: Smart fire alarm system: Mode summary table.

Modes	Type	All Services
Normal	Normal	All Services
No Alert	Degraded	All services excluding *Contact user*
No Internet	Degraded	All services excluding *Component tester* and *System maintainer*

7.3. Exception and Handler Summary Generation and Analysis

Figure 18 presents the generated exception summary table and Fig. 19 presents a slice of the generated handler summary table for the smart fire alarm system. Based on the paths generated in the summary information, we can determine the criticality of the use cases, exceptions and handlers.

As an example, the exception summary table can help in deducing that the `Change Display Color` use case has a high significance since it is included in 10 possible sequences and may raise hardware exceptions. Similarly `AlertUser` also appears in 4 possible sequences and may raise network exceptions. These use cases and associated handlers should be assigned a higher priority and/or dependability level.

By analyzing the `ReconnetToNetwork` handler, it can be seen that it is a crucial handler as it handles the `PowerOutage`, `CyberAttack`, and `NoInternet` exceptions (as seen in Fig. 19). All three of these exceptions need to be addressed immediately, since a lack of network connectivity hampers critical system functionality.

35

Exception Type	Exception Name	Exception Source	Associated Actors	Handler	Interrupt Type	Path
ENVIRONMENT_EXCEPTION	PowerOutage	UseSmartFireAlarmSystem	HUMAN::User PHYSICAL_ENTITY::MobileDevice	{ReconnectToNetwork}	Interrupt and Fail	UseSmartFireAlarmSystem >> {ReconnectToNetwork} !I&F GLOBAL::SystemMaintenance >> {ReconnectToNetwork} !I&F GLOBAL::TestComponents >> {ReconnectToNetwork} !I&F GLOBAL::TurnOffAlarm >> {ReconnectToNetwork} !I&F GLOBAL::SoundHeatAlarm >> {ReconnectToNetwork} !I&F GLOBAL::NotifySprinklerSystem >> {ReconnectToNetwork} !I&F GLOBAL::ChangeDisplayColour >> {ReconnectToNetwork} !I&F GLOBAL::FireResponse >> {ReconnectToNetwork} !I&F GLOBAL::AlertFireDepartment >> {ReconnectToNetwork} !I&F GLOBAL::AlertUser >> {ReconnectToNetwork} !I&F GLOBAL::OpenEmergencyDoor >> {ReconnectToNetwork} !I&F GLOBAL::SoundSmokeAlarm >> {ReconnectToNetwork} !I&F GLOBAL::SoundCarbonMonoxideAlarm >> {ReconnectToNetwork} !I&F GLOBAL::FireResponseCarbonMonoxideSensor >> {ReconnectToNetwork} !I&F
	CyberAttack	UseSmartFireAlarmSystem	HUMAN::User PHYSICAL_ENTITY::MobileDevice	{ReconnectToNetwork}	Interrupt and Fail	UseSmartFireAlarmSystem >> {ReconnectToNetwork} !I&F GLOBAL::SystemMaintenance >> {ReconnectToNetwork} !I&F GLOBAL::TestComponents >> {ReconnectToNetwork} !I&F GLOBAL::TurnOffAlarm >> {ReconnectToNetwork} !I&F GLOBAL::SoundHeatAlarm >> {ReconnectToNetwork} !I&F GLOBAL::NotifySprinklerSystem >> {ReconnectToNetwork} !I&F GLOBAL::ChangeDisplayColour >> {ReconnectToNetwork} !I&F GLOBAL::FireResponse >> {ReconnectToNetwork} !I&F GLOBAL::AlertFireDepartment >> {ReconnectToNetwork} !I&F GLOBAL::AlertUser >> {ReconnectToNetwork} !I&F GLOBAL::OpenEmergencyDoor >> {ReconnectToNetwork} !I&F GLOBAL::SoundSmokeAlarm >> {ReconnectToNetwork} !I&F GLOBAL::SoundCarbonMonoxideAlarm >> {ReconnectToNetwork} !I&F GLOBAL::FireResponseCarbonMonoxideSensor >> {ReconnectToNetwork} !I&F
	DoorBlocked	OpenEmergencyDoor	PHYSICAL_ENTITY::Door	{WarnUser}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> FireResponse >> OpenEmergencyDoor >> {WarnUser} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> FireResponse >> OpenEmergencyDoor >> {WarnUser} !I&C UseSmartFireAlarmSystem >> SoundCarbonMonoxideAlarm >> FireResponseCarbonMonoxideSensor >> OpenEmergencyDoor >> {WarnUser} !I&C
HARDWARE_EXCEPTION	SmokeSensorFailure	SoundSmokeAlarm	DEVICE::SmokeSensor	{RunSmokeSensorHardwareTest}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundSmokeAlarm >> {RunSmokeSensorHardwareTest} !I&C
	HeatSensorFailure	SoundHeatAlarm	DEVICE::HeatSensor	{RunHeatSensorHardwareTest}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> {RunHeatSensorHardwareTest} !I&C
	CarbonMonoxideSensorFailure	SoundCarbonMonoxideAlarm	DEVICE::CarbonMonoxideSensor	{RunCarbonMonoxideSensorHardwareTest}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundCarbonMonoxideAlarm >> {RunCarbonMonoxideSensorHardwareTest} !I&C
	AlarmFailure	SoundHeatAlarm SoundSmokeAlarm SoundCarbonMonoxideAlarm	DEVICE::Alarm	{RunAlarmHardwareTest}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> {RunAlarmHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> {RunAlarmHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundCarbonMonoxideAlarm >> {RunAlarmHardwareTest} !I&C
	ButtonFailure	TurnOffAlarm	HUMAN::User DEVICE::Alarm HUMAN::FireMarshal	{RunButtonHardwareTest}	Interrupt and Continue	UseSmartFireAlarmSystem >> TurnOffAlarm >> {RunButtonHardwareTest} !I&C
	DoorNotWorking	OpenEmergencyDoor	PHYSICAL_ENTITY::Door	{RunEmergencyDoorHardwareTest}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> FireResponse >> OpenEmergencyDoor >> {RunEmergencyDoorHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> FireResponse >> OpenEmergencyDoor >> {RunEmergencyDoorHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundCarbonMonoxideAlarm >> FireResponseCarbonMonoxideSensor >> OpenEmergencyDoor >> {RunEmergencyDoorHardwareTest} !I&C
	DisplayColorChange	ChangeDisplayColour	PHYSICAL_ENTITY::Display	{RunDisplayHardwareTest}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundHeatAlarm >> FireResponse >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> FireResponse >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundCarbonMonoxideAlarm >> FireResponseCarbonMonoxideSensor >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C
	DisplayColorRevert	ChangeDisplayColour	PHYSICAL_ENTITY::Display	{RunDisplayHardwareTest}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundHeatAlarm >> FireResponse >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> FireResponse >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C UseSmartFireAlarmSystem >> SoundCarbonMonoxideAlarm >> FireResponseCarbonMonoxideSensor >> ChangeDisplayColour >> {RunDisplayHardwareTest} !I&C
	BatteryLevelNotFound	TestComponents	PHYSICAL_ENTITY::MobileDevice	{RunIBSHardwareTest}	Interrupt and Continue	UseSmartFireAlarmSystem >> TestComponents >> {RunIBSHardwareTest} !I&C
NETWORK_EXCEPTION	NoInternet	UseSmartFireAlarmSystem	HUMAN::User PHYSICAL_ENTITY::MobileDevice	{ReconnectToNetwork}	Interrupt and Fail	UseSmartFireAlarmSystem >> {ReconnectToNetwork} !I&F
	UserUnavailable	AlertUser TestComponents	HUMAN::User PHYSICAL_ENTITY::MobileDevice	{ContactUser}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> FireResponse >> AlertUser >> {ContactUser} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> FireResponse >> AlertUser >> {ContactUser} !I&C UseSmartFireAlarmSystem >> SoundCarbonMonoxideAlarm >> FireResponseCarbonMonoxideSensor >> AlertUser >> {ContactUser} !I&C UseSmartFireAlarmSystem >> TestComponents >> {ContactUser} !I&C
	SprinklerSystemUnreachable	NotifySprinklerSystem	SOFTWARE::SprinklerSystem	{ReconnectSprinklerToNetwork}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> NotifySprinklerSystem >> {ReconnectSprinklerToNetwork} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> NotifySprinklerSystem >> {ReconnectSprinklerToNetwork} !I&C
	CenteralMonitoringStationUnreachable	AlertFireDepartment	SOFTWARE::CentralMonitoringStation	{ReconnectFDToNetwork}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> FireResponse >> AlertFireDepartment >> {ReconnectFDToNetwork} !I&C UseSmartFireAlarmSystem >> SoundCarbonMonoxideAlarm >> FireResponseCarbonMonoxideSensor >> AlertFireDepartment >> {ReconnectFDToNetwork} !I&C
SOFTWARE_EXCEPTION	CenteralMonitoringStationUnavailable	AlertFireDepartment		{Rename}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> FireResponse >> AlertFireDepartment >> {Rename} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> FireResponse >> AlertFireDepartment >> {Rename} !I&C UseSmartFireAlarmSystem >> SoundCarbonMonoxideAlarm >> FireResponseCarbonMonoxideSensor >> AlertFireDepartment >> {Rename} !I&C
	SprinklerSystemUnavailable	NotifySprinklerSystem	SOFTWARE::SprinklerSystem	{Rename}	Interrupt and Continue	UseSmartFireAlarmSystem >> SoundHeatAlarm >> NotifySprinklerSystem >> {Rename} !I&C UseSmartFireAlarmSystem >> SoundSmokeAlarm >> NotifySprinklerSystem >> {Rename} !I&C

Figure 18: Smart fire alarm system: Generated exception summary.

```
┌──────────────────────────┬──────────────────────────┬──────────────────────────────────────┬──────────────────────────────────────┐
│ Handlers                 │ Dependent Use Cases      │ Handled Exceptions                   │ Actors                               │
├──────────────────────────┼──────────────────────────┼──────────────────────────────────────┼──────────────────────────────────────┤
│ ReconnectToNetwork       │ UseSmartFireAlarmSystem  │ ENVIRONMENT_EXCEPTION::PowerOutage    │ HUMAN::User                          │
│                          │                          │ ENVIRONMENT_EXCEPTION::CyberAttack    │ PHYSICAL_ENTITY::MobileDevice        │
│                          │                          │ NETWORK_EXCEPTION::NoInternet         │                                      │
├──────────────────────────┼──────────────────────────┼──────────────────────────────────────┼──────────────────────────────────────┤
│ WarnUser                 │ OpenEmergencyDoor        │ ENVIRONMENT_EXCEPTION::DoorBlocked    │ PHYSICAL_ENTITY::Door                │
│                          │                          │                                      │ HUMAN::User                          │
│                          │                          │                                      │ PHYSICAL_ENTITY::EmergencyDoor       │
│                          │                          │                                      │ PHYSICAL_ENTITY::MobileDevice        │
├──────────────────────────┼──────────────────────────┼──────────────────────────────────────┼──────────────────────────────────────┤
│ RunSmokeSensorHardwareTest │ SoundSmokeAlarm        │ HARDWARE_EXCEPTION::SmokeSensorFailure │ DEVICE::SmokeSensor                 │
│                          │                          │                                      │ DEVICE::Alarm                        │
│                          │                          │                                      │ HUMAN::User                          │
│                          │                          │                                      │ PHYSICAL_ENTITY::MobileDevice        │
├──────────────────────────┼──────────────────────────┼──────────────────────────────────────┼──────────────────────────────────────┤
│ RunHeatSensorHardwareTest │ SoundHeatAlarm          │ HARDWARE_EXCEPTION::HeatSensorFailure  │ DEVICE::HeatSensor                  │
│                          │                          │                                      │ DEVICE::Alarm                        │
│                          │                          │                                      │ HUMAN::User                          │
│                          │                          │                                      │ PHYSICAL_ENTITY::MobileDevice        │
└──────────────────────────┴──────────────────────────┴──────────────────────────────────────┴──────────────────────────────────────┘
```

Figure 19: Smart fire alarm: Generated handler summary (slice).

Figure 20 shows that the handler use cases `ReconnectedToNetwork` and `ReconnectFDToNetwork` are responsible for returning the system to a normal mode of operation from `NoInternet` and `NoAlert` emergency modes. As we discussed earlier, `ReconnectedToNetwork` is a crucial handler, and this table indicates that successful completion of this handling is required to revert back to normal mode. Thus, this information must be taken into consideration at the design phase. The mode summary table (Table 2) lists the services available in each mode and aids in understanding if the services in a mode satisfy the minimum requirement of the system. For example, user cannot perform a test on the system or perform maintenance in the `NoInternet` mode, which does not hamper the ability of the system to detect fire and sound alarm.

```
+-----------------------+--------------------------------+
| Use Case              | Mode Switch                    |
+-----------------------+--------------------------------+
+-----------------------+--------------------------------+
|ReconnectToNetwork     |Normal --> NoInternet           |
|ReconnectToNetwork     |NoInternet --> Normal           |
|ReconnectFDToNetwork   |Normal --> NoAlert              |
|ReconnectFDToNetwork   |NoAlert --> Normal              |
+-----------------------+--------------------------------+
```

Figure 20: Smart fire alarm system: Generated mode switch table.

8. Comparative Analysis

We used the smart store system and smart fire alarm system case studies to demonstrate our work. To further assess the effectiveness and usefulness of UCM4IoT as a requirements development approach, we carried out an informal comparative evaluation. Specifically, we selected two approaches for this purpose: 1) the standard use case approach [9], because this is the most

commonly used technique for use case-based modelling, and 2) exceptional use cases [4], due to the level of support for exception handling, which exceeds other existing approaches. Use cases for both systems were developed based on the same problem statement using the standard approach, exceptional use cases, and UCM4IoT. The goal was to analyze whether UCM4IoT can assist in discovering more system functionalities and exceptions than existing approaches, hence leading to a more correct and complete specification. We compared the approaches on the basis of four criteria: exceptions identified, core functionality identified, recovery measures specified, and actors identified. Taking these aspects into consideration, we posed the following evaluation questions (EQ) to compare and assess.

- EQ1: Does eliciting requirements with UCM4IoT help in discovering a higher number of (exceptional) situations over exceptional use cases and the standard process, hence leading to a more complete requirements specification?

- EQ2a: Does eliciting requirements with UCM4IoT help in revealing and defining core system functionalities that are not identified with exceptional use cases and the standard approach, hence leading to a more complete requirements specification?

- EQ2b: Does eliciting requirements with UCM4IoT help in revealing and defining system functionalities, in particular, recovery measures, that are not identified with exceptional use cases and the standard approach, hence leading to a more complete requirements specification?

- EQ3: Are we able to discover additional actors that were not identified with standard and exceptional use cases?

Our findings for each of the two case studies are presented here.

8.1. Smart Store Case Study

Since tool support is only available for the UCM4IoT language, we initially documented all three use case models without using the tool.

EQ1: Number of exceptions increased by 83.3% from exceptional use cases to UCM4IoT. We have identified 11 exceptions with UCM4IoT, and 6 exceptions with exceptional use cases. No exception was discovered while developing use cases with the standard process as it does not have the concept

of exceptions, nor does the approach enable discovery of such exceptional situations.

EQ2a: UCM4IoT increased the interaction steps in use case models by 15.9% than exceptional use cases, and 30.8% than the standard approach. In the standard use case models, there were 39 interactions steps, in exceptional use cases 44, and in UCM4IoT 51. We did not consider interactions steps in handlers for this investigation. UCM4IoT helped to identify several core functionalities of the system that did not come up while eliciting requirements with standard and exceptional use cases process. UCM4IoT ensures that all the actors participating in achieving a goal must have interaction steps with the system, and the different types of exceptions also led to specification of more interactions that otherwise would be easily missed.

EQ2b: With UCM4IoT, we defined recovery measures to handle the discovered exceptions. In the process of writing standard use cases, we developed 21 use cases. However, we did not identify functionalities such as handling fire hazard, holding payment in case of payment service issues, requesting camera to take more images if the system is unable to recognize a user, automatically notifying service person in case of a hardware failure, etc. There is a significant increase of 166.7% in defining the recovery measures with UCM4IoT than exceptional use cases. We defined eight handlers with UCM4IoT, whereas there are only three handlers defined with exceptional use cases, and none with standard use cases.

EQ3: New functionalities of the system defined by UCM4IoT introduced new actors. The classification of actors also helped to identify new actors that were not discovered earlier. We identified 30.8% more actors with UCM4IoT than exceptional use cases, and 70% new actors than the standard process. Four new exceptional actors, namely `PhysicalEntity::EmergencyExit`, `PhysicalEntity::Police`, `Device::AttackAlertSwitch`, `Software::FireDetectionSystem` are discovered with UCM4IoT that were not identified with exceptional use cases.

In the standard approach, the smart shelf was added as an actor - no sensors or readers part of the smart shelf were identified as actors in this phase. Subsequently, additional actors constituting the smart shelf (`Device::WeightSensor`, `Device::PressureSensor`, `Device::TagReader`) as well as the `Human::ServicePerson` actor were added to the UCM4IoT use cases.

8.2. Smart Fire Alarm Case Study

EQ1: Applying the UCM4IoT approach on the Smart Fire Alarm System resulted in a 50% improvement in the number of exceptions identified

compared to the exceptional use cases approach. The total number of exceptions found using exceptional use cases was 12, while the total for UCM4IoT was 18. New exceptions were discovered in 6 different use cases. In the `UseSmartFireAlarmSystem` use case, two exceptions were found using exceptional use cases, and three exceptions were found using UCM4IoT. Meanwhile, in the `TestComponents` use case, there were no exceptions found using exceptional use cases, while one exception was discovered using UCM4IOT. In the `Open Emergency Door` use case, one exception was found using exceptional use cases, and two exceptions were found using UCM4IOT. In the `NotifySprinklerSystem` and `AlertFireDepartment` use cases, there was one exception found using exceptional use cases, while two exceptions were discovered using UCM4IOT. Finally, in the `Change Display Color` use case, one exception was found using exceptional use cases, and two exceptions were revealed with the help of UCM4IOT.

EQ2a: Eliciting requirements for Smart Fire Alarm System with UCM4IoT helped in revealing and defining core system functionalities that were not identified with exceptional use cases. Applying UCM4IoT resulted in a 18.2% more interaction steps in the use cases than exceptional use cases and 23.8% with standard process. There were 21 interaction steps for standard, 22 for exceptional use cases while there were 26 interaction steps using UCM4IoT. The increase in interaction steps was caused by the need to accommodate timeouts, new exceptions, missing actors and network communications. In order to handle exceptions, they need to be first detected, so timeouts were added to the UCM4IoT environment. UCM4IoT's categorization of exceptions into Software, Hardware, Environment and Network exceptions helped visualize the system into these 4 components. When reviewing the use cases there were missing network communications that were not considered when developing with exceptional use cases.

EQ2b: There was a 18.2% increase in recovery measures defined as handlers when using the UCM4IoT approach compared to exceptional use cases. Handlers introduced additional functionalities to the system. Two new handlers called `RunIBSHardwareTest` and `RunEmergencyDoorHardwareTest` were created to handle the `BatteryLevelNotFound` and `DoorNotWorking` exceptions, which were discovered using UCM4IoT. The `DisplayColorRevert` and `NoInternet` exceptions were handled using existing handlers.

EQ3: We did not discover any new actors with UCM4IoT. Number of actors for smart fire alarm is same for all three approaches.

8.3. Summary

Figure 21 summarizes our findings for the smart store and smart fire alarm system case studies.
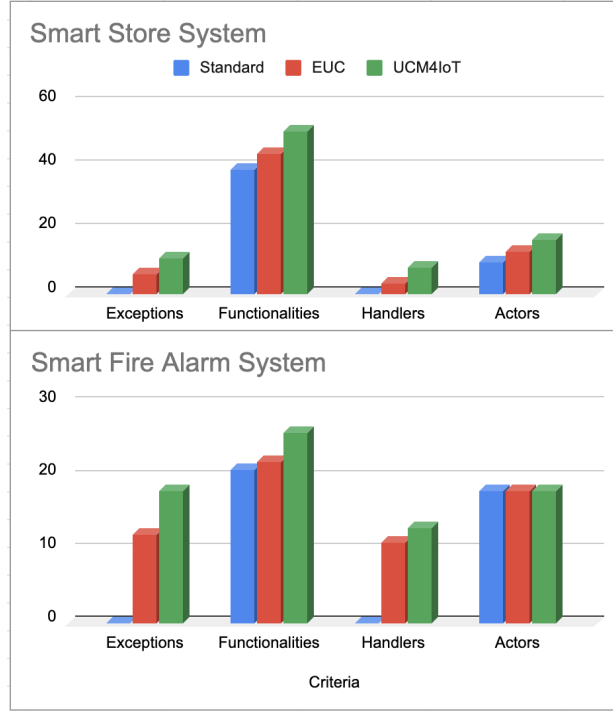

Figure 21: Summary of Findings

Based on both case studies, it is evident that using the domain-specific language, UCM4IoT brought about significant improvements in the requirements specification process of IoT-based systems and led to the development of a more complete and precise requirements model. It allowed an overall increase in the number of exceptional situations discovered and led to identifying missing system functionality (required to handle such situations) as well as core system functionality, thus ensuring that system designers implement mechanisms to handle these exceptions.

9. Discussion

This work is a first step towards developing a domain-specific requirements development language for IoT. Having IoT-specific language constructs

enables modellers to take into consideration the different facets of the domain from the early stages of development.

UCM4IoT has been applied on several case studies of non-trivial nature selected from different IoT application domains: smart store system, smart fire alarm system, and smart lights system. The latter is not covered in this paper. As illustrated in Section 8, there is evidence that UCM4IoT is indeed useful and effective in establishing a more complete and correct requirements model for IoT systems. As the next step in this direction, we intend to apply UCM4IoT on industrial case studies.

The UCM4IoT models can benefit designers in the later phases of development. Identification of hardware exceptions in critical goals could be an indication to have fault tolerance mechanisms (such as hardware redundancy) built into the system at the design phase. Software exceptions can be taken as an indication for incorporating fault prevention mechanisms or additional system functionality by refining the initial specification for critical system operations (for instance, `SoftwareException::PaymentServiceDown` may lead to a decision further downstream to require customers to provide an alternate payment method). Network exceptions can help in making decisions on the need for offering alternate connectivity options (for instance, `NetworkException::CustomerUnreachable` may be avoided by making a public Wi-Fi network accessible to all customers inside the smart store).

Decisions on invoking adaptive mechanisms may be dependent on non-functional requirements (NFRs) or quality constraints associated with a system. These quality attributes may dictate the kind of handler to be used as well as indicate the criticality level of each user or sub-functional goal.

As future work, we plan on investigating the impact on handlers when taking NFRs into account at the elicitation phase.

A modelling environment offering syntax-directed editing and validation checks is very useful for documenting correct and complete use cases. The generated exception and handler summary tables provide a global view of the use cases allowing useful insights and inferences to be drawn regarding the requirements. Automatically mapping the textual use cases to a use case diagram makes it very easy for modellers to keep the graphical representation consistent with the textual model. Moreover, the UCM4IoT model can be mapped to a (partial) IoT ARM domain model. The use cases can be further used to generate test cases for the system. We intend to extend the UCM4IoT environment in this direction.

10. Related Work

In this section, we discuss existing work on use cases as well as requirements modelling for IoT systems.

10.1. Use Case Modelling

Sindre and Opdahl [20] present a method for eliciting security requirements by extending use cases. Misuse cases only address security risks and and was designed to mitigate potential threats exposed by misusers. Savić et al. [21] propose a DSL, SilabReq, that allows use cases to be defined at different abstraction levels depending on the type of stakeholder.

Cockburn [7] informally addresses the notion of failure scenarios in standard use cases with alternative situations. Amyot [22] proposes use case maps as part of the URN standard for specifying functional requirements and causal relationships between use cases. This approach implicitly addresses the detection of undesirable scenarios.

Mustafiz et al. [16] propose a requirements engineering process, DREP, to guide developers in considering reliability and safety concerns of reactive systems starting at the use case level. Two types of exceptions, context-affecting and service-related, are used to define exceptions and associated dependability handlers. DREP does not come with any tool support for use case modelling.

UCM4IoT is based on the notion of exception handling in use cases proposed in [4] and [16]. UCM4IoT takes this further by proposing IoT-specific concepts in use case modelling along with a dedicated environment for editing, validating and analyzing use cases.

Maleki et al. [23] propose a framework for exception discovery and handling in the elicitation phase to be used for developing test cases. Requirements are defined with graphs in three levels of granularity (goals, scenarios, and sub-functions) along with a fault tree to present probable error or failure. While UCM4IoT can be extended for model-based requirements testing, at this time, it does not cover testing.

While there exists several modelling tools with support for use case diagrams, very few environments are available for textual use case modelling. Visual Use Case (`http://www.visualusecase.com/`) supports standard use cases editing and mapping to an activity diagram. UCEd [24] also provide support for textual use case modelling. However, these tools do not cover exceptions or any IoT-related aspects and are not based on MDE techniques.

The RUCM [25] use case modelling approach supports global and bounded alternatives that could be used to define exceptions, but no explicit language-level support is provided for IoT systems.

To the best of our knowledge, UCM4IoT is the only existing one to provide support for model-driven requirements development with textual use case modelling for IoT systems.

Our environment supports discovery and documentation of UCM4IoT use cases as well as generation of exception summary information that can be used for further analysis of the requirements. While UCM4IOT currently does not provide integrated support for use case diagrams, we use an extended form of use case diagrams for graphically representing and summarizing our use cases.

10.2. Requirements Modelling for IoT

Reggio [26] introduces IoTReq, a goal-oriented method for elicitation and specification of IoT system requirements, which uses UML profiles for domain modelling. Different types of behaviour are specified including unexpected behaviour or constraints (expressed with a sequence or activity diagram) imposed by the system that need to be made impossible by the IoT system. In comparison, exceptions (as used in our work) are situations which hinder fulfillment of goals. Meacham et al. [27] propose a requirements modelling approach for IoT systems combining Volere template (for organizing requirements in English), UML use cases and SysML diagrams. Their approach considers exceptional cases in use case diagrams by finding abnormal conditions, but, it does not specify types of exceptions and actors like UCM4IoT. There is no support for elicitation with textual use cases. Although this work targets the IoT domain, their method is generic and does not specify any IoT-specific requirements. Sosa-Reyna et al. [28] propose a methodology to develop IoT systems with a service-oriented approach. They used formal transformation rules to go from one phase to another in their four phase model-based system development methodology. In the first phase of requirements engineering, UML use case and activity diagrams are used to elicit both functional and non-functional requirements. They also presented a metamodel to generate smart vehicle applications following their proposed methodology.

Silva et al. [29] proposed to modify the ISO and IEEE standards IEC/IEEE 12207:2017 of requirements engineering for IoT systems. They defined the

Table 3: Comparison of existing work (supports (✓), does not support (x), unknown/unclear (-)

Approach	Tool Support	IoT Support	Analysis Support	Elicitation	Specification	Validation
Sindre and Opdahl [20]	X	X	Threat analysis from the misuse cases	✓	✓	X
Savić et al. [21]	Model to Model transformation, SilabReq models to UML models	X	Requirements analysis from different abstraction levels	✓	✓	X
Cockburn [7]	X	X	X	✓	X	X
Amyot [22]	UCM Navigator for drawing use case maps	X	Use case maps to minimize gap between requirements and design	✓	✓	X
Mustafiz et al. [16]	X	X	X	✓	X	X
Maleki [23]	X	X	Supports to derive software critical path	✓	✓	X
Reggio [26]	X	✓	Supports finding unexpected behaviour and constraints to goals	✓	✓	X
Meacham [27]	X	✓	Abnormal condition analysis from use case diagram	✓	✓	X
Sosa-Reyna et al. [28]	X	✓	X	✓	✓	X
Silva et al.[29]	X	✓	-	✓	✓	✓

requirements engineering of IoT in three sub-processes: i) process scope definition, ii) IoT system definition, and iii) IoT system requirements definition. Each sub-process then follows the ISO and IEEE standard process. This work presents a process for requirements engineering, but unlike our work, it does not propose any domain-specific language or environment to facilitate the elicitation and specification of IoT system requirements.

A comparison of the related work is presented in Table 3. Currently, limited methods, techniques, or tools are available for requirements engineering of IoT systems. Most of the existing work focus on design and development phases. However, to minimize errors and changes in the downstream phases, a well-defined set of requirements is essential. UCM4IoT will help fill this gap, ensuring IoT requirements including exceptional and adaptive behaviour are discovered and documented from the early stages of software development.

11. Conclusion

We have proposed a requirements development language, UCM4IoT, catered for IoT systems. Our textual use case language provides IoT-specific language constructs for discovering the different types of actors and interactions participating in an IoT system. Our approach enables exceptional scenarios associated with the different facets of IoT (hardware, software, network, and environment) to be identified during requirements elicitation. This is followed by discovering adaptive system behaviour as handling mechanisms. UCM4IoT allows specification of these potential exceptions in use cases with supporting handler use cases to document exceptional system interactions. UCM4IoT also supports specification of different modes of operation, including the available services in each mode. A textual modelling environment for UCM4IoT has been developed to assist modellers in writing unambiguous and complete use cases. The tool enables syntax highlighting, type-checking, cross-referencing, and global validation of use cases. Support for generation of exception and mode summary information is also provided to enable exploration and static analysis of the use cases. We also extended the UML use case diagram with IoT-specific elements that align with our textual language.

Our approach is demonstrated and evaluated with two IoT applications, a smart store system and a smart fire alarm system, encompassing hardware, software, human users, and physical entities all working in coordination to provide the system functionalities and fulfill the user goals.

As future work, we plan on integrating NFRs in UCM4IoT and providing support for dynamic analysis of requirements.

References

[1] L. Atzori, A. Iera, G. Morabito, The internet of things: A survey, Computer networks 54 (15) (2010) 2787–2805.

[2] G. Reggio, M. Leotta, M. Cerioli, R. Spalazzese, F. Alkhabbas, What are IoT systems for real? an experts' survey on software engineering aspects, Internet of Things 12 (11 2020). `doi:10.1016/j.iot.2020.100313`.

[3] D. Parachuri, A. S. M. Sajeev, R. Shukla, An empirical study of structural defects in industrial use-cases, in: Companion Proceedings of the 36th ICSE, ACM, 2014, p. 14–23. `doi:10.1145/2591062.2591174`.

[4] A. Shui, S. Mustafiz, J. Kienzle, C. Dony, Exceptional use cases, in: Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, 2005, pp. 568–583.

[5] P. Boutot, M. R. Tabassum, S. Mustafiz, Ucm4iot: A use case modelling environment for iot systems, in: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 2021, pp. 767–776. `doi:10.1109/MODELS-C53483.2021.00123`.

[6] I. Jacobson, I. Jacobson, M. Christerson, A. P. Staff, P. Jonsson, G. Övergaard, Object-oriented Software Engineering: A Use Case Driven Approach, ACM Press Series, ACM Press, 1992.

[7] A. Cockburn, Writing Effective Use Cases, Crystal collection for software professionals, Addison-Wesley, 2001.

[8] E. Nasr, J. McDermid, G. Bernat, Eliciting and specifying requirements with use cases for embedded systems, in: Object-Oriented Real-Time Dependable Systems (WORDS 2002), 2002, pp. 350 – 357. `doi:10.1109/WORDS.2002.1000073`.

[9] S. Sendall, A. Strohmeier, From use cases to system operation specifications, in: UML 2000 — The Unified Modeling Language, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 1–15.

[10] R. Wirfs-Brock, What it really takes to handle exceptional conditions, in: forUSE 2002 Proceedings, Ampersand Press, 2002.

[11] A. Bassi, M. Bauer, M. Fiedler, T. Kramp, R. van Kranenburg, S. Lange, S. Meissner, Enabling Things to Talk: Designing IoT Solutions with the IoT Architectural Reference Model, 1st Edition, Springer Publishing Company, Incorporated, 2016.

[12] ISO Central Secretary, Information technology — reference architecture for service oriented architecture (soa ra) — part 1: Terminology and concepts for soa, Standard ISO/IEC 18384-1:2016, International Organization for Standardization, Geneva, CH (2016).
URL `https://publications.opengroup.org/standards/soa/c119`

[13] I. Sommerville, Software engineering 10th edition, ISBN-10 137035152 (2015) 18.

[14] M. Bano, D. Zowghi, N. Ikram, M. Niazi, What makes service oriented requirements engineering challenging? a qualitative study, IET software 8 (4) (2013) 154–160.

[15] N. Sasikala, K. Rangarajan, Study on use case model for service oriented architecture development, International Journal of Web Technology 1 (1) (2012) 1–4.

[16] S. Mustafiz, J. Kienzle, DREP: A requirements engineering process for dependable reactive systems, in: Methods, Models and Tools for Fault Tolerance, Springer, 2009, pp. 220–250.

[17] S. Mustafiz, J. Kienzle, A. Berlizev, Addressing degraded service outcomes and exceptional modes of operation in behavioural models, in: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, SERENE '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 19–28. `doi:10.1145/1479772.1479776`.

[18] R. Li, T. Song, N. Capurso, J. Yu, J. Couture, X. Cheng, IoT applications on secure smart shopping system, IEEE Internet of Things Journal 4 (6) (2017) 1945–1954. `doi:10.1109/JIOT.2017.2706698`.

[19] S. Karjol, A. Holla, C. B. Abhilash, An IoT based smart shopping cart for smart shopping, in: CCIP 2017, 2018, pp. 373–385. `doi:10.1007/978-981-10-9059-2-33`.

[20] G. Sindre, A. L. Opdahl, Eliciting security requirements with misuse cases, Requirements engineering 10 (1) (2005) 34–44.

[21] D. Savic, A. R. da Silva, S. Vlajic, S. Lazarevic, V. Stanojevic, I. Antovic, M. Milic, Use case specification at different levels of abstraction, in: International Conference on the Quality of Information and Communications Technology, 2012, pp. 187–192. `doi:10.1109/QUATIC.2012.64`.

[22] D. Amyot, Use case maps as a feature description notation, in: Language Constructs for Describing Features, Springer, 2001, pp. 27–44.

[23] H. Maleki, A. Jamshidi, M. Mohammadi, A framework for effective exception handling in software requirements phase, in: Fundamental Research in Electrical Engineering, Springer, 2019, pp. 397–411.

[24] S. S. Some, An environment for use cases based requirements engineering, in: RE '04, IEEE, 2004, p. 364–365.

[25] T. Yue, L. Briand, Y. Labiche, Facilitating the transition from use case models to analysis models: Approach and experiments, ACM Trans. Softw. Eng. Methodol. 22 (2013) 5:1–5:38.

[26] G. Reggio, A UML-based proposal for IoT system requirements specification, in: MiSE '18, ACM, 2018, p. 9–16.

[27] S. Meacham, K. Phalp, Requirements engineering methods for an internet of things application: fall-detection for ambient assisted living, in: BCS SQM/Inspire Conference 2016, 2016.

[28] C. M. Sosa-Reyna, E. Tello-Leal, D. Lara-Alabazares, Methodology for the model-driven development of service oriented iot applications, Journal of Systems Architecture 90 (2018) 15–22.

[29] D. Silva, T. G. Gonçalves, A. R. C. da Rocha, A requirements engineering process for IoT systems, in: Proceedings of the XVIII Brazilian symposium on software quality, 2019, pp. 204–209.