Stephen F. Austin State University

SFA ScholarWorks

Electronic Theses and Dissertations

5-2024

Formalization of a Security Framework Design for a Health Prescription Assistant in an Internet of Things System

Thomas Rolando Mellema
Stephen F Austin State University, mellematr@jacks.sfasu.edu

Follow this and additional works at: https://scholarworks.sfasu.edu/etds

Part of the Information Security Commons, Logic and Foundations Commons, Software Engineering Commons, and the Theory and Algorithms Commons

Tell us how this article helped you.

Repository Citation

Mellema, Thomas Rolando, "Formalization of a Security Framework Design for a Health Prescription Assistant in an Internet of Things System" (2024). *Electronic Theses and Dissertations*. 559.
https://scholarworks.sfasu.edu/etds/559

This Thesis is brought to you for free and open access by SFA ScholarWorks. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of SFA ScholarWorks. For more information, please contact cdsscholarworks@sfasu.edu.

Formalization of a Security Framework Design for a Health Prescription Assistant in an Internet of Things System

Creative Commons License

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 License.

This thesis is available at SFA ScholarWorks: https://scholarworks.sfasu.edu/etds/559

FORMALIZATION OF A SECURITY FRAMEWORK DESIGN FOR A HEALTH PRESCRIPTION ASSISTANT IN AN INTERNET OF THINGS SYSTEM

By

THOMAS ROLANDO MELLEMA, Bachelor of Science

Presented to the Faculty of the Graduate School of

Stephen F. Austin State University

In Partial Fulfillment

Of the Requirements

For the Degree of

Master of Science

STEPHEN F. AUSTIN STATE UNIVERSITY

May 2024

FORMALIZATION OF A SECURITY FRAMEWORK DESIGN FOR A HEALTH PRESCRIPTION ASSISTANT IN AN INTERNET OF THINGS SYSTEM

By

THOMAS ROLANDO MELLEMA, Bachelor of Science

APPROVED:

Christopher Ivancic, Ph. D., Thesis Director

James Adams, Ph. D., Committee Member

Jeremy Becnel, Ph. D., Committee Member

Matthew Beauregard, Ph. D., Committee Member

Forrest Lane, Ph.D.
Dean of Research and Graduate Studies

ABSTRACT

Security system design flaws create greater risks and repercussions as the systems being secured further integrate into our daily life. One such application example is incorporating the powerful potential of the concept of the Internet of Things (IoT) into software services engineered for improving the practices of monitoring and prescribing effective healthcare to patients. A study was performed in this application area in order to specify a security system design for a Health Prescription Assistant (HPA) that operated with medical IoT (mIoT) devices in a healthcare environment [1]. Although the efficiency of this system was measured, little was presented to provide verification of the given framework details to ensure the absence of design flaws that might cause security errors within the final implementation. Formal software modeling has long been utilized as a tool to combat ambiguity, incompleteness, and inconsistencies in a given system design, but these modeling methods lack frequent research application to modern technological concepts for the purpose of preventing security vulnerabilities. This study translates components of an existing security framework proposal for an IoT HPA system through the lens of three different formal design methods: Z-notation, TLA+, and Petri Nets. Each formal model is then expanded on in order to demonstrate the beginning iterative steps of how each specification method can be applied to help improve the completeness, correctness, and accuracy of any given design for a high-level security system.

CONTENTS

ABSTRACT I

LIST OF FIGURES IV

1 INTRODUCTION 1

2 LITERATURE REVIEW 5

2.1 Formal Software Design Background and Use 5
2.1.1 Z-notation: Object and Behavior Specification 8
2.1.2 TLA+: Automated Instance Checking 10
2.1.3 Petri Nets: Solving for Concurrency 12

2.2 Securing IoT Systems 14
2.2.1 Use Case: mIoT HPA Framework 15

3 DEFINING SECURITY COMPONENTS (Z-NOTATION) 18

3.1 Defining Security Object Sets 18

3.2 Handling Assignment and Attributes 22

3.3 SAT Generation Schema 29

4 VERIFYING AUTHORIZATION (TLA+) 32

4.1 Blacklist TLC Model Example 34

4.2 Verifying All Access Invariants 37

4.3 Correcting Authorization Algorithm 41
4.3.1 Multiple Role Assignment 41
4.3.2 Supporting Context Awareness 44

5 DESIGNING FOR CONCURRENCY (PETRI NETS) 48

5.1 U2D Petri Net Designs 48

5.2 Proving State Reachability with Incidence Matrix 52

6 FORMAL DESIGNS ANALYSIS 57

6.1 Access Control Object Z-Schemas 57

6.2 Authorization TLA+ Spec & TLC Model 58

6.3 Mutual Exclusion Petri Net Proof 60

7 CONCLUSIONS 62

REFERENCES 63

APPENDIX 65

LIST OF FIGURES

Figure 2.1 - Birthday Book Z Schema Example 9
Figure 2.2 – $H_2$0 Petri Net Example 13
Figure 3.1: HPA Authorization Components 19
Figure 3.2 - Authorization Core Object Z Schemas 20
Figure 3.3 - User Assignment Schema 23
Figure 3.4 - Operation Assignment Schema 25
Figure 3.5 - Permission Assignment Schema 27
Figure 3.6 - Context Awareness Schema 29
Figure 3.7 - SAT Generation Schema 30
Figure 4.1 - Generate SAT Algorithm 33
Figure 4.2 - Blacklist Input Model 35
Figure 4.3 – NotRevoked Invariant Example 36
Figure 4.4 - SAT Generation Invariants 38
Figure 4.5 - Sample HPA Input Model 39
Figure 4.6 - Generated SAT Output 40
Figure 4.7 - HPA Multiple Roles 41
Figure 4.8 - 1st Multi-Role Output 42
Figure 4.9 - PlusCal Role Correction 43
Figure 4.10 - Corrected TLC Roles Output 43
Figure 4.11 - Input Model w/ Context Constraints 44
Figure 4.12 - ContextAware Invariant Error Run 45
Figure 4.13 - Context Value Load Fix 46
Figure 4.14 - SAT Output w/ CC & CT 47
Figure 5.1 - U2D Request & Response Process 49
Figure 5.2: U2D Delegated Petri Net 50
Figure 5.3: Two Devices with Mutex 52
Figure 5.4 - Petri Net Incidence Matrix 53
Figure 5.5 - Mutex Reachability Equation 54
Figure 5.6 - Proof for No System Solution 55

1 INTRODUCTION

The potential for using technology to help automate the physical world has increased over recent years as extensive and practical solutions continue to be researched and implemented in various real-world contexts. An example of this includes the use of widespread smart devices, sensors, and actuators on the edge of a network to track and manipulate the environment around us based on the real-time data processing from that network, commonly referred to as an Internet of Things (IoT). While the numerosity of these devices can produce powerful data and control of an environment, this solution context also encourages the majority of edge devices in an IoT network to be low-powered and resource-constrained in order to meet the practicality standards of implementing these physical network points in abundance. These intentional resource limits make cumbersome encryption algorithms and advanced authorization logic difficult to implement for the edge of the network and therefore give way to key security concerns over a broad attack surface [2]. Encouraged by this, various research has been done to try and address the heightened security risks that come with implementing an IoT system into real-world applications. One piece of this research set out to propose a detailed IoT framework for the security systems surrounding a health prescription assistant (HPA) that would theoretically provide many conveniences to the normal healthcare process through the use of medical smart device IoT (mIoT) technology [1].

Formal software design methods have long been utilized as solutions to help provide a provably secure and/or complete design of any especially critical software system. There has been a multitude of design methods used to improve upon various steps in the design process and/or to demonstrate subtle use cases and their possible execution flaws within applications of a system. The goal that they all have in common is that each method seeks to provide a measurably complete or measurably correct overview of any component of a software system using ideas derived from the same rules and reasoning used in discrete mathematics. Because these methods frequently require the system design and all additions to be completely logically sound, they provide more reasoning power with less ambiguity than traditional informal software design practices.

The usefulness of an IoT environment with its high automation potential is mirrored by its heightened risk of an increased area of exposure of smart devices that are susceptible to cyber-attacks. Thus, as the number of systems that utilize an IoT environment increase exponentially [2], so does the need for creating well-designed security systems and services that oversee monitoring, authorizing, and authenticating the various devices and interactions throughout the IoT system. Furthermore, an especially sensitive application context, such as an IoT system managing healthcare services between staff and patient users, places an even greater emphasis on the importance of having reliable and complete security so that IoT system errors do not put the lives or health of the patients enrolled in the system through any undue risk.

This specific context is where formal software design can be an extremely applicable tool to use to meet the highly intensive need for these security software systems to be complete, correct, and fundamentally safe. However, among current research, there is a lacking application of these methods to an increasingly implemented and vulnerable system concept such as IoT. This is possibly due to the low popularity of its use in the software industry due to the perceived low return-on-effort provided by the formalization process, and the relatively high mathematical/reasoning skills typically required to create the formal designs [3]. This assumed low-payoff perception could be consequently causing critical system flaws to be left undiscovered until implementation has already begun or during the live use of the application, where repercussions would be more costly and unsafe, rather than discovered in the preliminary design phase of a system. Therefore, the demonstrations, verifications, and analyses of the formal methodologies applied in this paper are an attempt to provide a fresh angle of research application to a concept area that invites the need for precise and secure system modeling.

Several established formal methods have been chosen to be used in this study to give an existing proposed mIoT security framework the opportunity to be translated through the detailed design lens that is formal specification. The translations apply these methods to an HPA security system in an IoT environment while concurrently evaluating the correctness and completeness of the given research's original designs as each model is presented. The three formal modeling methods that are utilized within this study include Z-notation, TLA+, and Petri Nets. Each of the three models presented are then analyzed on what the

design provided, how it improved and secured the HPA system, and further steps that can be taken with each method to continue increasing the benefits or applicability onto a real-world complete HPA mIoT system. Conclusions are then drawn on the usability of the example formal methods and what unique benefits they can provide for ensuring the software design process effectively meets a set of given security requirements.

2 LITERATURE REVIEW

The research sources available on the Internet of Things along with its emphasized security requirements are abundant due to its relatively recent uptrend in industry interest. In comparison to this, the research available for formal software design is more chronologically spaced out, explanatory in nature, and rarely applied to the same IoT security system topics. This review introduces the sources referenced in this paper that were used to help create and verify the three formal methods presented as the experiment. Then a more detailed summary is given for the Health Prescription Assistant (HPA) IoT system proposed in the research that was chosen to formally model from the paper "*An Internet of Things-Based Health Prescription Assistant and Its Security System Design*", as it relates to the HPA security components that are subsequently translated into each of the three formal models [1].

2.1 Formal Software Design Background and Use

The formal specification and verification of software is a process that has been developed and used on computer systems for over fifty years [4]. It is a process that relies on heavily precise notation and logically sound object/behavior specification of either what a system is and/or what the system is required to accomplish. If a formal model is held up to these standards, then the model (and therefore system) is given the potential to be reasoned about on the same logically accurate level of mathematical reasoning. This reasoning potential can provide several unique benefits to the design of a system including

the precise understanding of the model and its properties, preemptive logical error detection in the design before code implementation begins, the potential for automated testing or edge case checking, and provably correct deductions about the properties of a system [5]. The preceding benefits are what is hoped to be gained from modeling the example IoT HPA security system in the three chosen formal methods demonstrated in this paper. Although most formal languages usually offer the potential to derive all of the previously listed benefits within its own methodology, each formal method presented in this paper shows a clear specialization in the type of improvements the method can add to the design compared to the other methods shown.

Another common benefit that is unique to formal modeling as opposed to traditional software modeling methods is universal standardization of all designs made with the same formal method. Frequently, informal software system designs can be inconsistent between different applications, heterogeneous technology stacks, or even between separate components within the same application. Formal models abstract away all of the hardware specifics and library dependencies to where the shared design of the system is purely conceptual while remaining complete and logically valid [6]. This allows for additional components to the system to be checked under the same standard as all additions before it. In a real-world case, this standardization has also helped to allow better mental synchronization for engineers on the details of a how a system works for complex products being made at Amazon Web Services [3].

However, although the benefits of formal modeling can be uniquely powerful, these processes are largely under-utilized in the current state of the software development industry [3]. This can be due to a widely shared belief that formal modeling requires too much skill, takes too many hours to create, and does not have a cost-effective return on effort compared to traditional informal design methods [3]. In the appropriate problem contexts, this high-entry fee of formalization can be outweighed by the various strengths gained from a standardized specification, but the return on investment would most likely be correlated to the complexity of the system being designed and the criticality that all requirements are effectively met. A recent source reporting on the industry use of formal verification commented on this by saying, “With the ever-increasing complexity of software and the layers of abstraction, we have reached a time when writing secure, efficient and resilient code requires some level of formal verification to be done, if not for the whole software at least for the important sub-systems involved” [7].

A final assertion on the worth of balancing the benefits and costs of formal modeling is beyond the scope of this study. Although these factors are considered and analyzed based on the results of this experiment that provides three formal method demonstrations, the primary goal of this study is to explore the potential of making security systems more secure through formal modeling beyond what is already presented in currently accepted standards of informal system design. This is the main motivation behind choosing three different formal methods that should each emphasize a different area of improvement to the software design process. Each presented design should then translate to concrete

security improvements in the final implementation of the example healthcare IoT security system.

To better understand each formal design presented in this paper, basic formal design structure and use are summarized in the following sections as it relates to each of the three chosen methods: Z-notation, TLA+, and Petri Nets.

2.1.1 Z-notation: Object and Behavior Specification

Z notation is a formal specification language that specializes in precisely modeling object properties, behaviors, and interactions using similar notation and logic used under set theory in mathematics [8]. It was first developed in the 1970s and has been used for several reputable industry design specifications since then [9]. It is not a specification language that is directly executable, but it rather focuses on giving an unambiguous representation of the abstract objects of a system by defining their properties, interactions with other pre-defined objects, and expected input and output [9].

Specifications under this method can accomplish this by creating schemas. Schemas are named after the object or behavior that it is trying to define, and they show definitions in two separate sections: declarations and predicates [8]. The declaration or top section gives the variables along with their variable types that are needed to outline the properties of the system object. The predicate or bottom section specifies any restraints on the declared variables that must hold true at all times for each defined property of the system [8]. To illustrate a basic example, schema examples for a birthday book are given in Figure 2.1 [10].

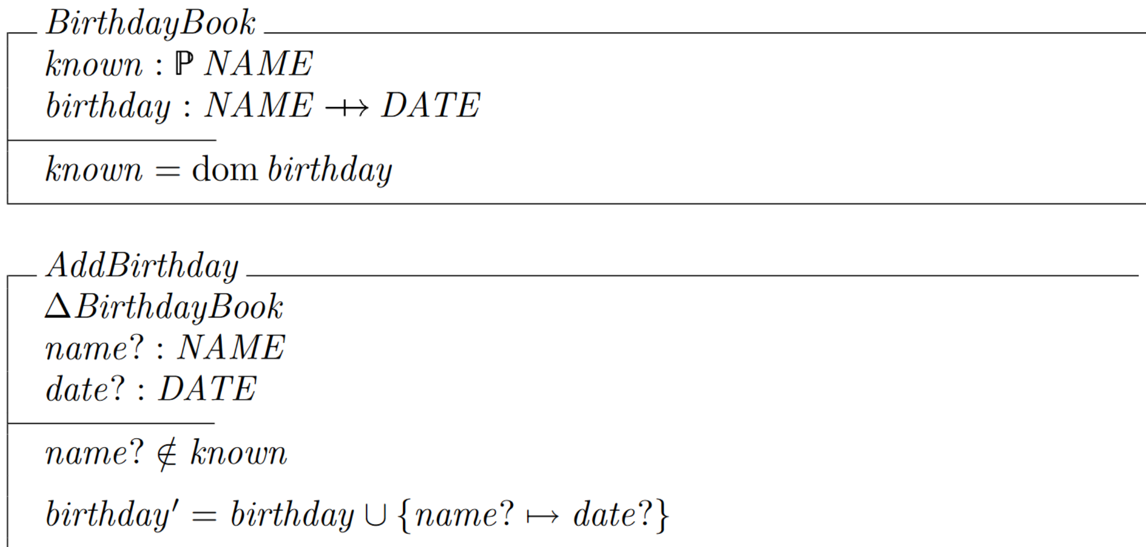

Figure 2.1 - Birthday Book Z Schema Example

In this schema example provided from reference [10], we have an initial object declared in *BirthdayBook*. It represents an object that would hold the names and dates of birthdays in a record book. Variable *birthday* would be a partial mapping function of names to dates and variable *known* would be a set of these names. The reason this relation requires partial definition is because it is possible in this book to map the same name to different dates in order to represent the birth dates of two different people with the same name. Since the predicate specifies that the *known* set equals the domain of the function *birthday*, then we know that this variable would represent the entire list of known birthday names. The second schema shows the details for changing an object in this system by adding a birthday. It details input variables for name and date and describes how the name must be new, as it is not an element of the *known* set, and how these inputs are added to the existing list of birthdays through a set union of the new mapping.

Further iterations on schemas like this are how one could use Z and mathematical sets of unique values to define an entire system design. This is the basis of how this paper defines the security authorization components of the HPA system in Z notation. Full mathematical notation can be found in the Z notation references used for the creating of the designs in this paper [8-9]. These schemas were created using downloaded zed style options in LaTeX and references to commands and explanations were gathered from these zed sources [11-12]. Use of Z in research examples of different software systems were also used as reference for presentation and organizations of designs in this thesis [13-14].

2.1.2 TLA+: Automated Instance Checking

Temporal Logic of Actions (or TLA+) is a formal modeling language that is used to provide a translated write-up of an algorithm presented in the example framework using downloaded tools and resources provided on the TLA+ website [15]. It was invented by Leslie Lamport in the late 1980s as a way to describe systems through the definition of connecting mathematical formulas [16]. Recently, it has found industry use at Amazon due to its ability for its specifications to be executed in verifiable ways to find subtle edge cases and/or provide automated testability in the design of highly complex systems independent of the actual code implementation [3].

Testability of a design under TLA+ is first achieved by creating a specification file in the TLA+ language. The language is based off of specifying set behaviors using propositional and predicate logic [16]. This is similar mathematical reasoning described and used for Z notation. Documentation for the syntax of this language can be referenced

in the language textbook [16] or within the website reference that contains many beginning TLA+ concepts [17]. As an alternative, specifications made within the TLA+ Toolbox can also run from algorithms created in PlusCal. PlusCal is another algorithm language that automatically translates to TLA+ code using options provided in the TLA+ Toolbox [18]. Many users of this formal method prefer the use of PlusCal over raw TLA+ code [17], and that is also how the specification presented in this paper is generated.

Once an algorithm is specified in this language, the user needs to identify system invariants in order to be able to verify the correctness of the design in question. System invariants are logical formulaic definitions of what must be true for all related properties in the invariant throughout all steps of execution of the algorithm [17]. A typical simple and initial invariant of a system would be a type invariant. This would make sure that all given properties in the initial state of the system would match the data types defined in this invariant, and if it fails, then that would signify that the input model is incorrect, or the properties of the algorithm start in an incorrect state [17]. However more frequently, invariants are defined to be used to check throughout the execution of an algorithm over several different input cases in order to find an enumerated path that breaks the invariant and thereby show a flaw in the system design [16].

Execution models of TLA+ specifications can be created with the included software within the TLA+ Toolbox with the TLC Model Checker. Models allow for specifying ranges of instance checking to do automated testing with the connected algorithm [17]. Within the model you can also choose only a subset of all defined invariants to check

against so that design tests can target specific areas of logic within the system. These models serve as the input cases for the HPA users and mIoT devices within the system formalization presented in this paper, and they are verified within the translated authorization algorithm given in the original HPA research.

2.1.3 Petri Nets: Solving for Concurrency

Beyond system specification in its initial static state, there are potential issues about a system's dynamic flow of execution that can cause erroneous behavior even if the final implementation matched original informal specifications. Petri Nets are another formal modeling method that require designs to accurately represent the state flow properties of software execution. Still modeled under mathematical structures and reasoning behind the design, Petri Nets also can primarily serve as a clear visualization of how input and output can show certain behavior throughout execution of a system [19].

A Petri Net design structure requires several elements in every net. A place (or circle) that represents a possible state of the system. A transition (or a bar or box) that represents a state change between places. Tokens (dots within the circle) are the input to a Petri Net and are required for a transition to fire, and they are always contained within a place. There must then be an arc (arrow) that is directed towards the transition that requires a token from the connected place. Arcs can also require a certain amount of tokens in order to fire the connected transition [19]. All of these structure rules represent abstract moments in time for what state a system can be in. These Petri Net graphs can also be translated into a corresponding matrix to represent all values of the system in a specific state. A basic

example of the execution of a small Petri Net can be found in an existing research article that gave an overview of the functionalities of these nets. This example is shown in Figure 2.2 [20].

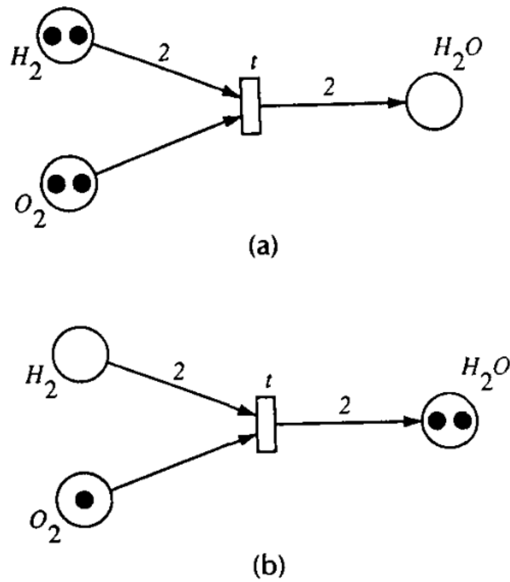

Figure 2.2 – H_20 Petri Net Example

From this example [20], the place values in the graph represent the atoms and a subsequent molecule of water. The arcs connecting H_2 and O_2 to transition t then show what is required for that state to change or for the transition to fire. The dots represent the beginning token values for each place that the system has, and the second diagram shows what tokens were passed to the transitioned state and what tokens were left in their initial place after the transition t fired. Since the O_2 arc only required one token, there is one token left in that place after the transition [20].

Any standard Petri Net can be translated to a matrix representation of all current values of each place and transition to represent the whole current state of the system. Further

explanation for the setup of this matrix can be found in the associated research [20]. What is especially powerful about this matrix representation is that other states of the graph can be calculated as reachable or unreachable based on the solution to a matrix equation called the incidence matrix. This incidence matrix is what is used in the designs of this paper to demonstrate a proof of state reachability by the device request structure of the Petri Nets modeled after the HPA system.

2.2 Securing IoT Systems

The Internet of Things is a software system concept that was originally offered decades ago but has been gaining traction recently due to its increased implementation viability through using modern technologies. Various recent studies have been performed around the concepts of IoT systems as they relate to the many security risks that naturally come with their implementation. An encompassing survey paper done in 2019 covered the prominent vulnerabilities present in each architectural layer of an IoT system [2]. Of the IoT layers covered: sensing, network, middleware, gateway, and application layer, each had a major security threat that was either directly or indirectly caused by a failure in authentication or authorization protocols between some area of communication done within the IoT system [2]. Besides access and access control attacks specifically listed in the survey, other attacks mentioned such as node capturing, cloud malware injection, and secure on-boarding are made initially possible through a failure in request authorization or user authentication since each attack starts with restricted access to a different area of the

IoT system [2]. The survey goes on to list scenarios in IoT application that require significant improvement before they can be safely implemented and available for widespread use, and one of these suggestions is as follows: "Whenever a device wants to interact with another device, an authentication process should be implemented. Digital certificates can be a promising solution to provide seamless authentication with bound identities that are tied to cryptographic protocols" [2]. This aforementioned gap in available authentication research for the security of IoT systems is part of the basis of the research that is utilized in this thesis as the experimental use-case of a given IoT security framework to formally design.

2.2.1 Use Case: mIoT HPA Framework

The primary research source for this thesis is an experiment done to try and improve the security standards surrounding an IoT-implemented health prescription assistant (HPA) by proposing a detailed security system [1]. This theoretical system is presented with models of components, component relationships, and services within the HPA and the specific authentication and authorization processes that would be used to ensure correct access within all communication done within the system. The authorization service detailed in the paper is in charge of issuing a digital token, which they frequently refer to as a security access token (SAT). After a user has been authorized, this SAT is primarily used to verify all access rights in subsequent communication the user (or client) has with the system and its devices [1]. This proposed implementation of utilizing an SAT is similar to the idea of a "digital certificate" suggested as a solution from the survey paper [2].

The provided use case also presents a unique handle on access control compared to its presented research counterparts within its related works section by proposing a delegated context-aware capability-based access control (DCCapBAC) model to handle user and request authorization [1]. This type of access control firstly emphasizes that all device permissions are assigned to the roles and not the users of the system. The research asserts that this style of access control offers flexibility and scalability required of an IoT environment [1]. In addition to this, the research explains that the context awareness of the access control allows for further control and security of the mIoT devices because access can be variable to the current values of the requested devices outlined by their associated context constraints [1]. These constraints can be loaded during SAT generation and will be sent along with the authorized request within the condition script (CS), and this script is simple stack execution that is verified within the ACLogic engine embedded in a smart gateway device close to the edge of the network. This context awareness also allows relative ease when updating IoT edge access policy because only the logic with the cloud authorization service would have to be updated to load the new CS, and the devices and gateways at the edge of the network can handle evaluating the new policy without re-implementation of these lower-powered systems which is frequently a costly and time-consuming process [1]. The access control model associated with supporting these capabilities is the primary focus of the Z schemas and TLA+ specification presented in this thesis.

The research goes on to emphasize its use of delegating constraint verification logic to smart gateway devices close to the edge of the network. This differs from common implementations of verification flow by not having it verified at the edge of the network within the mIoT device and by not having the authorization service perform it on the cloud, as both of these methods can cause inefficiencies in a quickly scalable environment such as IoT [1]. This delegated approach offers a good balance of resource and device management with smaller request delivery delays [1]. This approach is emphasized in the presentation of the Petri Net execution design in this paper.

The provided design details within the authorization service that define component sets, algorithms for SAT generation, and the process to verify these SATs outline the start-to-finish example framework that is translated into formal software designs as the primary experiment in this thesis [1]. These details include set representations, flow diagrams, and high-level algorithms throughout the paper to illustrate its contents. Although the presented details of the system seemed encompassing, its completeness for given sets of HPA use cases or its correctness for expected input/output with the given algorithm wasn't validated within the original paper. This validation and other aspects of design clarity are the foundation of what the created formal designs in this thesis are attempting to address.

3 DEFINING SECURITY COMPONENTS (Z-NOTATION)

The first aspects of the HPA security system that are formally modeled are the security components defined within its Authorization Authority service. This service is in charge of storing all access policies defined in the system, and it accomplishes this through context-aware capability-based access control (CCapBAC) logic so that every user's access is defined on their assigned role capabilities instead of uniquely defining access for each user. The service enforces this access control logic by generating a security access token (SAT) that is attached to every request within the system for a medical IoT (mIoT) device with the intent to send a lightweight and efficient validation script, or the condition script (CS), that will be verified later within the local network of the device. The SAT issued by the authorization service contains a list of authorized actions based on the mIoT device(s) they are requesting, and generates this list based on pre-defined access permission. These permissions account for users, roles, device operations, and context values (context-aware) to be later evaluated with a 'permit' or 'deny' access value by the ACLogic engine within the device network's smart gateway.

3.1 Defining Security Object Sets

We will define the necessary sets, objects, and functions in various schemas to outline the full authorization model that will each apply standard Z-notation formatting. These schema definitions will be needed in the authorization service to accomplish complete and accurate SAT generation and to uphold all access rights in any theoretical implementation

of this framework by making sure object definitions are complete and unambiguous translations of the original HPA security components. Figure 3.1 depicts each of these components and the conceptual flow of how they map to each other through assignment relations along with each of the attribute constraints that can limit these assignments [1].

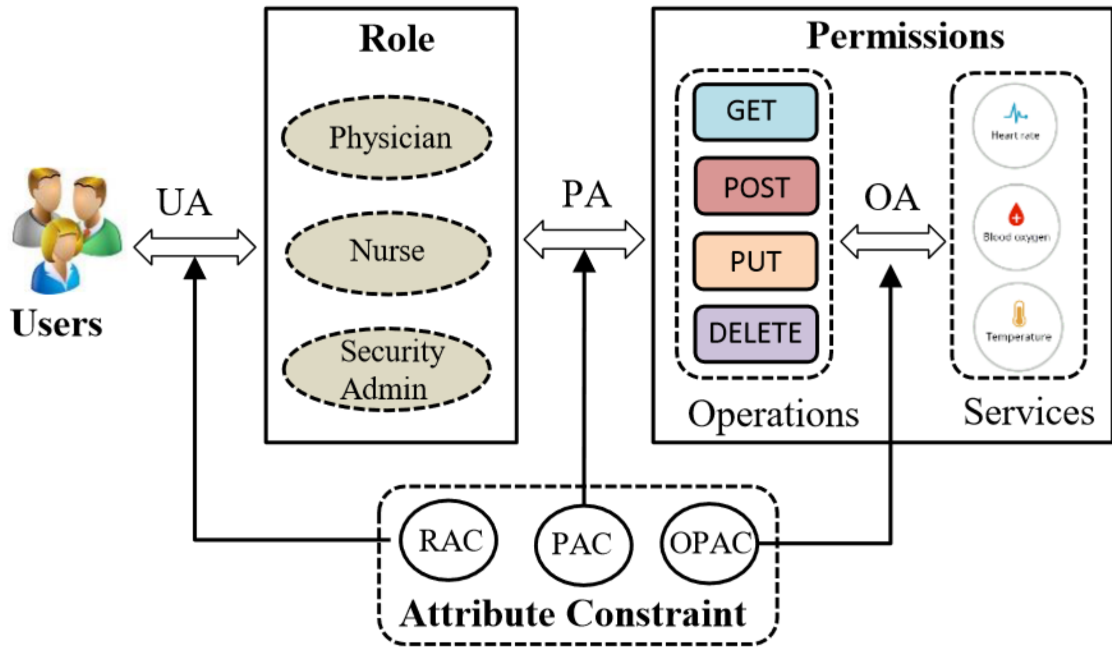

Figure 3.1: HPA Authorization Components

With these components shown from the example system, we can see a few core objects that will be integral to design the beginning details of our full Z-specification. First to define data types relevant to determining access control, we introduce four unique types of stored data: Subject, Role, Operation, and Resource. These separate data concepts establish the four basic type Z-definitions available in our authorization schemas:

[*SUBJECT*, *ROLE*, *OPERATION*, *RESOURCE*]

All initial object definitions consist of defining sets based around these data types. Since the value types are unique from each other, direct comparison and equality between different basic types is not logically sound across predicates and variables in all presented schemas. Therefore, we require four separate initial schemas to establish the definitions of the core objects in our authorization system which can be seen in Figure 3.2.

Users

$$all_users, black_list : \mathbb{P}(SUBJECT)$$
$$num_users : \mathbb{N}$$

$$all_users = \{user : SUBJECT \bullet user_1, user_2, user_3, ..., user_{num_users}\}$$
$$black_list \subseteq all_users$$

Roles

$$all_roles : \mathbb{P}(ROLE)$$
$$num_roles : \mathbb{N}$$

$$all_roles = \{role : ROLE \bullet role_1, role_2, role_3, ..., role_{num_roles}\}$$

Operations

$$all_actions : \mathbb{P}(OPERATION)$$
$$num_actions : \mathbb{N}$$

$$all_actions = \{ACT : OPERATION \bullet ACT_1, ACT_2, ACT_3, ..., ACT_{num_actions}\}$$

Services

$$all_resources : \mathbb{P}(RESOURCE)$$
$$all_services : \mathbb{P}(\mathbb{P}(RESOURCE))$$
$$num_resources, num_services : \mathbb{N}$$

$$all_resources = \{RES : RESOURCE \bullet RES_1, RES_2, RES_3, ..., RES_{num_resources}\}$$
$$all_services = \{S : \mathbb{P}(RESOURCE) \mid S \subseteq all_resources \bullet S_1, S_2, ..., S_{num_services}\}$$

Figure 3.2 - Authorization Core Object Z Schemas

The *Users* schema has two sets defined of type *SUBJECT*. A subject represents any entity that could be using the HPA system, as it won't always be physical users because it could also be automatic services or other mIoT devices that send a request for each other [1]. The first set in the first schema represents the list of *all_users* that are currently defined in the system with the amount being represented by *num_users*. The *black_list* represents the list of all defined users that have been added to the revocation list needed later in SAT generation [1]. The total users contained in the blacklist must be less than or equal to the amount of all defined users, and this is specificied in the predicate through the subset constraint on the *black_list* variable.

The next schema presented for *Roles* creates a similarly defined set using the set of *all_roles*. It is worth mentioning here that the original set given in the use case [1] intended each element of *role* to actually be a group of users, so this would remove the need for our schema to define it under its own *ROLE* unique data type. However for simplicity and clarity of the system, the set of roles are a separately defined set of values, as that would better represent the HPA needing specific role definitions in order to create access permissions.

The *Operations* schema is defined similarly and represents the possible network actions available to request for each mIoT device. This concept representation is also why the elements with in the *all_actions* set is represented by the *ACT* variable, as this is the JSON object tag name of the required set of actions referenced later during SAT generation.

The *Services* schema details the set of *all_resources* (or mIoT devices) in the HPA system as well as the set of *all_services* where each element contains a subset of all resources that exist within the system. The rest of the presented Z designs only refers to the *all_resources* set when deciding permission assignment, as the acutal executing resource for the mIoT device request is all that is contained within the final SAT since that is what is defined later in the use-case [1]. The defintion of *all_services* only accomplishes further representation of original design by adding a property that represents the original set definitions of services provided in the use-case [1].

Another detail of note in all four beginning schemas is that the variable for defining the size of each core component set of the HPA authorization service is defined under the natural number set and not the integer or whole number set. This is because the natural number set starts at 1, and there must be at least one of each security component defined in the HPA system at all times.

3.2 Handling Assignment and Attributes

We now need to handle specifications for mapping the core components to each other in a way that defines all authorized access within the system. With many common access control implementations [1], we would have enough unique basic data defined with the four core schemas already given, but since access control with the use-case is context-aware and capability based, we need to define two new basic types:

[*ATTRIBUTE*, *CONTEXT*]

The attribute basic type represents a background tuple of *{name, value}* that can be added as any string values in the actual implementation of the system, since the use-case specifies attributes to be custom-defined by the security system admin. However, this tuple is not needed in the rest of the Z designs presented for assignment because we only need to know if the corresponding data type is mapped to the attribute or not. We won't need to check or compare the actual values of attributes given to successfully define access.

The context data type acts similarly to the user and role types defined before, but it is only relevant after assignment and SAT generation have been completed.

With all necessary data types now defined, we need to define assignment between users and roles in the HPA system. The schema for this is shown in Figure 3.3.

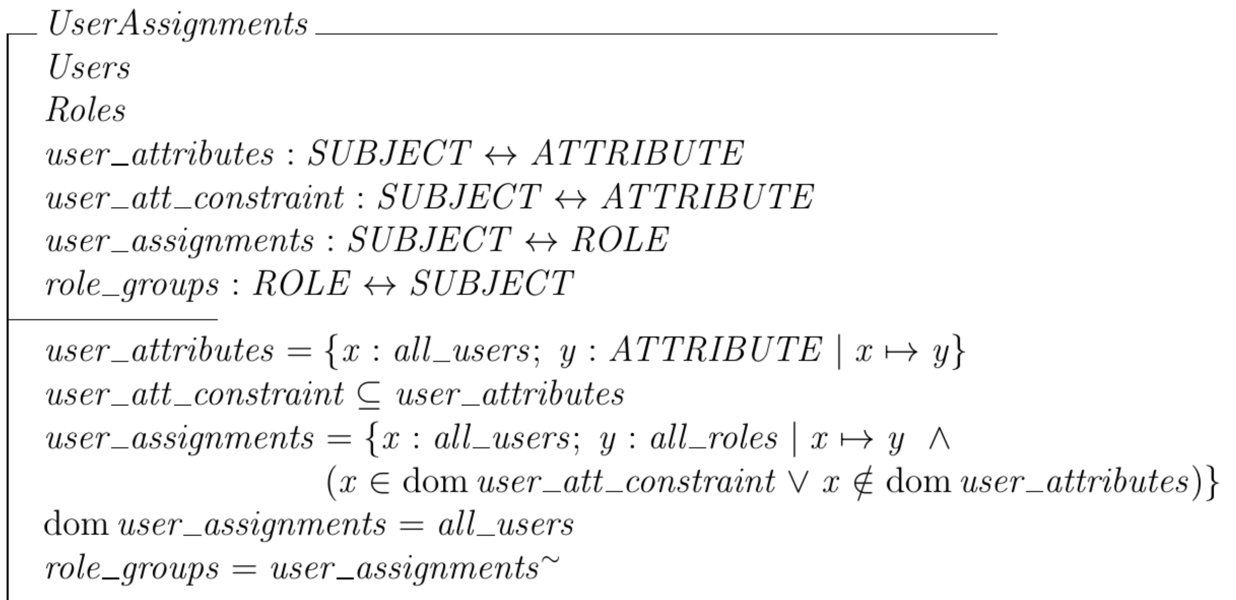

Figure 3.3 - User Assignment Schema

This assignment schema inherited the properties of the *Users* and *Roles* schemas and added four new variables. The first two are used for enforcing attribute assignment and constraints on the set of user assignments. Attributes are not required to be assigned to all users, but the attribute constraint set will always be a subset of the set of users with defined attributes, as this set represents all users left over after the admin-defined constraints have been applied and filtered on the list of all attributed users. Therefore, the full *user_assignment* mapping domain consists of all users assigned to roles from this constrained set and all users assigned to roles that did not exist in the original attribute set. A side-effect of this design intention is that all users that have no attributes assigned will automatically keep any assigned roles and are never filtered out until their attributes are assigned. This design caveat is in keeping with the original use-case requirements as attributes are only mentioned to have the purpose to add flexibility and customization to HPA access control and not to be the primary tool for limiting user access to devices [1].

The HPA use-case also defines that all users must be assigned a role, so the set union of our constrained user set, and all unattributed users should equal the domain of *all_users*. This is specified in the predicate of the Z schema. The *role_groups* variable is a property to address the previously mentioned requirement of the original design that each role element will contain a list of assigned subjects. While we didn't match this definition previously for the *Roles* schema, the *role_groups* set should effectively accomplish the same representation by populating through the Z-notation inverse set of *user_assignments*.

This schema will not require mention or predicates for the *black_list* set of users, since blacklisted users can still be assigned roles.

The next assignments to be made will be from available network actions onto the mIoT devices defined within our HPA system. This assignment can be specified in the following *OperationAssignments* schema in Figure 3.4.

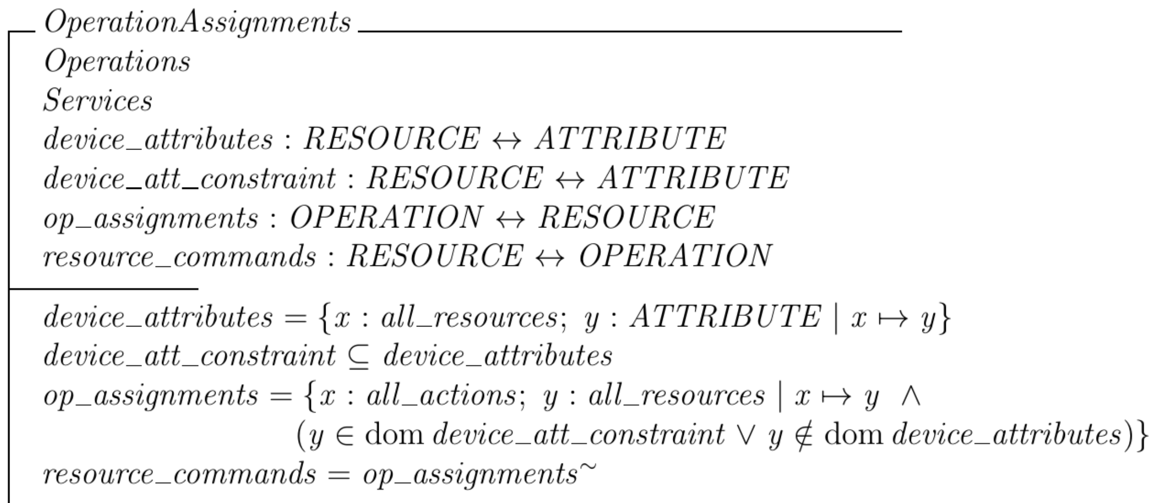

Figure 3.4 - Operation Assignment Schema

This schema accomplishes operation assignment with very similar logic described in the previous *UserAssignments* schema. Attributes are defined and attached to devices in the range of the mapping of *op_assignments* rather than the domain, since device attributes are much more useful and relevant to managing the HPA system than any network operation attributes, and this is how it is also described in the original use-case system. This causes the constraint set and predicate on assignment mapping to restrict the mapped-to variable, *y,* instead of the mapped-from variable, *x*. The schema also contains a set to reference all

resource_comands because this was an additional requirement in the use-case for the S_i set, or an element in the set of services *SS* [1].

This schema is also the first case in the presented Z designs that causes a clarity/correction from the original specifications given in the use-case. Throughout several points in the paper, it references operation assignment to be from service (resource) to operation, and then later it defines the mapping as operation to resource. Therefore, this schema clarifies the conflicting details by defining it as operation mapped to resource. This is important for further specification as the sets $RES \times OP$ and $OP \times RES$ are not equal.

The research also defined the set of possible operations assigned to a given service to be the proper subset of the set of operations, or:

$$OA \in S_i \times OP_i \mid OP_i \subset OP$$

This would mean that a given service would never be allowed to be assigned all operations available in the HPA system, since the proper subset would have to be unequal to the set of all operations *OP* due to the definition of proper subset. If the z schema supported this detail in the specification, it would need to a predicate statement to not allow all elements of *all_actions* to be mapped to the same element within *all_resources*. However, this seems like an unnecessary limiting factor to the design of the system that would prevent a resource such as a heartrate monitor from getting permissible access to all operations for the roles of a senior physician or security admin for example. Therefore, the presented Z schema does not include this predicate, and it counts this as a correction to the original given design's requirement of the proper subset relationship.

The final assignment schema that needs specification is the *PermissionAssignments* schema. This schema will inherit the previous two assignment schemas and make one final mapping definition necessary to define full authorization from system user to actions on an mIoT device. This schema is given in Figure 3.5.

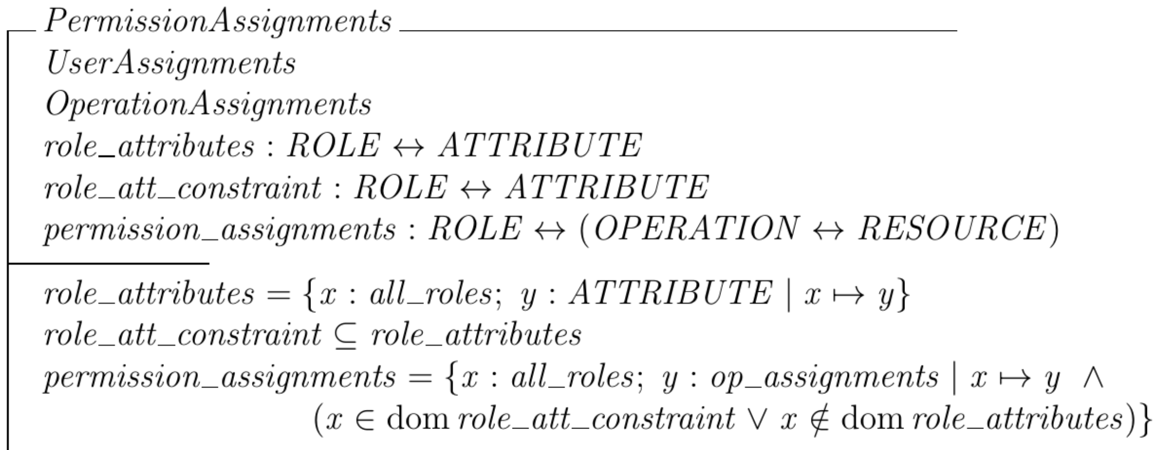

Figure 3.5 - Permission Assignment Schema

Since permission requirements given in the HPA research specify that it consists of role mapped to the relation of operation to resource, the *permission_assignments* set in this schema must restrict its possible range values to those existing elements of *op_assignments*, or the total set of possible network actions on each mIoT device. The domain type for this set is set to *all_roles* instead of the possibly more intuitive set of *range(user_assignments)*. This is because role permissions in this system will be able to be defined without requiring users to be first assigned to each role. Furthermore, the domain of this set does not need to be explicitly set equal to the set of *all_roles* to account

for the existence of roles with no permissions assigned to that role, as it would likely be the case for a Guest role of the system for example.

For the rest of the variables, the schema accomplishes attribute constraints similar to its inherited schemas, but this time it will be based on pre-defined attributes on each role in the system. The *role_attributes* set will represent the custom attributes tied to each role representing the same set as *PAA* that is presented in the use-case. The original *RAA* set in the use case is better defined by the *user_attributes* set previously outlined in this design.

The HPA example also illustrates definitions and the use of the policy set and how each role can be assigned a set of these policies. In the paper, it defines a policy to contain a subset of the permission assignments and the operation assignments [1]. The Z designs in this thesis will not specify actions based on policy, as this would be redundant definition of the same types already presented in the preceding schemas with no added payoff of clarification or functionality, since all access defined from policy can still be accomplished using the permission assignments already stated in the system.

The final component needed to fully implement our CCapBAC model will be the details behind supporting context. As mentioned earlier, context is an admin-defined set of values that can be of any type, and they are mapped off of the available resources in the HPA system. These contexts will also have their own attributes that will act as the values to check for when implementing context constraints. The *ContextAware* schema illustrates these definitions in Figure 3.6.

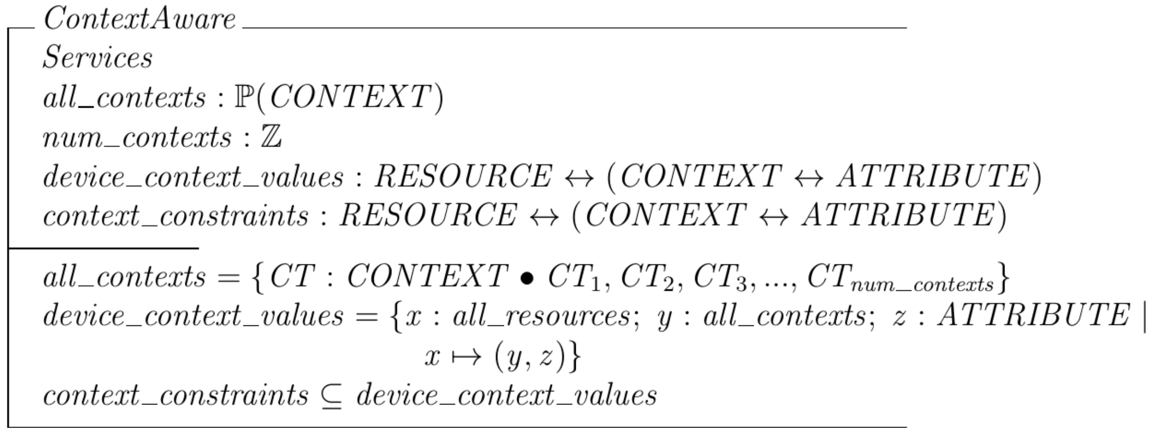

Figure 3.6 - Context Awareness Schema

The total number of contexts here is designated with an integer value in order to represent that zero contexts is a valid state of the HPA system, if the system admin does not have a need to enforce context constraints on requests. Context values and constraints are also tied to the device or set of requested devices. This is implied in the original given research, but it isn't clearly specified that contexts are dependent on which device is being requested. Since the final SAT JSON object only builds context from the requested resource, we specify these mapping details as shown in the previous schema [1].

3.3 SAT Generation Schema

With our full design of authorization components and mappings defined in Z, we can now expand upon our Z-designed system by adding further details on what it means to check and generate access permissions for a potential user requesting to use the HPA service. The schema for this definition is shown in Figure 3.7 below.

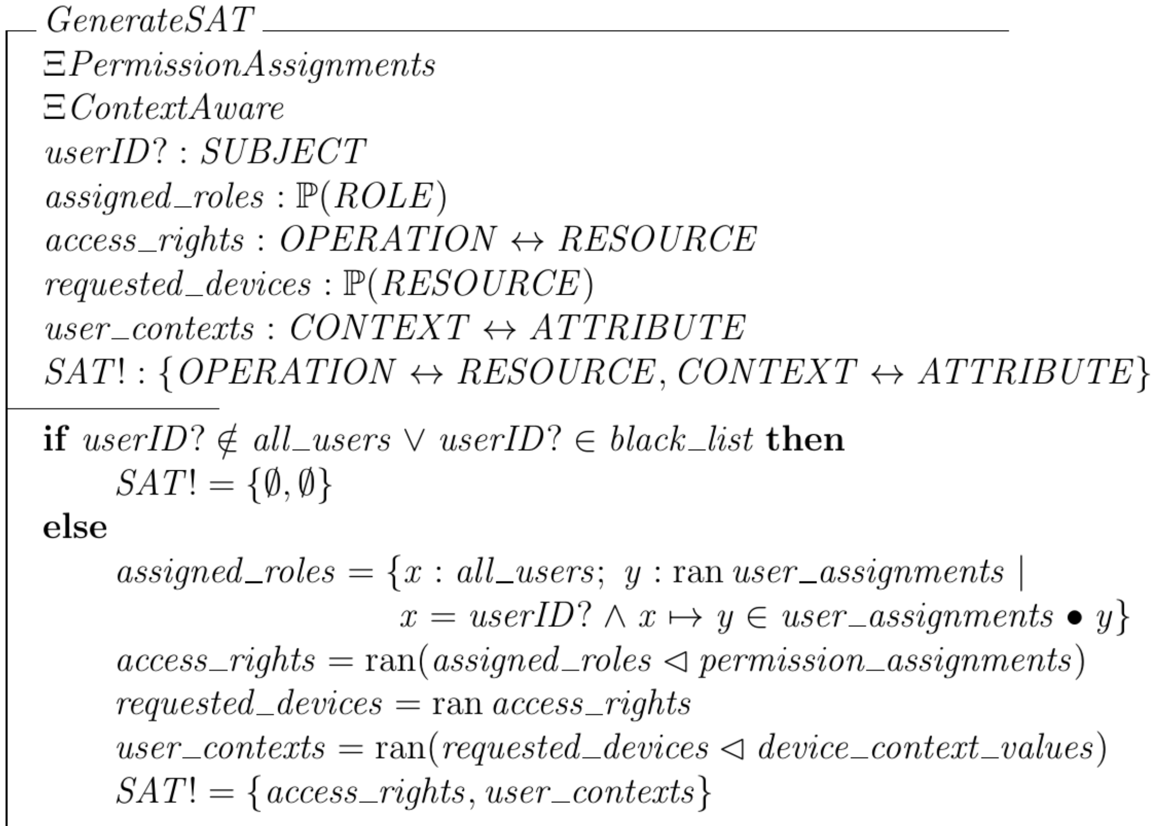

Figure 3.7 - SAT Generation Schema

Since authorization only validates and builds user permissions, the inherited schemas at the beginning of this design are marked to show no change in values of the system through execution of this function. The only input variable *userID?* is enough for our authorization system to find all appropriately assigned permissions to the user, provided that the requesting user ID exists in the system and is not blacklisted.

Through set mappings and domain restrictions, an assigned user can transitively build its corresponding set of network actions on related mIoT devices into the output *SAT!* variable. User contexts can also be derived from this list of requested devices, and these

contexts only need to be included in the final SAT output, since the context variables are evaluated later in the system at the smart gateway of the local network of the requested devices. If there needed to be schema design details on how the context constraints would actually get evaluated before fully permitting the mIoT request, then it would likely need to be detailed in a separate *SATValidation* schema, as it is out of the scope of this specific function in our HPA system.

These schemas and set formulas provide our complete definition of the components and component relationships contained within the authorization service in our HPA system, and they go further to provide the visual specification for how authorization components would be utilized in building access definitions for the generated SAT that is evaluated in the next formal design of this HPA system.

4 VERIFYING AUTHORIZATION (TLA+)

The next part of the example HPA system to formally specify will be done using the TLA+ specification language to model its presented algorithm that generates security access tokens (SATs) within the HPA authorization service. This TLA+ spec (named *HPA.tla* in Appendix) allows for automated instance checking on an input set of user access policies needed to execute the algorithm's logic for generating SAT permissions. Any unintended execution in the specification represents an error in the given HPA authorization service algorithm in enforcing correct access rights for authenticated users. In a real-world context, unintended failures in the execution of authorization would cause devices to be incorrectly accessible or inaccessible thus preventing the appropriate healthcare to be prescribed to the effected patient.

Once the TLA+ specification is complete, the TLC Model Checker is able to find any instances in a given input range that do not satisfy all invariants declared in the system throughout execution. Since a full run of the algorithm being modeled represents a successfully created SAT for the various input *UserID* values, an incomplete or TLC error run of this algorithm represents the event that a UserID has been denied access to the HPA system because they failed an access-invariant. Therefore, the necessary system invariants defined for the TLA+ specification represent all enforced conditions that must be true in order for a user to be authorized. Thus, the TLA+ specification combined with the TLC model allow for iterative testing of the accuracy of the presented algorithm by

automatically finding execution traces with unexpected outputs for accepted and denied authorizations.

For SAT generation, the Authorization Authority is tasked with taking an authenticated user ID and outputting an encrypted SAT that carries all information regarding permissions and context-aware access rights [1]. Figure 4.1 shows the SAT generation algorithm presented in the example HPA system [1]:

```
Algorithm 1: Generate Security Access Token
1  function GenerateSAT (userID);
   Input  :User ID (userID)
   Output :Signed SAT
2  if userID in revocationList then
3  |  return NULL ;
4  else
5  |  SAT ⟵ new SATInstance() ;
6  |  SAT.add(userID) ;
7  |  role ⟵ getRole(userID);
8  |  policies ⟵ getPolicyXACML(role);
9  |  for policy in policies do
10 |  |  for permission in policy do
11 |  |  |  /*
12 |  |  |  * MSN = Medical Sensor Node
13 |  |  |  * Add permission for MSN
14 |  |  |  */
15 |  |  |  SAT.addResources(permissin.MSN);
16 |  |  |  SAT.addActions(permissin.MSN.Action);
   |  |  |
17 |  |  |  /*
18 |  |  |  * EMR = Electronic Medical Record
19 |  |  |  * Add permission for EMR
20 |  |  |  */
21 |  |  |  SAT.addResources(permissin.EMR);
22 |  |  |  SAT.addResources(permissin.EMR.Action
   |  |  |
23 |  |  end
24 |  end
25 |  for res in SAT.Resources do
26 |  |  cc ⟵ getContextConstraint(res) ;
27 |  |  SAT.add(cc) ;
28 |  end
29 |  return Sign(SAT,PrivateKey) ;
30 end
```

Figure 4.1 - Generate SAT Algorithm

The above algorithm shows the source of what is translated into a TLA+ specification that is then able to undergo automated instance checking with the TLC Model Checker once an input model set is defined within the checker. There are parts of this algorithm that remain constant for every request and/or do not relate to the user access definitions outlined in the formalizations contained in this thesis. Therefore, variables relating to public/private key signing, SAT instance, or OB values are not included in the TLA+ translation for the evaluation of this algorithm.

4.1 Blacklist TLC Model Example

The logic in Figure 4.1 starts by checking if the authenticated user has been previously blacklisted by checking for the existence of the *userID* in the *revocationList* property, and if the user is found here, then the SAT returns with a null response ensuring that the mIoT request is not served. If the user is not blacklisted, then the algorithm falls into its usual process for loading all relevant information needed for permitted access in the proposed CCapAC design. This blacklisted check is the source of the first system invariant in the created specification code.

Invariants are defined states or properties of a system that are not allowed to change or be compromised throughout all possible executions of the system or else the system fails. We can use invariant definitions to check if access control is held up correctly by defining all invariant rules needed to allow a User ID access to a generated SAT. Therefore, if any

of these invariants fail, then access is forbidden and the subject requesting the service will be met with a denial response.

To introduce the most basic verification of correct algorithm logic in the TLA+ specification model with the TLC Model Checker, we will create an input set of users that will have at least one user blacklisted, and based on the specification code, the TLC model will output a representative access denied response to that user. Full TLA+ specification code, *HPA.tla*, is provided for reference in the Appendix.

The model checker requires an input of sets and their values to use in algorithm execution in order build all execution paths and find which path where any one of the system invariants fail. This first model example starts with a basic set of three users that are used iteratively as the different input *UserID* variables within the algorithm, with a single user defined with the *RevocationList* property shown in Figure 4.2.

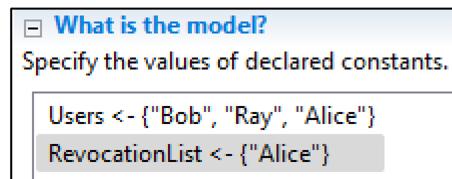

Figure 4.2 - Blacklist Input Model

Running the *HPA_Checker* model then gives an error trace of the system showing that there exists a user that is denied authorization because their user ID has been blacklisted. The output from this model run is shown in Figure 4.3.

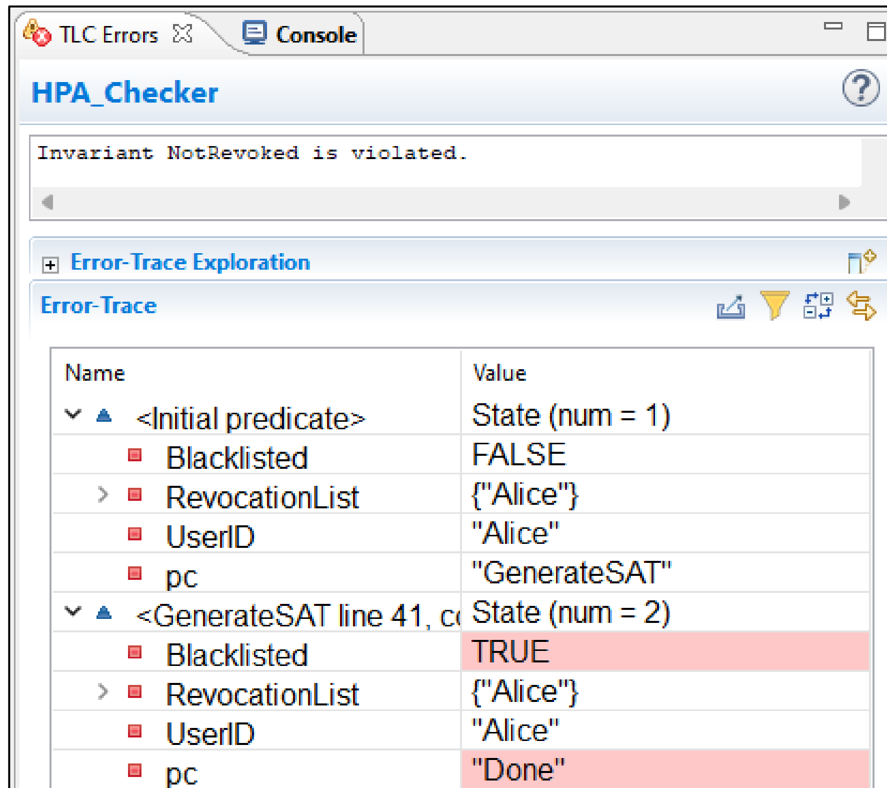

Figure 4.3 – NotRevoked Invariant Example

The system invariant *NotRevoked* represents a Boolean in the TLA+ specification that must remain TRUE at all times, or the opposite of the *Blacklisted* Boolean, or else mIoT device access is denied in order to represent that all users must not be revoked to be able to use the HPA system. Since "Alice" was a user that is defined in the *RevocationList* and is also a user in the HPA system, the model checker found a state of the system where *Blacklisted* was set to TRUE, which caused the *NotRevoked* invariant to be set to FALSE. This ultimately and correctly indicated an execution path where an unauthorized user existed from the given input and would have been denied an SAT in the real system.

4.2 Verifying All Access Invariants

In order to translate the full algorithm into a TLA+ specification, we need to define all system invariants, similar to the previous one defined in Figure 4.3. All system invariants that follow the original framework access design and also align with logic in the given algorithm are listed below alongside their invariant name in *Module HPA* (Appendix):

- *NotRevoked* – User ID must not be blacklisted.
- *RoleAssigned* – User ID must be assigned to at least one Role.
- *PermissionAssigned* – Role(s) must be assigned at least one permission.
- *ContextAware* – If context constraints exist for a resource in the SAT, then the SAT must include the same context values (CT) in order to pass context validation during SAT Verification.

These invariants outline all requirements for a valid SAT object to be generated and forwarded back to the user. Written in PlusCal and then translated into raw TLA+ code, the module specification has algorithm logic mirrored to load necessary access values into the SAT from user to role to permission (which is the set of operations mapped to devices). Every next successful load of a new value to the SAT triggers a new TLA+ label and the next invariant to be evaluated. The invariants are enforced by checking the cardinality of the sets stored within the SAT object. Since the last three of the four invariants all require a value to exist, these cardinality checks simply force the number elements loaded into each object within the SAT to be greater than zero. These invariants defined in PlusCal

syntax, along with useful functions for the algorithm logic, are pictured in Figure 4.4 below.

```
22 define
23     NotRevoked == Blacklisted = FALSE
24     RoleAssigned == pc = "LoadedRole" => Cardinality(SAT["Role"]) > 0
25     PermissionAssigned == pc = "LoadedPermissions" => Cardinality(SAT["Permissions"]) > 0
26     ContextAware == pc = "LoadedContext" => (Cardinality(SAT["CC"]) > 0 =>
27                                          Cardinality(SAT["CC"]) = Cardinality(SAT["CT"]))
28     PullSetElement(set) == CHOOSE x \in set:TRUE
29     GetDevice(perm) == {perm[key]: key \in DOMAIN perm}
30
31 end define;
```

Figure 4.4 - SAT Generation Invariants

Now that we have a full TLA+ specification with system invariants defined, we can run a set of instances through the TLC Model Checker to find cases where the system would fail these invariants based on the selected access inputs. For our first full HPA input test model, we define sets and assignments under similar structure as the Z schemas for the same objects formalized earlier in the paper. These input values are for: Users, Revocation List, Roles, Operations, Resources, User Assignments, Operation Assignments, and Permission Assignments. Definitions for context constraints and context values mapped from resources are covered in a later section of this chapter, so this input model only focuses on testing and verifying that access logic for the algorithm has expected output and satisfies all defined invariants. The full input model entered within the *HPA_Checker* file is shown in Figure 4.5.

What is the model?

Specify the values of declared constants.

```
Users <- {"Bob", "Ray", "Marsha"}
RevocationList <- {"Alice"}
Roles <- {"Nurse","Doctor","Admin"}
Operations <- {"GET", "POST"}
Resources <- {"pill_box", "pacemaker"}
OpAssignment <- [ GET |-> {"pill_box","pacemaker"}, POST |-> {"pill_box"} ]
UserAssignment <- [ Bob |-> {"Nurse"}, Ray |-> {"Doctor"}, Marsha |-> {"Admin"} ]
PermissionAssignment <- [ Nurse |-> {[GET |-> {"pill_box"}]}, Doctor |-> {[POST |-> {"pill_box"}], [GET |-> {"pill_box"}]}, Admin |-> {[GET |-> {"pacemaker"}]} ]
```

Figure 4.5 - Sample HPA Input Model

In this model we have three users and none of them are now blacklisted. Each user is assigned to a unique role with unique permissions for overlapping network actions that are available for a set of two mIoT devices. Based on this instance, our expected output from running the model would be that there should be no execution path possible that would invalidate any of the previously defined system invariants. All users should transitively build a set of permissions from their assigned roles and there are no context constraints currently defined on these devices in this instance, so context values are not needed in any iteration of an SAT.

When running the model, TLA+ and TLC functionalities do not only specialize in efficiently enumerating all possible execution paths to find potential flaws. You can also output relevant object data throughout or at the end of execution to further check that the design is getting the output that you expect. In order to see if all SAT objects for each user are getting loaded accurately, we display the results of the algorithm in the model output as shown in Figure 4.6 below.

User Output

TLC output generated by evaluating Print and PrintT expressions.

```
[ Role |-> {"Nurse"},
  Permissions |-> {{[GET |-> {"pill_box"}]}},
  CC |-> {},
  CT |-> {},
  User |-> {"Bob"} ]
[ Role |-> {"Doctor"},
  Permissions |-> {{[GET |-> {"pill_box"}], [POST |-> {"pill_box"}]}},
  CC |-> {},
  CT |-> {},
  User |-> {"Ray"} ]
[ Role |-> {"Admin"},
  Permissions |-> {{[GET |-> {"pacemaker"}]}},
  CC |-> {},
  CT |-> {},
  User |-> {"Marsha"} ]
```

Figure 4.6 - Generated SAT Output

Output for input model was possible due to no error traces being thrown which confirms that none of the system invariants were violated throughout algorithm execution representing a valid SAT being generated for all three users in this instance. The SAT object in this specification is stored as a function with the domain coming from a pre-defined set of SAT keys. These keys represent equivalent JSON parameters given in the SAT example within the research [1]. Since the domain is a set of strings, they are unordered and not displayed conveniently. However, with this output, it is confirmed that each user got their correct set of unique permissions on either of the two mIoT devices defined in the HPA. This output would suggest that the original algorithm given in the research that is now translated to TLA+ has correct and sufficient logic for loading

necessary SAT values for a given user in order to enforce effective CCapBAC authorization.

4.3 Correcting Authorization Algorithm

This section demonstrates how further iterations of TLA+ specification can find flaws in algorithm designs based on either algorithm output or breaking defined invariants. You can then improve upon system invariants or the logic of the design and this would directly translate to added benefits to the originally modeled system.

4.3.1 Multiple Role Assignment

To further assess the accuracy of the SAT generation algorithm we can change the input model from the previous test to give more variation on the possible role and permission assignments it can create. One such possible variation of an HPA input assignment scheme can be done as shown in Figure 4.7.

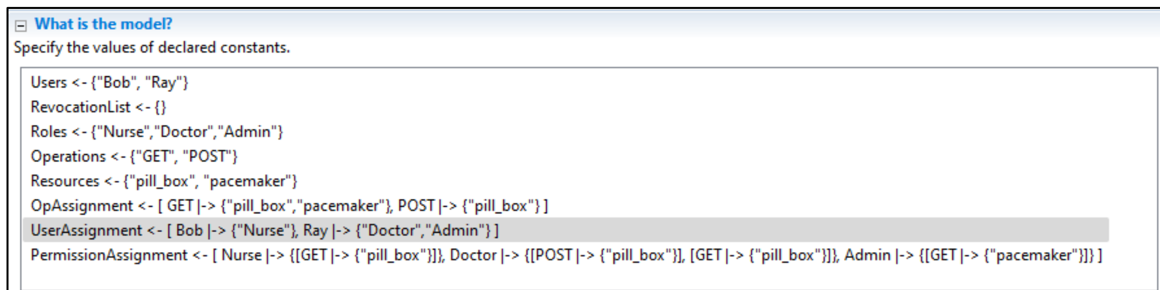

Figure 4.7 - HPA Multiple Roles

In this new input model, we can see that a user is assigned to multiple roles, Ray is defined as a Doctor and an Admin. This is specified to be a valid assignment in the HPA system by the use-case and previously formalized Z schema requirements, and it could also

represent a very common use case in a real-world healthcare system. Therefore, it follows that Ray would be expected to be assigned all of the permissions of a doctor and all of the permissions of an admin.

If we run the TLC Model Checker and check through manually coded output, we can see the final resulting SAT based off of the current algorithm logic that would have all of Ray's assigned permissions. This output is pictured in Figure 4.8.

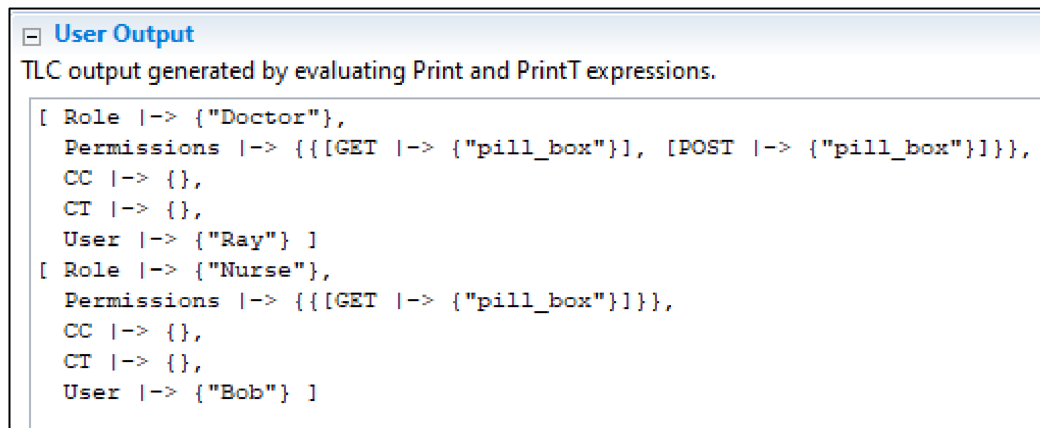

Figure 4.8 - 1st Multi-Role Output

The model was able to run without breaking any system invariants. However, if we check through the resulting values in each SAT key we see an incorrect output. Bob has all of the expected permissions of a nurse role, but Ray only has permissions for the *pill_box* resource when the input model suggests that he should also have GET access to the pacemaker since that is a permission assigned to the role of Admin.

The algorithm presented in the research uses a *getRole(userID)* pseudocode call to load the user's role and necessary policy and permissions for the user. The TLA+ spec code mirrored this functionality and chose a single role assigned for Ray when that input variable

got to that part of the execution. To fix this access right logic, we need to rework this assignment loading to gather all elements of *UserAssignment* such that the key is equal the given UserID. The way the syntax would look for this fix in PlusCal is shown in Figure 4.9 below.

```
43                  \*SAT["Role"] := SAT["Role"] \union {CHOOSE x \in Roles: x \in UserAssignment[UserID]};
44                  \*MULTIPLE ROLES ALGORITHM CORRECTION
45                  SAT["Role"] := SAT["Role"] \union UserAssignment[UserID];
```

Figure 4.9 - PlusCal Role Correction

Now that the *Role* key in the SAT object is loaded with all mappings of *UserAssignment* instead of just one element, we can re-run the same TLC model and verify that the user Ray is getting all necessary permissions assigned to them. Re-run output is pictured below in Figure 4.10.

User Output

TLC output generated by evaluating Print and PrintT expressions.

```
[ Role |-> {"Nurse"},
  Permissions |-> {{[GET |-> {"pill_box"}]}},
  CC |-> {},
  CT |-> {},
  User |-> {"Bob"} ]
[ Role |-> {"Doctor", "Admin"},
  Permissions |->
      { {[GET |-> {"pacemaker"}]},
        {[GET |-> {"pill_box"}], [POST |-> {"pill_box"}]} },
  CC |-> {},
  CT |-> {},
  User |-> {"Ray"} ]
```

Figure 4.10 - Corrected TLC Roles Output

The corrections made for this algorithm show how being forced to formally specify and run conceptual tests of a function can reveal gaps in the original pseudocode for many high-level algorithm proposals.

4.3.2 Supporting Context Awareness

Now we need to make sure that the SAT generation algorithm can logically support full CCapBAC capabilities, so it needs to account for device context values and context constraints. We can rework the existing input model to include mappings for these attributes to be eventually loaded into the final SAT object. Mappings of resource to context attributes represent the similarly defined structures in the previous *ContextAware* Z schema. The new input model is shown in Figure 4.11.

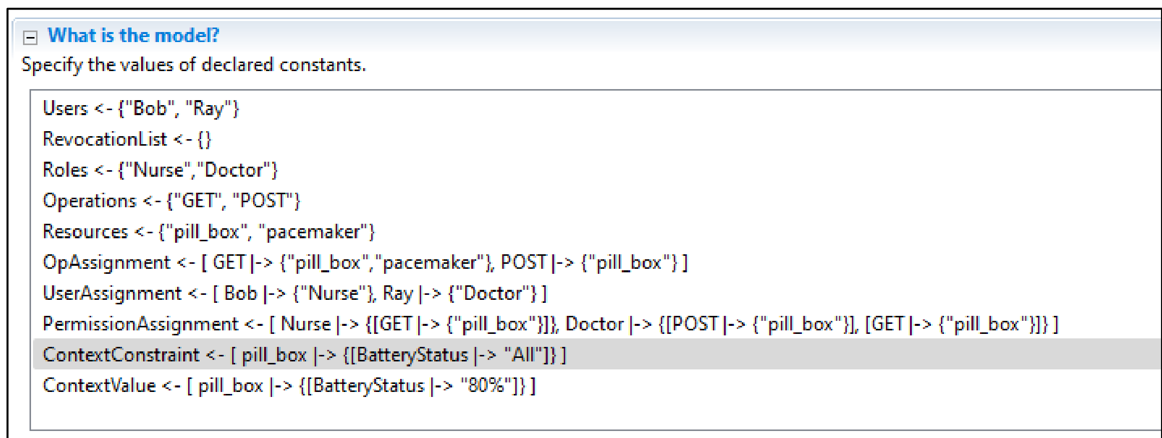

Figure 4.11 - Input Model w/ Context Constraints

From this instance, we have a constraint on the medical pill box that limits its battery status to a certain value. In this case, the constraint would allow all battery statuses to be valid in the serving of a pill box request, but since the defined invariants on the system earlier required all constraints to have matching context values loaded, the SAT object will

still need to load the 80% battery that is the current charge left on the pill box. If we re-run the model checker with the new input range, then we should be expecting to see the "CC" and the "CT" SAT keys loaded with the constraint and value respectively stored in the authorization database. Figure 4.12 shows the model checker output when re-running the algorithm against this input set.

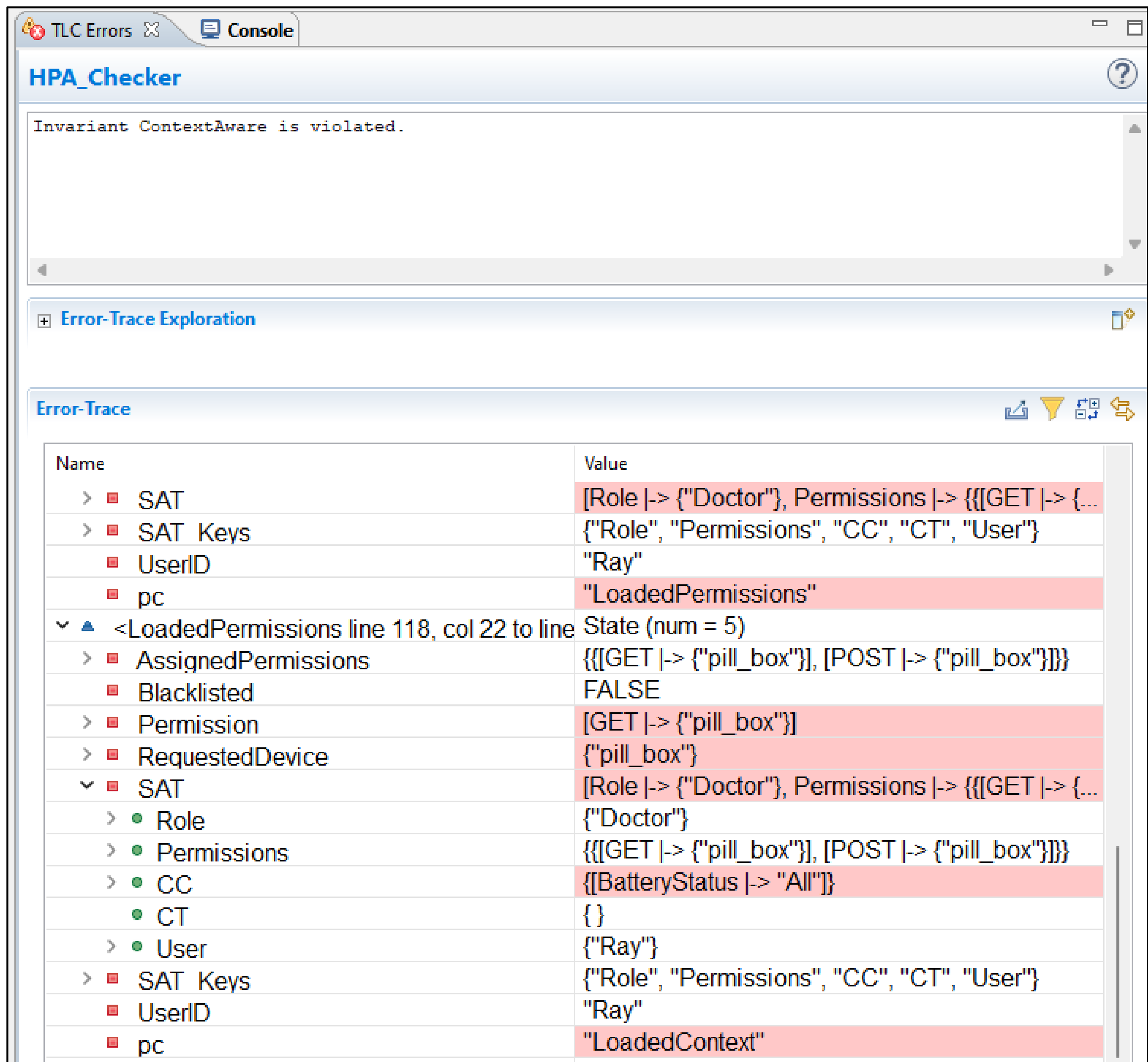

Figure 4.12 - ContextAware Invariant Error Run

The TLC Error Trace displays that the tested instance violated the *ContextAware* invariant that was defined with our original SAT access system invariants. This invariant forces the cardinality of all defined context constraint sets to equal the cardinality of the set of context values loaded into the SAT object. The input model did define a matching context value to the loaded *{BatteryStatus |-> "All"}* constraint that we can see was successfully loaded into the SAT.CC object in Figure 4.12. This implies that there is a flaw in the SAT loading logic itself.

Re-checking the original Algorithm 1 given in the use-case shows that the algorithm only loaded values for *cc* or context constraints and never specified logic to also load the associated context values mapped from the requested resource. Since the spec code mirrored this flaw, the SAT object in the model was also missing the required context values. Fixing this error in the logic is shown in Figure 4.13.

```
56                              SAT["CC"] := ContextConstraint[PullSetElement(RequestedDevice)] ||
57                              SAT["CT"] := ContextValue[PullSetElement(RequestedDevice)];
58                              \*^MISSING CONTEXT VALUE LOAD CORRECTION
```

Figure 4.13 - Context Value Load Fix

With the added logic to load the matching context value to the context constraint for the requested resource, we expect that a re-run of this specification produces an SAT object with full CCapBAC relevant values defined. Figure 4.14 shows the output of the re-run of the TLC model with this fix.

User Output

TLC output generated by evaluating Print and PrintT expressions.

```
[ Role |-> {"Doctor"},
  Permissions |-> {{[GET |-> {"pill_box"}], [POST |-> {"pill_box"}]}},
  CC |-> {[BatteryStatus |-> "All"]},
  CT |-> {[BatteryStatus |-> "80%"]},
  User |-> {"Ray"} ]
[ Role |-> {"Nurse"},
  Permissions |-> {{[GET |-> {"pill_box"}]}},
  CC |-> {[BatteryStatus |-> "All"]},
  CT |-> {[BatteryStatus |-> "80%"]},
  User |-> {"Bob"} ]
```

Figure 4.14 - SAT Output w/ CC & CT

Thus, with the logic fix in place, we have output of a fully verifiable SAT object with both input runs of the model that would represent the JSON object passed on to smart gateway verification with the ACLogic engine in a real-world implementation.

This process of iterating over the security authorization algorithm and defining new ranges of instances to check can be repeated to further improve completeness of the authorization service design while focusing on prioritizing the most prevalent and/or highest repercussion use-cases that could cause the HPA system to not operate securely.

5 DESIGNING FOR CONCURRENCY (PETRI NETS)

Although the proposed framework in the use-case has a very detailed plan for controlling access rights logic efficiently and how the flow of verification and communication work, it does not give design/execution details on addressing the issues of device concurrency. Race conditions that can happen when running medical devices concurrently could mean fatal results in the context of handling patient care with the HPA system. IoT systems especially must account for this since realizing the full potential of an IoT system encourages many devices to run in unison and with frequent network communication. This can make concurrency issues more likely to happen than in the usual technological context and therefore must be accounted for in this security system if it were ever to be fully implemented in a real-world environment.

5.1 U2D Petri Net Designs

Petri Nets offer a standardized specification method to visualize, demonstrate, and solve these concurrency issues. The research example presents a diagram to visualize the communication flow in the case of authorizing a user-to-device (U2D) request within their newly proposed delegated approach of authorization within the IoT system [1]:

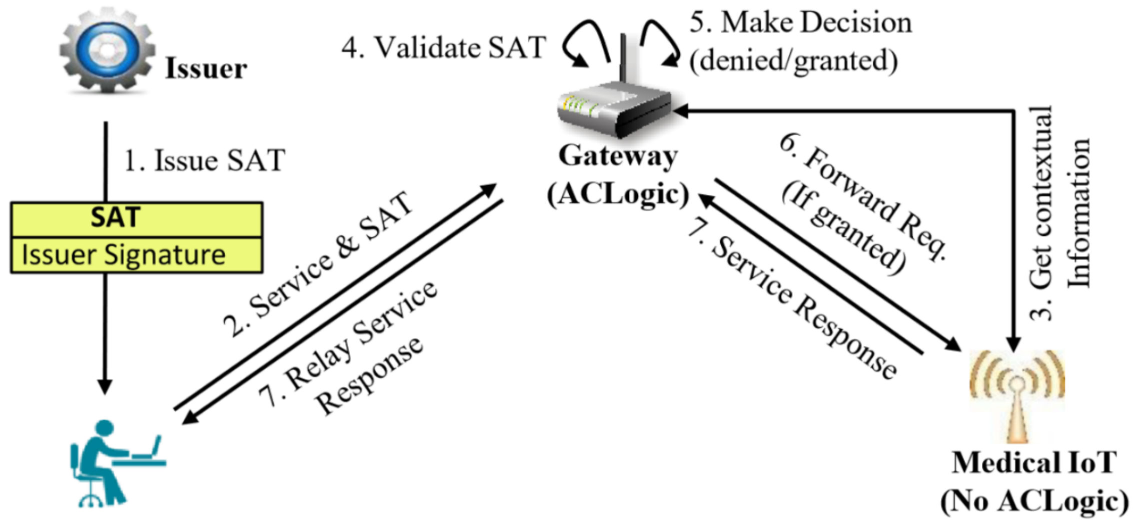

Figure 5.1 - U2D Request & Response Process

From this figure presented in the example research, it is clear that the SAT validation logic is accomplished at the smart gateway. This gateway is local to the network that the requested mIoT device is a part of. Focusing on verification, we can translate this request state flow into the first Petri Net design.

The translation of this diagram into a Petri Net design can be as it is displayed in Figure 5.2.

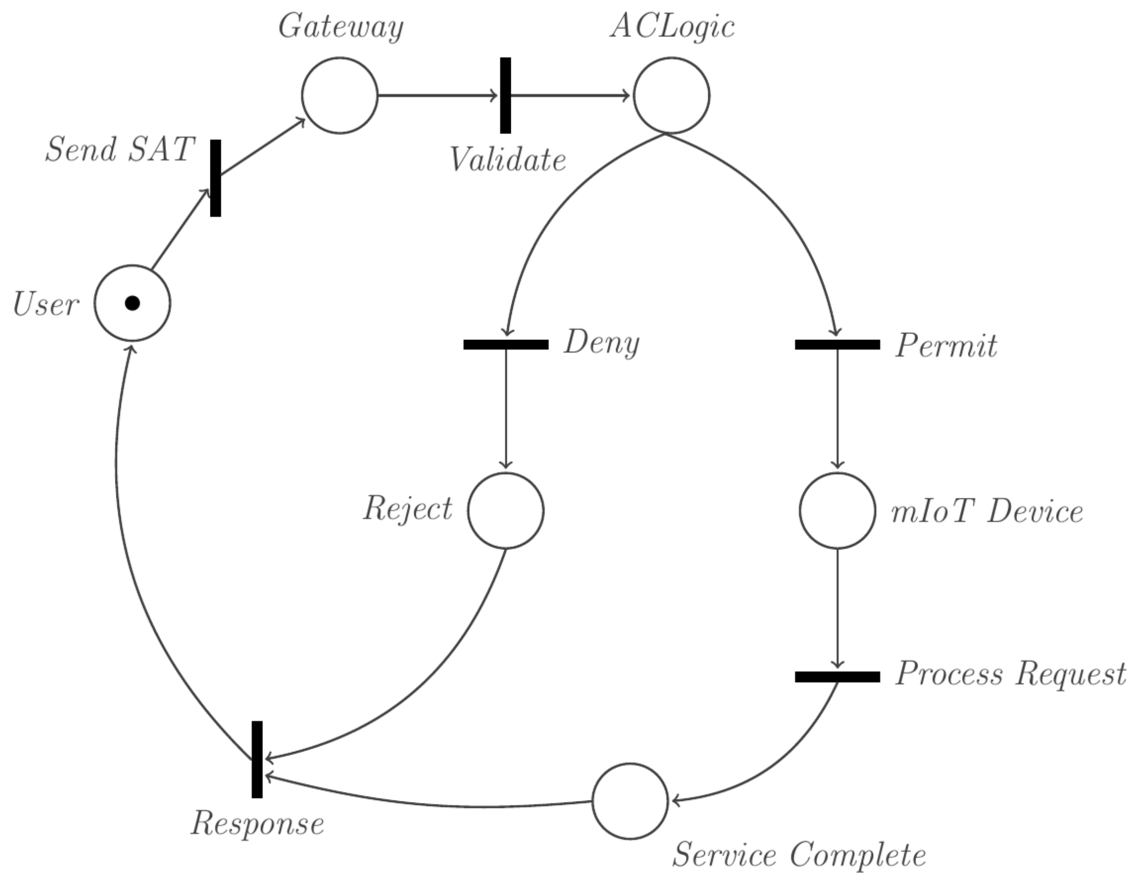

Figure 5.2: U2D Delegated Petri Net

Standardization of this design with a Petri Net provides a clearer visual of the execution flow for the intended process of serving a user request in this system that delegates authorization logic to the smart gateway instead of having it run on the mIoT service device. The flow of user input to response output can be seen from each place in the net through each possible transition state. Each place represents a possible current state of the user's request, including the case where the ACLogic engine denies authorization within the smart gateway of the local network before the request ever gets forwarded to the mIoT device.

However, this current design still does not solve any case of concurrency. One problem that commonly applies to simultaneously communicating devices in a network is when we would want to introduce mutual exclusion. For example, suppose the HPA system was implemented for a patient that needed two implanted mIoT devices to be available at all times so that each device could prescribe their related healthcare function, but we also wanted to always ensure that both of these medical devices did not act at the same time. This could be due to ensuring the effectiveness of each device performing their full function without the interference of the other device executing simultaneously on the same patient. The request process details given in the original research for the example HPA system, or the previously given U2D petri net design could not guarantee against that, as both devices would essentially have their own separate Petri Net to represent their execution.

However, combining the Petri Net design in Figure 5.3 below to account for the simultaneous request for both devices, we can see a possible solution to the case of mutual exclusion. To simplify this second Petri Net design, places and transitions involved in deciding a permit or deny for a request within SAT validation have been removed from the flow of the net, as those states of the execution would not be involved in enforcing mutual exclusion.

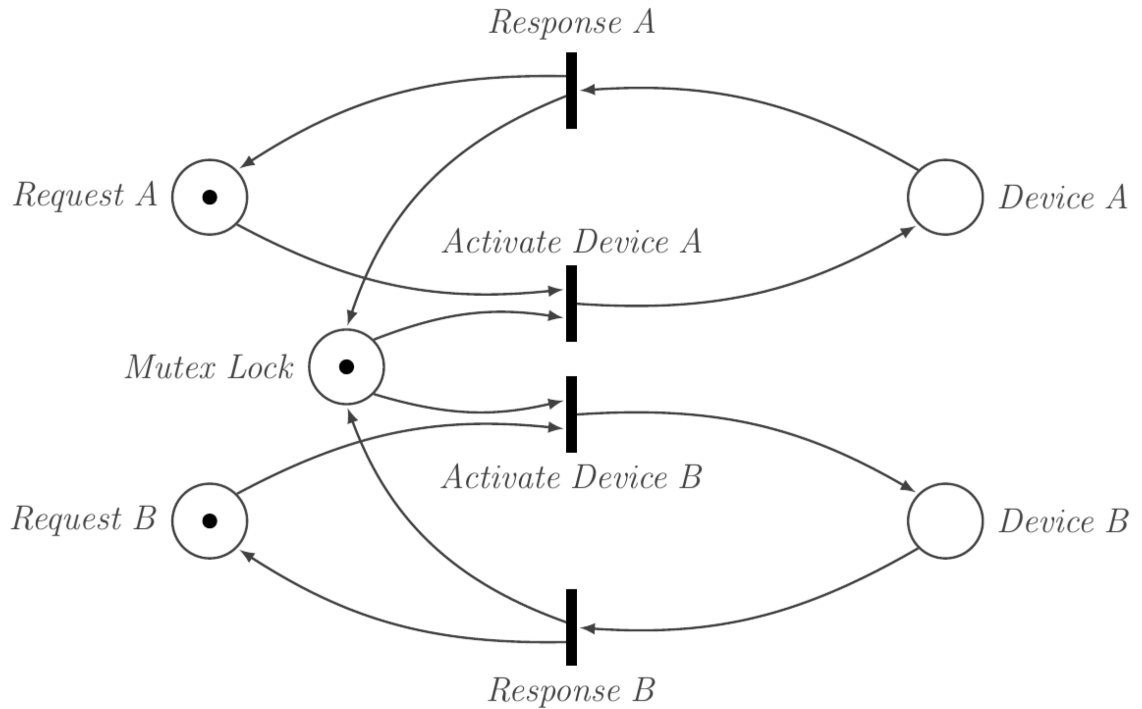

Figure 5.3: Two Devices with Mutex

In this new design, we see the representation of how a U2D request for two devices would be served at the same time, but with a newly added place: *Mutex Lock*. This new place acts as a foolproof guard against having the HPA system serve both requests to both devices at once, since the *Mutex Lock* place must be given a token from the firing of a response transition from either device *A* or *B* before it is able to provide a token for either *Activate Device* transitions to fire.

5.2 Proving State Reachability with Incidence Matrix

We can prove that the assertion that this added state always guarantees against simultaneous execution of these two devices. Petri Nets have a formal design feature of

being able to be translated to an incidence matrix. This matrix, *IM*, shown in Figure 5.4 below, represents all state-to-transition relationships for all cases at any given moment in the system. We have the *columns* representing the transitions in the previous Petri Net, the *rows* representing each place, and the matrix values representing how each transition effects the corresponding place in the amount of net tokens.

$$
\mathbf{IM} = \begin{array}{c} \\ \end{array}
\begin{array}{cccc}
\text{AD_A} & \text{AD_B} & \text{Resp_A} & \text{Resp_B} \\
\end{array}
$$

$$
\mathbf{IM} = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & -1 & 1 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{array}{l} \text{Req_A} \\ \text{Dev_A} \\ \text{Mutex} \\ \text{Dev_B} \\ \text{Req_B} \end{array}
$$

Figure 5.4 - Petri Net Incidence Matrix

We also can represent the initial place values with vector, M_0, to represent a state where both devices are requested with one token in each *Req_A*, *Mutex*, and *Req_B,* and a target state vector, *M*, to represent both devices being activated with 1 token in *Dev_A* and *Dev_B*. This target vector would represent a state that would fail our mutual exclusion requirement, so proving against a possible state, such as both mIoT devices being fired at once, would be equivalent proving that no solution exists for the corresponding matrix equation that include both of these vectors, the incidence matrix, and a solution vector. Therefore, the formula we are trying to verify there is no solution for can be represented in the following in Figure 5.5:

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & -1 & 1 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

$$M = M_0 + IM\vec{v}$$

Figure 5.5 - Mutex Reachability Equation

Solving for the reachability of the target state means finding the solution for the $\vec{v}$ transition vector that would give the amounts of each transition required to reach the target vector in terms of x, y, z, and w. No solution existing for this formula would mean that there are no values of this transition vector that could satisfy this equation and therefore no combination of transitions that would achieve the Petri Net to the M state, or the state where both *Device A* and *Device B* has been activated at once. Using matrix multiplication to write out these equations in a linear system of equations, the proof that no solution exists is equivalent to showing there is no solution to the following linear system:

$$\begin{cases} -x + z = -1 \\ x - z = 1 \\ -x - y + z + w = -1 \\ y - w = 1 \\ -y + w = -1 \end{cases}$$

Through basic algebra, we can see that the first equation is identical to the second equation, and the fourth equation is identical to the fifth equation, so this reduces the system to only three equations. In Figure 5.6, the proof below shows that there is no possible solution which satisfies all equations at once:

$$\begin{bmatrix} x - z = 1 \\ y - w = 1 \\ -x - y + z + w = -1 \end{bmatrix}$$

Proof. No solution exists for the above system:

$$\begin{aligned} x &= z + 1 && \text{(1st system equation)} \\ -1 &= -(z + 1) - y + z + w && \text{(substituting } x \text{ in 3rd equation)} \\ y &= w && \text{(algebraic arithmetic)} \\ w &= w + 1 && \text{(substituting } y \text{ in 2nd equation)} \\ 0 &\neq 1 && \end{aligned}$$

$\square$

Figure 5.6 - Proof for No System Solution

Since the system in Figure 5.6 can be algebraically simplified to a false statement, there is no solution for the system. Therefore, there are no values of the transition matrix that can turn the initial vector state into the target vector state.

Now we know beyond any doubt that the Petri Net design successfully enforces mutual exclusion for these two mIoT devices because there is not a possible set of transitions that can take the net from two simultaneous requests to the simultaneous execution of each device. If the real-world implementation of this security system matched the process flow requirements presented of the mutual exclusion Petri Net in Figure 5.3, then we have guaranteed mutually exclusive activation of the two desired mIoT devices in the final implementation of the system by matching our formal model design.

6 FORMAL DESIGNS ANALYSIS

In this paper we have demonstrated the beginning iterative steps of three different formal methods used to design specific concept areas of the HPA security system presented in the existing research. These formal designs provided corrections and further additions to the service logic that could be translated to fewer bugs and authorization issues in the theoretical final implementation of this mIoT system. Details provided in the example use-case allowed the designs to have a starting point to build unambiguous security requirements off of, but each of the chosen three formal methods offered further design potential that expanded upon the given designs. The analyses in this section review what each design provided in direct translation from the original security framework, what it added or corrected in the designs due to the precise specifications required in each method, and what each formal method could further provide to the HPA system or any other security system if further details were given and further research/testing was conducted.

6.1 Access Control Object Z-Schemas

The Z-schemas that modeled the security components in the example HPA system added clarity to previously ambiguous set definitions and set relations, yet this ambiguity was not clear at first. It also added new system design details to set definitions, set relations, and constraints on the given authorization objects.

The schema designs then went on to specify the set notational details of expected input and output for the primary algorithm in the HPA example for generating SATs. This

schema not only provided a detailed logical flow of where user access comes from within the defined authorization structure, but it also gave a quick and concise visual representation of what the high-level logic of the algorithm would be. This visual representation is something the Z notation accomplishes better than similar set notation methods such as TLA+. Although the TLA+ automated execution of the same algorithm was an easier method to find corrections within the original given design, a security system and the team implementing it in the real-world might find worthwhile value in taking the benefits of using both formal methods simultaneously due to the different specialized benefits from each.

Many more aspects of an example HPA system can be modeled in Z. Schemas to specify *AddUser*, *BlacklistUser*, *VerifySAT*, and *AssignDeviceOperations* are all examples of further specification that would be useful in real implementation and would demonstrate the further design potential of Z. The schemas presented in this thesis outlined all available components and relationships presented in the example use-case, but Z notation shows its effective design representation when modeling aspects of a system throughout state changes, and these designs have not yet been created for the use-case.

6.2 Authorization TLA+ Spec & TLC Model

TLA+ specifications provided the opportunity for automated instance checking. Utilizing this opportunity required detailed set logic of the algorithm to be specified, and the definition of system invariants to force the designer to know exactly what defines a

broken or successful algorithm. It also added details to the SAT generation algorithm itself throughout the process of getting a working TLA+ spec together. Due to these details, misses in the original specifications were able to be concretely found and fixed with verification that the fix worked on the same input models that broke it before, all without needing actual code implementation. This instance checking with the TLC Model Checker provided holes in the generation/verification logic that could have been potentially fatal had it been implemented as is in a real-world doctor-to-patient environment.

TLA+ is a powerful specification language that has high potential of expertise to better model and evaluate a system concept. All set specifications and concurrency proofs accomplished with the other methods could have been fully modeled in TLA+, but it potentially would not have given as much visual clarity to the design.

The system invariants defined in this thesis worked for beginning demonstrations as the algorithm being modeled essentially was transitive set arithmetic. Invariants in general, however, can become much more complex, accurate, and encompassing of what a system needs to do and how it does not need to do it. The invariants even in the use-case can be much further expanded to evaluate sets of *possible* input models instead of having to specify the entire model yourself. Non-deterministic outcomes of serving mIoT requests and properties of liveness of the variable access sets that could be defined can all be added to the HPA example to make it more effective at being secure for every possible outcome.

The TLAPS (TLA+ Proof System) is an additional toolset that works on top of TLA+ specifications and proofs to validate properties of a system. Proofs like the incidence matrix

done with Petri Nets would be a lower-level proof relative to the complexity that these tools can handle modeling.

The potential modeling power of the tool used for formal specification would be related to the complexity of the design being modeled. These tools might not be worth the specification and rigorous notation required if the properties of the sets being modeled can be fully and effectively captured in simpler informal design methods.

6.3 Mutual Exclusion Petri Net Proof

Petri Net specification of a U2D request served in the proposed delegated gateway HPA system allowed for added visual clarity on the possible process flow that could occur with serving an mIoT device request. The original HPA example did provide details on the order of execution of a request after the SAT has been generated. However, the original given designs were semi-convoluted since they had to include details of public/private key verification and physical device mnemonics that were not relevant to the actual verification of the SAT. The simple Petri Net design provided in this paper abstracted away all of those details, so the designer would be able to see the relevant aspects of the system that would dictate successful SAT verification.

The improved designs also allowed for the ability to address possibly concurrency issues with the system and presented a solution for the possible requirement of mutual exclusion. This solution was also able to be mathematically proven to be correct through solving an

incidence matrix vector representation of the system for all possible states of the request diagram.

Petri Nets excel in concurrency-related design contexts, and these designs could be expanded for more situations that would arise in the actual implementation of an mIoT HPA system. Larger nets with more challenging properties could be modeled under these designs to visualize and solve more complicated request flow between mIoT devices in the HPA system. For example, with a mutex lock implemented in the proposed designs in this thesis you could add to the design and then have to solve any potential problems of deadlock and/or resource sharing. Petri Nets excel in visualizing these issues compared to most modeling methods, and the formality of their structure still allows concrete properties to be proven the values in these designs.

7 CONCLUSIONS

The goal of creating secure application software is usually seen as primarily an implementation challenge with less stress on solving security issues at the design phase. Through the use of formal specification methods more often in the security environment, we can help ourselves improve the core security concepts about a system before coding even begins. Although the formal modeling process is rigorous and requires a relatively high mathematical learning curve to fully realize the return on investment, the demonstrations of three separate specification methods provided in this thesis showed the potential on what those benefits can be for improving clarity, pre-emptively solving execution flaws, and providing provably correct additions to a security system that is intended to function in a high-risk environment such as patient healthcare. As systems become more complex and vulnerabilities have higher consequences in an increasingly technological world, re-emphasizing the potential to find security solutions at the design phase could cause a worthwhile improvement in the safety and privacy of the final released product before the real-world consequences even have to occur.

REFERENCES

1. Mahmud Hossain et al., “An Internet of Things-Based Health Prescription Assistant and Its Security System Design,” *Future Generation Computer Systems*, vol. 82, pp. 422-439, May 2018, doi: 10.1016/j.future.2017.11.020.

2. Vikas Hassija et al., “A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures,” *IEEE Access*, vol. 7, pp. 82721-82743, 2019, doi: 10.1109/ACCESS.2019.2924045.

3. Chris Newcombe et al., “How Amazon Web Services Uses Formal Methods,” *Communications of the ACM*, vol. 58, pp. 66-73, April 2015, doi: 10.1145/2699417.

4. Ashish Darbari. *A Brief History of Formal Verification*, 2019. Accessed: Nov. 1, 2023. [Online]. Available: https://www.eeweb.com/a-brief-history-of-formal-verification/

5. University of Koblenz-Landau. (2006). Formal Verification of Software. [Online]. Available: https://formal.kastel.kit.edu/~beckert/teaching/Verification-SS06/01intro.pdf

6. R. Presson. “TLA+.” pron.gtihub.io. Accessed: May 10, 2023. [Online.] Available: https://pron.github.io/posts/tlaplus_part1#tla-in-practice

7. S. Raju, K. Rytarowski. *An introduction to Formal Verification for Software Systems*, 2020. Accessed: July 3, 2022. [Online]. Available: https://www.moritz.systems/blog/an-introduction-to-formal-verification/

8. J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof.* University of Oxford, England. 1996.

9. J. Bowen, “The Z Notation: Whence the Cause and Whither the Course?” *Engineering Trustworthy Software Systems*, vol. 9506, pp. 103-151, 2016, doi: 10.1007/978-3-319-29628-9_3.

10. J. M. Spivey, “The Z Notation: A Reference Manual,” *Programming Research Group*, University of Oxford, England. 1989.

11. Mike Spivey, *A Guide to the zed Style Option.* Westlands Grove, Stockton Lane, York, England, December 1990.

12. Leo Freitas, “Standard Z-LaTeX style explained,” *Community Z Tools (CZT)*, University of York, United Kingdom. September 2008.

13. Jonathan Jacky et. al., “Formal Specification of Control Software for a Radiation Therapy Machine,” Radiation Oncology Department RC-08, University of Washington, Seattle, WA, January 9, 1997.

14. Emmanuel Tuyishimire, Antoine Bigomokero Bagula, “A Formal and Efficient Routing Model for Persistent Traffics in the Internet of Things,” in *Int. Conf. on Information Comm. Tech. and Society*, March 2020. doi: 10.1109/ICTAS47918.2020.234002.

15. Leslie Lamport, *Learning TLA+*, 2021. Accessed: Sept. 22, 2022. [Online]. Available: http://lamport.azurewebsites.net/tla/learning.html

16. Leslie Lamport, “Specifying Systems,” *The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Microsoft Research, July 2002.

17. Hillel Wayne, *Learn TLA+*, 2022. Accessed: May 4, 2022. [Online]. Available: https://learntla.com/index.html.

18. Leslie Lamport, *A PlusCal User’s Manual,* P-Syntax Version 1.8. March 11, 2024. Accessed: March 25, 2024. [Online]. Available: https://lamport.azurewebsites.net/tla/p-manual.pdf

19. Rober A. McGuigan, “Petri Nets,” *Applications of Discrete Mathematics*, Department of Mathematics, Westfield State College, ch. 24, pp. 431-453.

20. Tadao Murata, “Petri Nets: Properties, Analysis and Applications,” *Proceedings of the IEEE*, v. 77.4, 1989, pp. 541-580.

21. Benmiloud Mohammed, “It’s Time to Learn drawing Petri Nets in TikZ,” February 21, 2021. Accessed: March 26, 2023. Available: https://latexdraw.com/petri-nets-tikz/

APPENDIX

A. TLA+ Specification: *HPA.tla*

HPA | HPA_Checker

```
1  ------------------------------- MODULE HPA -------------------------------
2  EXTENDS Integers, Sequences, TLC, FiniteSets
3  CONSTANTS Users, RevocationList
4  CONSTANTS Roles
5  CONSTANTS Operations, Resources
6  CONSTANTS UserAssignment, OpAssignment
7  CONSTANTS PermissionAssignment
8  CONSTANTS ContextConstraint, ContextValue
9  ASSUME Cardinality(Users) >= 1
10
11 (* --algorithm SAT_Generation
12 variables
13     SAT_Keys = {"User", "Role", "Permissions", "CC", "CT"};
14     UserID \in Users;
15     Blacklisted = FALSE;
16     AssignedPermissions = {};
17     Permission = {};
18     RequestedDevice = {};
19     SAT = [key \in SAT_Keys |-> {}];
20
21
22 define
23     NotRevoked == Blacklisted = FALSE
24     RoleAssigned == pc = "LoadedRole" => Cardinality(SAT["Role"]) > 0
25     PermissionAssigned == pc = "LoadedPermissions" => Cardinality(SAT["Permissions"]) > 0
26     ContextAware == pc = "LoadedContext" => (Cardinality(SAT["CC"]) > 0 =>
27                                          Cardinality(SAT["CC"]) = Cardinality(SAT["CT"]))
28     PullSetElement(set) == CHOOSE x \in set:TRUE
29     GetDevice(perm) == {perm[key]: key \in DOMAIN perm}
30
31 end define;
32
33 begin
34     CheckBlacklisted:
35         if UserID \in RevocationList then
36             Blacklisted := TRUE;
37             SAT := {};
38         else
39             SAT["User"] := SAT["User"] \union {UserID};
40             LoadedUser:
41                 \*SAT["Role"] := SAT["Role"] \union {CHOOSE x \in Roles: x \in UserAssignment[UserID]};
42                 \*MULTIPLE ROLES ALGORITHM CORRECTION
43                 SAT["Role"] := SAT["Role"] \union UserAssignment[UserID];
44                 LoadedRole:
45                     AssignedPermissions := {PermissionAssignment[x]: x \in SAT["Role"]};
46                     SAT["Permissions"] := SAT["Permissions"] \union AssignedPermissions;
47                     LoadedPermissions:
48                         Permission := PullSetElement(PullSetElement(AssignedPermissions));
49                         RequestedDevice := PullSetElement(GetDevice(Permission));
50                         SAT["CC"] := ContextConstraint[PullSetElement(RequestedDevice)] ||
51                         SAT["CT"] := ContextValue[PullSetElement(RequestedDevice)];
52                         \*^MISSING CONTEXT VALUE LOAD CORRECTION
53                         LoadedContext:
54             print SAT;
55         end if;
56
57 end algorithm *)
58 \* BEGIN TRANSLATION (chksum(pcal) = "95bc59ac" /\ chksum(tla) = "e3ef0427")
59 VARIABLES SAT_Keys, UserID, Blacklisted, AssignedPermissions, Permission,
60           RequestedDevice, SAT, pc
61
```

```
57  end algorithm *)
58  \* BEGIN TRANSLATION (chksum(pcal) = "95bc59ac" /\ chksum(tla) = "e3ef0427")
59  VARIABLES SAT_Keys, UserID, Blacklisted, AssignedPermissions, Permission,
60            RequestedDevice, SAT, pc
61
62  (* define statement *)
63  NotRevoked == Blacklisted = FALSE
64  RoleAssigned == pc = "LoadedRole" => Cardinality(SAT["Role"]) > 0
65  PermissionAssigned == pc = "LoadedPermissions" => Cardinality(SAT["Permissions"]) > 0
66  ContextAware == pc = "LoadedContext" => (Cardinality(SAT["CC"]) > 0 =>
67                                          Cardinality(SAT["CC"]) = Cardinality(SAT["CT"]))
68  PullSetElement(set) == CHOOSE x \in set:TRUE
69  GetDevice(perm) == {perm[key]: key \in DOMAIN perm}
70
71
72  vars == << SAT_Keys, UserID, Blacklisted, AssignedPermissions, Permission,
73             RequestedDevice, SAT, pc >>
74
75  Init == (* Global variables *)
76          /\ SAT_Keys = {"User", "Role", "Permissions", "CC", "CT"}
77          /\ UserID \in Users
78          /\ Blacklisted = FALSE
79          /\ AssignedPermissions = {}
80          /\ Permission = {}
81          /\ RequestedDevice = {}
82          /\ SAT = [key \in SAT_Keys |-> {}]
83          /\ pc = "CheckBlacklisted"
84
85  CheckBlacklisted == /\ pc = "CheckBlacklisted"
86                      /\ IF UserID \in RevocationList
87                            THEN /\ Blacklisted' = TRUE
88                                 /\ SAT' = {}
89                                 /\ pc' = "Done"
90                            ELSE /\ SAT' = [SAT EXCEPT !["User"] = SAT["User"] \union {UserID}]
91                                 /\ pc' = "LoadedUser"
92                                 /\ UNCHANGED Blacklisted
93                      /\ UNCHANGED << SAT_Keys, UserID, AssignedPermissions,
94                                      Permission, RequestedDevice >>
95
96  LoadedUser == /\ pc = "LoadedUser"
97                /\ SAT' = [SAT EXCEPT !["Role"] = SAT["Role"] \union UserAssignment[UserID]]
98                /\ pc' = "LoadedRole"
99                /\ UNCHANGED << SAT_Keys, UserID, Blacklisted,
100                               AssignedPermissions, Permission, RequestedDevice >>
101
102 LoadedRole == /\ pc = "LoadedRole"
103               /\ AssignedPermissions' = {PermissionAssignment[x]: x \in SAT["Role"]}
104               /\ SAT' = [SAT EXCEPT !["Permissions"] = SAT["Permissions"] \union AssignedPermissions']
105               /\ pc' = "LoadedPermissions"
106               /\ UNCHANGED << SAT_Keys, UserID, Blacklisted, Permission,
107                               RequestedDevice >>
108
109 LoadedPermissions == /\ pc = "LoadedPermissions"
110                      /\ Permission' = PullSetElement(PullSetElement(AssignedPermissions))
111                      /\ RequestedDevice' = PullSetElement(GetDevice(Permission'))
112                      /\ SAT' = [SAT EXCEPT !["CC"] = ContextConstraint[PullSetElement(RequestedDevice')],
113                                            !["CT"] = ContextValue[PullSetElement(RequestedDevice')]]
114                      /\ pc' = "LoadedContext"
115                      /\ UNCHANGED << SAT_Keys, UserID, Blacklisted,
116                                      AssignedPermissions >>
```

```
116                                     AssignedPermissions >>
117
118 LoadedContext == /\ pc = "LoadedContext"
119                  /\ PrintT(SAT)
120                  /\ pc' = "Done"
121                  /\ UNCHANGED << SAT_Keys, UserID, Blacklisted,
122                                  AssignedPermissions, Permission,
123                                  RequestedDevice, SAT >>
124
125 (* Allow infinite stuttering to prevent deadlock on termination. *)
126 Terminating == pc = "Done" /\ UNCHANGED vars
127
128 Next == CheckBlacklisted \/ LoadedUser \/ LoadedRole \/ LoadedPermissions
129            \/ LoadedContext
130            \/ Terminating
131
132 Spec == Init /\ [][Next]_vars
133
134 Termination == <>(pc = "Done")
135
136 \* END TRANSLATION
137
138
139 ============================================================================
```

VITA

After graduating high school from The Colony High School, in The Colony, Texas, in 2013, Thomas Rolando Mellema entered Stephen F. Austin State University at Nacogdoches, Texas. He received the degree of Bachelor of Science from Stephen F. Austin State University in May 2018 with a double major in Computer Science and Mathematics. During the following six years, he was employed as a software developer at Elliott Electric Supply and NacSpace. In August 2018, he entered the Graduate School of Stephen F. Austin State University and received the degree of Master of Science in May 2024 with a major in Cybersecurity.

Permanent Address: Cole STEM Building, 1720 Clark Blvd
Nacogdoches, TX 75965

The style manual used in this thesis is the *IEEE Editorial Style Manual for Authors (V.11.12.18)*.

This thesis was typed by Thomas R. Mellema.