University of Southampton

University of Southampton Research Repository

Copyright © and Moral Rights for this thesis and, where applicable, any accompanying data are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis and the accompanying data cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content of the thesis and accompanying research data (where applicable) must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holder/s.

When referring to this thesis and any accompanying data, full bibliographic details must be given, e.g.

Thesis: Mehmet Said Nur Yagmahan (2024) "Generic Formal Patterns for Cloud Native Application Development", University of Southampton, School of Electronics and Computer Science, PhD Thesis, pages 1-121.

Data: Mehmet Said Nur Yagmahan (2024) "Generic Formal Patterns for Cloud Native Application Development".

University of Southampton

Generic Formal Patterns for Cloud Native Application Development

by

Mehmet Said Nur Yagmahan

ORCiD: 0000-0003-4981-4286

A thesis for the degree of Doctor of Philosophy

in the

Faculty of Engineering and Physical Science
School of Electronics and Computer Science

July 2024

Supervisors: Abdolbaghi Rezazadeh, Michael Butler

University of Southampton

Abstract

Faculty of Engineering and Physical Science
School of Electronics and Computer Science

Doctor of Philosophy

Generic Formal Patterns for Cloud Native Application Development

by Mehmet Said Nur Yagmahan

With advances in cloud computing and distributed systems, cloud-native applications provide immense flexibility to developers in terms of building scaleable and efficient applications and systems. One of the predominant architectures that epitomises this shift in modern application development is the service-oriented architecture (SOA). While SOA offers developers significant flexibility during system development, it inadvertently increases overall system complexity, which may result in design and implementation flaws.

To deal with complexity, formal methods offer abstraction. By conceptualising systems at higher abstraction levels, they help developers and system architects achieve a better grasp of the system's entirety and its nuances. Moreover, in service-oriented architecture and cloud-native applications, access control is a crucial component because it serves as the gatekeeper, specifying who can access the system or use which resources or services. Therefore, it should be designed robustly to protect resources and ensure the application's security.

In this research, we mainly focus on developing formal modelling patterns to assist cloud-native application developers in securely designing their cloud-native systems. Therefore, firstly, we will develop a set of formal modelling patterns for the functionalities of serverless systems in the Event-B environment. In the next stage, we incorporate an access control mechanism for the serverless system into our previously proposed patterns. Then, to illustrate the usefulness of our patterns and approach, we model two distinct scenarios of a project management application with serverless architecture. We conclude by summarising our findings and highlighting the research's prospective directions and potential applications.

Contents

List of Figures **ix**

List of Tables **xiii**

Declaration of Authorship **xv**

Acknowledgements **xvii**

Abbreviations **xxi**

1 Introduction **1**
- 1.1 Thesis Organisation 4

2 Web Services **5**
- 2.1 A Brief History of Cloud Computing 5
- 2.2 Microservices 7
- 2.3 Web Services 8
- 2.4 Serverless Technology 9
  - 2.4.1 Advantages of Serverless Architecture 10
  - 2.4.2 Disadvantages of Serverless Architecture 11
- 2.5 Cloud Platforms Offering Serverless 12
  - 2.5.1 Microsoft Azure 12
  - 2.5.2 Google Cloud 13
  - 2.5.3 Amazon Web Services 13
- 2.6 Security and Access Controls in AWS Environments 14
  - 2.6.1 AWS-Based Serverless Application Architecture 14
    - 2.6.1.1 Amazon Cognito 15
    - 2.6.1.2 AWS API Gateway 15
    - 2.6.1.3 AWS Lambda 16
    - 2.6.1.4 AWS IAM 16
  - 2.6.2 Authorization and Access Control in AWS 17
- 2.7 Conclusion 20

3 Formal Methods **21**
- 3.1 Formal Specification 22
- 3.2 Formal Specification Languages 22
  - 3.2.1 Model-based Specification Languages 22
  - 3.2.2 Algebraic Specification Languages 22

3.2.3 Formal Specification Languages used for Cloud Systems 23
3.2.4 TLA+ 23
3.2.5 Event-Calculus 23
3.2.6 Event-B 24
3.2.6.1 The Structure and Syntax 24
3.2.6.2 Refinement 25
3.2.6.3 Proof Obligation 26
3.2.6.4 Decomposition 26
3.2.6.5 Event-B Tool: Rodin 27
3.2.6.6 ProB (Animator and Model Checker for Event-B) 27
3.3 Related Work 28
3.3.1 Formal Methods in Web Services 28
3.3.1.1 Security of Cloud 29
3.3.1.2 Security of Customers in The Cloud 29
3.3.2 Web Service Composition 29
3.3.3 Access Control 30
3.3.4 Other Relevant Works 30
3.4 Conclusion 30

4 Formal Patterns for Serverless App 31
4.1 Request Handling Pattern (RHP) 32
4.2 Request Order Patterns 34
4.2.1 Single Service Request (SSR) 34
4.2.1.1 Representation RHP and SSR Patterns in Event-B 35
4.2.2 Linear Service Request (LSR) 38
4.2.2.1 Representation LSR Pattern in Event-B 39
4.2.3 Branching Service Request (BSR) 42
4.2.3.1 Representation BSR Pattern in Event-B 44
4.2.4 Chained Branching Service Request (CBSR) 46
4.2.4.1 Representation CBSR Pattern in Event-B 47
4.2.5 Comparison of ROP Patterns 50
4.3 POs of RHP / ROP Patterns 51

5 Formal Patterns for Authorization Mechanism 53
5.1 Formal Patterns for Authorisation Mechanism 54
5.1.1 A Non-Deterministic Authorization Mechanism 54
5.1.2 First Refinement: Introducing Sub-typing Generalization 55
5.1.3 Second Refinement: Replacing the Non-deterministic Authorizer with a Deterministic Authorizer 57
5.1.4 Third Refinement: The complete Access Control in AWS 61
5.2 POs of Authorization Mechanism Model 62
5.3 Conclusion 63

6 Case Studies 65
6.1 Case Study: Project Management System 65
6.1.1 Scenario 1: Updating Project Status 68
6.1.1.1 Model "updating project status" Functionality 69

6.1.1.2 Event-B Model of The Scenario 71
6.1.2 Scenario 2: "Promoting a User as Project Manager" 84
6.1.2.1 Model "Promoting a User as Project Manager" Functionality 86
6.1.2.2 Event-B model of Scenario 2 88
6.2 Case Study: Learning Management System 92
6.2.1 Scenario 3: Updating Project Status 95
6.2.1.1 Model ”Hand in an Assignment” Functionality 96
6.2.1.2 Event-B Model The Scenario 98
6.3 POs of Case Study Scenarios’ Models 105
6.4 Conclusion 106

7 Conclusion **107**
7.1 Contributions 108
7.2 Future Works 111

References **113**

List of Figures

2.1 The concept of cloud computing, adapted from [42] 6
2.2 Hypervisor and Container, adapted from [14] 7
2.3 The patterns of Web Services Collaboration 8
2.4 A general Structure of Serverless Concept, adapted from [115] 9
2.5 Structure of Application with Serverless Architecture 10
2.6 A Basic Structure of a Web App with Serverless Architecture in AWS Environment 14
2.7 The structure of a serverless application 16
2.8 The structure of an AWS IAM Policy 18
2.9 An AWS IAM Policy Example 18
2.10 The Relation between IAM Identities and Policies 19

3.1 Roughly the mechanism in Event-Calculus, adapted from [99] 24
3.2 The structure of an Event-B model 25
3.3 The follow of refinements in event-b Model 25
3.4 A view of ProB Model Checker 28

4.1 Access to Resources 31
4.2 A Generic Pattern for Request Handling in a Cloud-Native Apps 32
4.3 Single Service Request Pattern 35
4.4 Entities for Request Handling Pattern (Context01) 35
4.5 Event-B model of Request Handling Pattern 36
4.6 The structure of multiple requests which have LSR pattern 38
4.7 The tree-like diagram of a LSR pattern 38
4.8 The Event-B events in a LSR pattern 41
4.9 The structure of multiple requests which have BSR pattern 42
4.10 The Tree-like diagram of BSR Pattern 43
4.11 Response Events of subsequent request in BSR Pattern 44
4.12 Initiate Request to Service3 Events in BSR Pattern 45
4.13 The structure of multiple requests which have CBSR pattern 46
4.14 The Tree-like diagram of CBSR Pattern 46
4.15 The relation between chain number and branching 48
4.16 Guards and Actions About Chain in Initiate Request Events 48
4.17 Updating The Status of a Chain when a Request fails 49
4.18 To Represent $<>$ Condition in Event-B Model 49
4.19 Invariants in RHP and ROP Pattern Model 50
4.20 POs of RHP and ROP Patterns 51

5.1 A Non-Deterministic Authorization Mechanism 54
5.2 Resource and Resource Types 55
5.3 Refining Request Execution Based on Resource Types 56
5.4 An abstraction of permission mechanism in AWS environment 57
5.5 Event-B Invariants Representing Abstract Permission Mechanism 57
5.6 Authorization of a request 58
5.7 Event-B Encoding of the Request Authorization 59
5.8 Deterministic Authorizer Events 60
5.9 Introducing IAM user, IAM role and IAM policy 61
5.10 The Effect of New Features on Authorization Events 61
5.11 POs of Authorization Mechanism's Model 62

6.1 A Basic Structure of a Web App with Serverless Architecture in AWS Environment 67
6.2 The Structure of "Updating Project Status" Functionality 68
6.3 The Authorisation entities in "Updating Project Status" Functionality 69
6.4 The tree-like representation of "updating project status" functionality 70
6.5 Features Specific to AWS system 72
6.6 Features Specific to Case Study System and Scenario 72
6.7 Invariants of Abstraction for Scenario 1 73
6.8 Initialisation for Scenario 1 Abstraction 74
6.9 Request Initiation Events for Scenario 1 Abstraction 74
6.10 Some Events in Scenario 1 Abstraction 75
6.11 Local Action Event Project DB Table 76
6.12 Permission Entities for Scenario 1 77
6.13 Ref 1: Initialisation Event for Scenario 1 77
6.14 Ref 1: Visual of the Required Permissions for Fulfilment of "updating project status" functionality 78
6.15 Refining Non-Deterministic Authorisation into Deterministic for the *EPUpdProSt* endpoint 79
6.16 Refining Non-Deterministic Authorisation into Deterministic for the *FunUpdProData Function* 80
6.17 Application specific features: Roles, policies 81
6.18 Invariants Introducing Roles, policies 81
6.19 Ref 2: Initialisation Event for Case Study Scenario 1 82
6.20 Ref 2: Authorisation Verification events Permit Cases 83
6.21 The Structure of "Promoting a User as Project Manager" Functionality 84
6.22 The Authorisation entities in "Promoting a User as Project Manager" Functionality 85
6.23 The tree-like representation of ”Promoting a User as Project Manager” functionality 86
6.24 The Features Related to Scenario 2 Configuration 88
6.25 Abstract Invariants for Scenario 2 89
6.26 Initialisation for Scenario 2 Abstraction 90
6.27 Request Initiation Events for Scenario 2 Abstraction 90
6.28 Local Action Events for Scenario 2 Abstraction 91
6.29 Initialisation Events for Scenario 2 in Refinement Steps 92

6.30 A Basic Structure of Learning Management System 94
6.31 The Structure of "Hand in an Assignment" Functionality 95
6.32 The Fulfilling of "Hand in an Assignment" Functionality 96
6.33 The tree-like representation of ”Hand in an Assignment” functionality 96
6.34 Features Specific to the Case Study System and the Scenario 98
6.35 Invariants for "Hand in An Assignment" Scenario (Abstract Machine) 99
6.36 Initialisation Event in Abstraction Machine 100
6.37 Data Transition during Among Requests 101
6.38 Permission Entities for "Hand in An Assignment" Scenario 102
6.39 Ref 1: Initialisation Event for "Hand in An Assignment" Scenario 103
6.40 The required Role and Policy entities in Context 104
6.41 Ref 1: Initialisation Event for "Hand in An Assignment" Scenario 104
6.42 POs of The First Case Study Scenario Model 105
6.43 POs of The Second Case Study Scenario Model 105
6.44 POs of The Third Case Study Scenario Model 106

List of Tables

6.1 General Requirements for Learning Management System 66
6.2 Requirements of Admins 66
6.3 Requirements of Department Managers 66
6.4 Requirements of Project Managers 66
6.5 Requirements of Developers 66
6.6 Requests During the Process of Scenario 1 71
6.7 Requests During The Process of Scenario 2 87
6.8 General Requirements for Learning Management System 93
6.9 Requirements of Admins 93
6.10 Requirements of Head of School 93
6.11 Requirements Lecturers 93
6.12 Requirements of Students 94
6.13 Requests During The Process of ”Hand in an Assignment” Scenario 97

Declaration of Authorship

I , Mehmet Said Nur Yagmahan , declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as:

Signed:.. Date:..................

Acknowledgements

I would like to express my deep gratitude to people who have played a pivotal role in the successful completion of my doctoral thesis. The following acknowledgements are dedicated to those whose significant contributions have shaped this academic journey.

First and foremost, my deepest gratitude goes to my parents, Mehmet Zeki Yagmahan and Nezahat Yagmahan. They have consistently illuminated my path, much like a lighthouse guiding a ship to a safe harbor. Their unwavering support and encouragement have been the cornerstone of my academic pursuits.

I would also like to express my sincere thanks to my esteemed supervisors, Dr. Abdolbaghi Rezazadeh and Prof. Michael Butler. Their invaluable contributions, support, and motivational guidance have been instrumental in my commitment to rigorous study and the ultimate completion of my research. Their insightful suggestions have profoundly influenced the development of my project, leading to a more robust and refined outcome.

In addition, I extend my gratitude to each member of my family and my friends, whose moral support has been a constant source of strength throughout my academic journey.

Finally, I dedicate a special acknowledgement to my country, Turkey, more specifically the Ministry of National Education, which provided financial support to me during my academic research.

To Meryem Yagmahan

Bêyî dîtineke dawî,
Tu ji çûyî pira min ku direje bîranînên min ê zarokatiye dibû
Cihê te bi Seyda Mela Mehmûd re Bihuşta Nûranî be... [1]

[1] EN: Without a last seen
You, that is my bridge that goes to my childhood memories, left as well
May you are at Holly Heaven with Seyda Mela Mehmûd ...

Abbreviations

AC	Access Control
API	Application Programming Interface
AWS	Amazon Web Services
BSR	Branching Service Request
CBSR	Chained Branching Service Request
IAM	Identity Access Management
LSR	Linear Service Request
POs	Proof Obligations
RHP	Request Handling Pattern
ROP	Request Ordering Pattern
SSR	Single Service Request
TLA	Temporal Logic of Action
URL	Uniform Resource Locator
VDM	Vienna Development Method

Chapter 1

Introduction

Cloud computing has transcended the traditional paradigms of data and resource sharing, evolving into a robust platform for developing and deploying new types of distributed applications, called cloud-native applications. Cloud-native apps use new architectural models such as serverless, also known as FaaS (Function as a Service).

The serverless architectural paradigm provides access to autonomous, executable fragments of code, termed serverless functions. These self-contained code units are engineered to be invoked and executed without worrying about server-side management operations for a limited execution time. In a serverless application, the execution of a serverless function may satisfy a single or multiple task(s).For instance, a task could be writing to a specific table on a database. For more complicated functionalities, serverless functions may connect with multiple resources and services provided by the cloud platform.

In the context of serverless architectures, the functionalities of a cloud-native application are decomposed into discrete, stateless functions. These functions may interact with various cloud platform resources and services to fulfill their task. Therefore, functions act as a glue between cloud platform resources and services to form the backbone of an application's backend, embodying the essence of the serverless paradigm [26].

When different access levels for users (authorization) are required in a system, access control mechanisms come into the picture. Access control is a sophisticated mechanism that governs permissions regarding the access, utilisation, and management of data, resources [15], and functions. Therefore, an access control mechanism is the heart of cloud-native systems, specifically applications with serverless architecture, in terms of satisfying a functionality. Any conflict or inconsistency in the access control mechanism may cause malfunctions or considerable losses in a cloud-native system.

To avoid those issues, many cloud providers, like AWS and Microsoft Azure, have adopted a *shared responsibility model* [39, 105, 13]. This collaborative approach divides the tasks of maintaining security between cloud providers and cloud users. According to Amazon documentation [97], there are two key domains in *the shared responsibility model*, which are *"security of the cloud"* and *"security in the cloud"*.

Security of the Cloud: This domain defines the responsibility of the cloud providers. The cloud providers are responsible for keeping the cloud infrastructure secure. The responsibility includes safeguarding the fundamental physical and virtual infrastructure, ensuring proper isolation between customers, and maintaining the integrity of the core services.

Security in the Cloud: On the other hand, the *"security in the cloud"* domain shows what tasks and obligations are under the responsibility of cloud users. This responsibility covers the configuration of access levels and the overall management of who (or what) can access specific resources in the cloud environment. To do that, cloud providers offer some services and tools for cloud customers to manage and configure access to resources and services in their cloud accounts.

Various cloud providers provide different access control mechanisms that are specific to their cloud environments. For instance, Identity and Access Management (IAM) and Cognito services help to configure granular access control in the Amazon Web Services (AWS) environment, while Microsoft's Azure platform relies on Azure Active Directory to achieve similar ends.

Moreover, using cloud-native concepts provides a huge set of opportunities to develop more efficient and scalable applications compared with those developed in traditional ways in terms of resource usage. Therefore, cloud computing concepts, especially the serverless concept, have become popular. For example, according to a report by Right Scale in 2019 [36], 93% of enterprises use cloud services. The report also stated that serverless is the most common cloud concept among customers, with a 75% growth rate [36].

However, in the meantime, it poses its own challenges, such as those that arise from the complexity of service or function composition and the configuration of access control mechanisms. These challenges often expose defects in the development and configuration of the application itself. For instance, according to a report published by the Carnegie-Mellon Software Engineering Institute (SEI) [34], 70% of all system defects are related to system requirements and architecture. Furthermore, a recent study [68] highlights that customer misconfigurations were the underlying cause of 65% of cloud incidents. Those defects and misconfigurations can significantly increase the vulnerability of an application, leading to potential security exposures or malfunctions. Therefore, a rigorous approach is required to deal with the overall system complexity when cloud native application developers design their systems and the corresponding access control mechanisms in cloud environments.

Research Question : Given the considerable complexity of configuring resource management and access control mechanisms for cloud-native applications, how might the abstraction and refinement approaches in formal methods help cloud-native application developers to handle these complex configurations when they design their systems?

The Aim: The goal of this research is to provide formal modelling patterns to help cloud-native application developers to manage the complexity related to designing their serverless systems and corresponding access control mechanism in a cloud environment. We also aim to create some guidelines on how app developers can use the proposed model patterns to model a specific functionality of their systems.

Objectives: To achieve the above goal, we are conducting the following approach:

- **Modelling the Behaviour of a Request:** Firstly, in a cloud environment, a functionality is posed through requests. Therefore, a generic formal pattern to represent the detailed behaviour of a request life-cycle helps us to comprehend how components in a cloud environment work. To satisfy a specific functionality, the process may necessitate the engagement of various services, requiring the execution of several requests in a particular order. Therefore, by producing formal patterns for common case of request ordering, a clearer comprehension of the functionalities inherent to cloud-native applications can be achieved. Those patterns can assist cloud-native application developers in the modelling and development of the functionalities of their complex cloud-native systems.
- **Modelling the Access Control Mechanism:** Secondly, in the cloud environment, any request attempting to access a resource must undergo authorisation mechanism. Therefore, following the modelling of the detailed behaviour of a request, we introduce the access control mechanism and its effect on a request's acceptance or denial. By employing an abstraction strategy in the formal modelling of the access control mechanism, we aim to give a clearer perspective for comprehending the complex procedures of the authorisation mechanism in the cloud environment. This may also help cloud-native developers to design their systems in a more effective way in terms of configuring the access mechanisms that impact their cloud-native systems.
- **Scenarios as Case Studies:** Finally, three distinct functionalities of two different serverless applications have been chosen as case studies. These case studies are developed and formally modelled to understand how a cloud-native application works. This also helps to figure out how cloud services and resources connect to each other. Through the analysis of these case studies, this research aims to show how the proposed generic formal patterns can be instantiated into the model of a particular functionality of a specific cloud-native system.

1.1 Thesis Organisation

The report is organised as follows:

Chapter 2 explores cloud concepts and web services, detailing how technologies and concepts have evolved in cloud computing.

Chapter 3 begins by highlighting the importance of formal methods in software development and then describes specific formal languages, such as Event-Calculus, TLA+, and Event-B. After that, research concerning the implementation of formal methods in cloud-native systems and access controls is discussed.

Chapter 4 introduces the development of a pattern (RHP) to model the behaviour of the life-cycle of a request. And then by applying the RHP pattern, we produce several patterns for common cases of request ordering, representing a functionality in a serverless system.

In Chapter 5, we introduce a generic pattern for authorisation mechanism and show its connection with patterns developed in Chapter 4.

In Chapter 6, we implement our proposed patterns to model three distinct functionalities from two different case study systems, illustrating the usability of our patterns.

Finally, Chapter 7 concludes the report with a brief summary and an outline of future work.

Chapter 2

Web Services

This chapter outlines the evolution of cloud computing, transitioning from general concepts to the specifics of serverless architecture. It discusses cloud platforms that support serverless computing after highlighting the critical role of access control in cloud-native systems. The chapter concludes by addressing the inherent complexity of cloud-native systems, especially serverless systems, and advocating for formal methods as a solution to navigate this complexity.

2.1 A Brief History of Cloud Computing

Before cloud computing, companies managed their data and applications by building and using bare metal servers, requiring substantial investment. Building a bare metal server system not only incurred considerable costs but also demanded extensive efforts in terms of its construction, deployment, and management, making it an impractical option for many businesses. After that, on account of reducing these costs, servers are grouped into local networks to centralize file storage. This cluster of servers is referred to as a data center. Data centers provide shared computing resources, resulting in cost reductions. And then, hardware was virtualised to enhance resource usage efficiency and reduce the workload associated with physical hardware [109]. As depicted in Figure 2.2, virtualization introduces a middleware layer, known as a hypervisor, built atop the operating system (OS), which allows developers to run multiple operating systems on this virtualisation tier. Importantly, the virtualisation technology, which is a key component of the cloud computing concept [25], paved the way for the evolution of cloud computing [106, 110].

Cloud computing is a general term to identify a category of on-demand computing services. In the philosophy of cloud computing, resources are virtualised and readily accessible [86, 38]. Thanks to cloud technology, not only end-users/developers keep away from operational concerns about underlying hardware, but also the usage of the resource is increased. As highlighted in a report from the University of California, Berkeley [38], cloud computing has the

following essential characteristics: pay-per-use, elastic capacity, and virtualised resources. Furthermore, leading IT companies such as Microsoft, Google, Amazon, and IBM have built massive data centres offering virtualised/abstracted computing resources.

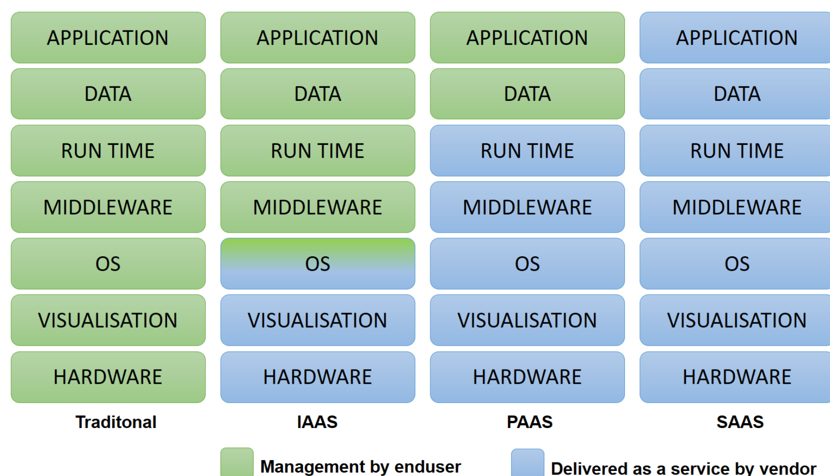


FIGURE 2.1: The concept of cloud computing, adapted from [42]

Advancements in virtualization technology have significantly increased the level of abstraction in cloud computing systems. This progress has led to the adoption of distinct concepts in the cloud computing paradigm, based on their level of abstraction. Figure 2.1 details the virtualization of the most prevalent cloud concepts: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Firstly, in IaaS architecture, which stands for Infrastructure as a Service, cloud providers offer highly scalable compute resources, networking, storage, and servers [16], whereas configuring the operating system, managing storage, and managing/deploying applications remain the responsibility of end-users. Notable examples of IaaS include Amazon Web Services [88], Microsoft Azure [7], and Google Compute Engine [73]. Moreover, in the case of PaaS architecture (Platform as a Service), both the operating system and server software are abstracted from end-users, in addition to the IaaS abstraction. AWS Elastic Beanstalk[91], Google App Engine [72], and Heroku [46] can be examples of the PaaS structure. Lastly, the highest abstraction level can be seen in the SaaS architecture (Software as a Service). In this architecture, even the applications are provided by vendors, leaving end-users with the role of merely using the application. Netsuite [67], Dropbox [30], and Salesforce [84] can be examples of software created with a SaaS structure.

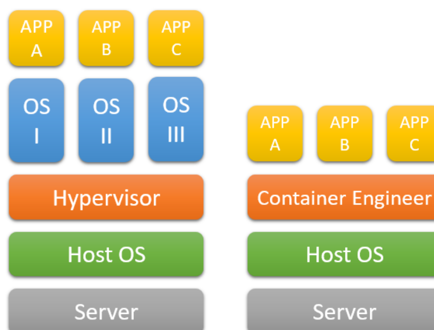


FIGURE 2.2: Hypervisor and Container, adapted from [14]

Containerisation takes virtualisation a step further, potentially reducing an application's dependence on a specific platform. The container architecture offers a logically segregated environment, facilitating the operation of applications. As a result, applications built within containers are platform-agnostic, potentially leading to significant improvements in resource utilization. However, in this architecture, developers are responsible for deploying containers and managing their administrative operations.

Briefly, throughout the evaluation of cloud technologies and tools, there are mainly two major trends that may be observed: increasing the efficiency of resource usage and decreasing the workload entrusted to end-users.

2.2 Microservices

In traditional software engineering, early applications were typically designed employing a monolithic architecture. In this architecture, an application and its dependencies are assembled into a single unit or container. As a result, the modules are not independent, so they do not run independently [29]. However, the monolithic structure causes some issues, including scalability, flexibility, and reliability. For instance, from a reliability viewpoint, if a single module of the system malfunctions, it could cause the entire system to fail.

In contrast, in a microservices architecture, the system is decomposed into small, autonomous components[55] that communicate with each other, often through APIs[62]. If a service in this architecture grows too large, it should be further divided into new, smaller services, maintaining the philosophy of simplicity inherent in microservices. As Fowler suggests[37], the idea behind the microservices architecture can be encapsulated by the phrase "do just one thing, but do it well". Therefore, because each component (microservice) is like a small, independent system, the agility of the overall system can be significantly enhanced in a microservices architecture[47].

2.3 Web Services

Web services are software components that are published, located, and invoked across the Web [77], making them accessible to other software systems [100, 43]. Similar to the microservices concept, web services also conceptualise the idea that individual applications can be divided into relatively small independent services [63]. In other words, each service specialises in a certain set of tasks, and their collaboration composes the overall software system. Before diving into how web services are composed, the types of web services will be discussed.

There are two kinds of web services that are commonly used: SOAP-based web services and RESTful web services [71]. First of all, SOAP, an acronym for Simple Object Access Protocol, is primarily an XML-based protocol used for communicating with a web service. It is used between the service consumer and the service provider. A service consumer could also be another service. A SOAP-based web service is published in WSDL (Web Services Description Language), providing an XML document that defines the web service. Moreover, these web services communicate with each other by using SOAP messages [63]. SOAP defines how services talk to each other. On the other hand, REST, standing for Representational State Transfer, is an architectural design that describes constraints related to web service implementation. RESTful services have URLs (Universal Resource Identifier) that provide access to them. System resources are managed and manipulated by HTTP methods, which are PUT, GET, POST, and DELETE [48]. Both Rest and SOAP web services allow developers to create their own APIs, enabling communication between services. However, RESTful web services can use different formats, including JSON, XML, or plain text, for their messaging, whereas SOAP only allows XML [65]. Moreover, the fact that these messaging formats are language-agnostic enables developers to build a system with web services that are developed in different languages or technologies.

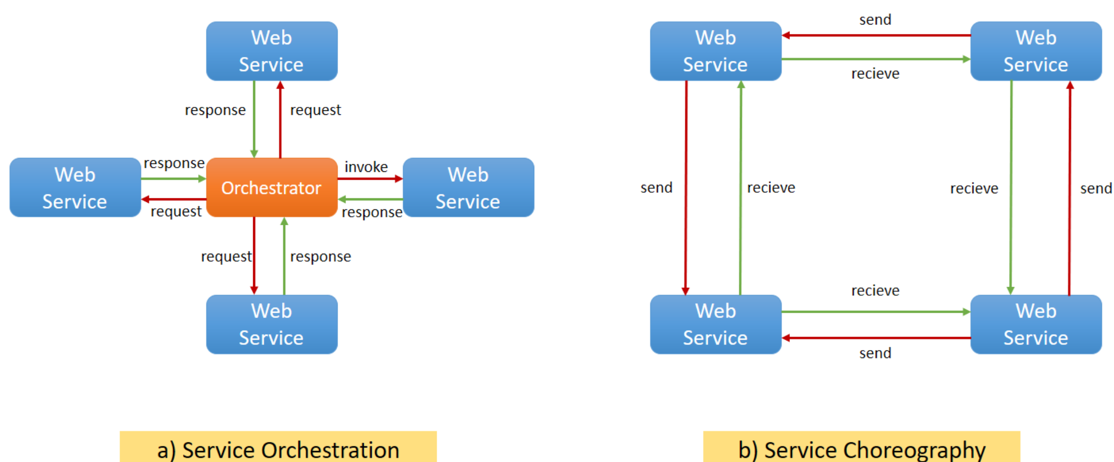


FIGURE 2.3: The patterns of Web Services Collaboration

When it comes to the collaboration of services, as illustrated in Figure 2.3, two patterns are common: service orchestration and service choreography. In the orchestration pattern, an orchestrator service organises the interactions among different services, following a centralised pattern. In contrast, the choreography pattern promotes decentralisation, where a web service communicates directly with other services. In this approach, no single service holds control over the others [63].

2.4 Serverless Technology

Serverless computing, an increasingly sought-after solution in the computing landscape, has been observed in research papers since 1995 [55]. However,the definition of the term "serverless" has changed over time. Nowadays, the term "serverless" is widely used for the Function as a Service (FaaS) architecture, even though, as Roberts [80] remarks, there is no consensus on what serverless precisely means.

FaaS, defined as "an event-driven cloud execution model" [52], offers to break down an application into a set of small stateless functions [64]. As previously noted, service-oriented architectures offers to split a monolithic structure into services, each focusing on specific business solutions. Then, with the advent of the microservices paradigm, an application or system can be divided into more fine-grained components, known as microservices. Lastly, the FaaS concept goes a step further, suggesting that an application be divided into even finer-grained functions. Because of the fact that these functions are more fine-grained than microservices, they could be termed nanoservices [55].

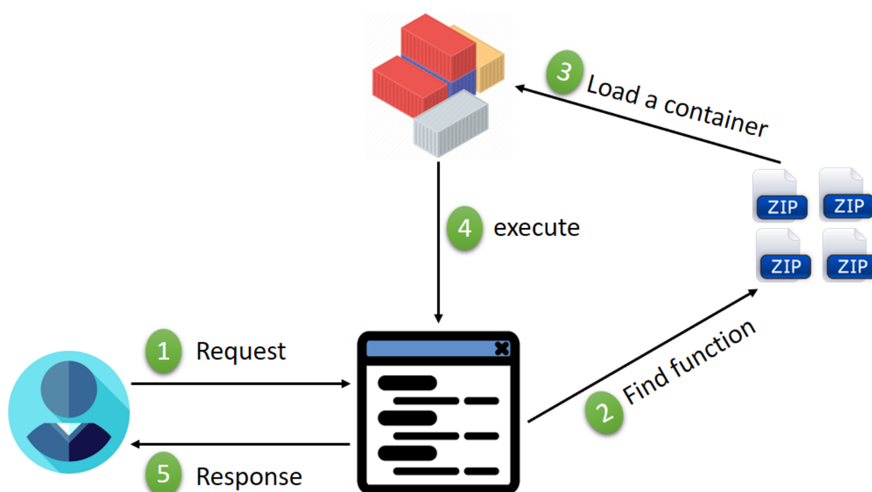


FIGURE 2.4: A general Structure of Serverless Concept, adapted from [115]

Figure 2.4 demonstrates a basic pipeline for a serverless architecture, generally involving the following steps:

1 Firstly, a user sends a request to the system.

2 The system finds related functions that fulfil the request's needs.

3 The function and related resources are loaded into a Docker container.

4 The container is executed.

5 The result is sent to the user as a response.

Furthermore, containerization is a key concept in FaaS architecture [64, 85]. To be more precise, short-lived container environments form the backbone of the serverless architecture. For instance, whenever a request for a serverless application is made, a short-lived container is created, and the serverless function code starts to run inside it. The containers have limited time to stay up idle. If it expires, the container is removed, and the resources keeping to response the request are freed. Moreover, although in serverless architectures, containers are widely used, the developers who build an app with serverless architectures are kept away from operations about deploying or managing a container.

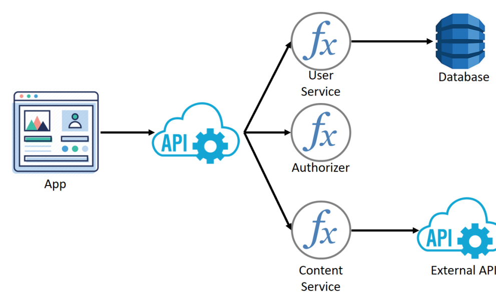


FIGURE 2.5: Structure of Application with Serverless Architecture

In a nutshell, serverless architecture, namely FaaS, as demonstrated in Figure 2.5, offers developers the ability to run function code in a compute service to trigger any cloud or third-party APIs without requiring any server-side or backend operations. Therefore, it can be said that functions run in a compute service, and APIs make up the skeleton of the serverless architecture.

2.4.1 Advantages of Serverless Architecture

Serverless architecture can have substantial benefits in terms of cost or workload from the application developer's point of view. For instance, a study by Villamizar et al. [107] shows that companies that build their systems with serverless architecture instead of monolithic architecture can drop infrastructure costs by up to 77%. The following points illustrate some of the key advantages of this architectural pattern:

- **Less server-side operations and more focus on business:** Most infrastructure and server-side operations are abstracted away from developers. This leads to developers concentrating more on their core business solutions.
- **Dynamic resource allocation:** Resources are dynamically allocated or deallocated based on the system's needs, resulting in considerable resource efficiency.
- **Reduced operational cost:** Another striking upside of serverless technology is its potential cost reduction. Cloud providers offering serverless technology typically implement a pay-per-use billing model, which allows end-users to pay only for resources when they are in use. To put it another way, no resource is allocated or chargeable until the serverless functions run [32]. This feature, coupled with auto-scaled resources, can provide immense savings, particularly when web traffic is highly fluctuating. To take the study by Adzic and Chatley [6] as an example, the cost reduction can see even 98%.
- **Enhanced system security:** While developers still need to address issues like authentication, authorization, and code vulnerabilities, cloud platforms handle most other security aspects [115]. The cloud platforms can also provide tools to assist developers in making their systems more secure. For example, Amazon offers the IAM (Identity and Access Management) [93] service for developers to specify access control for their resources.
- **Reduced Complexity:** The use of cloud services and serverless functions may reduce the complexity of the application's code.

2.4.2 Disadvantages of Serverless Architecture

However, serverless technology is not a silver bullet in all circumstances. In some cases, serverless may not be a good solution. Some limitations of serverless architecture include:

- **Short-lived functions:** Due to the fact that serverless functions have a limited execution lifecycle time, for instance, a lambda function can stay alive for up to 900 seconds [95], serverless architecture proves an ineffective solution for tasks involving long-running operations.
- **Vendor lock-in risks:** The use of cloud services, tools, and APIs makes an application dependent on the cloud platform. Therefore, switching to a cloud provider may necessitate code modifications, changes to services, or even remodelling the whole structure of the application. This may make switching cloud providers exceedingly difficult. Moreover, reliance on cloud providers may adversely affect the performance of an application and potentially lead to system downtime. An illustrative example of this issue occurred in 2016 when Facebook decided to shut its Parse service, causing significant disruption for numerous applications [115]. This incident forced app developers to find an alternative service or solution instead of Parse.

- **Security Concerns:** The shared environment of serverless architectures can raise security concerns. The responsibility of securing the serverless application is divided between the cloud provider and the developer, which may lead to ambiguities.

2.5 Cloud Platforms Offering Serverless

With advances in cloud concepts and the technology of Web services, business solutions can be designed, built, and offered on the Internet. Several major IT companies, such as Facebook, Microsoft, Twitter, and Google, provide resources and web services to enterprises. The enterprises use them in their business solutions as third-party components [100].

Besides the vast array of web services, many cloud providers, including Amazon, Google, and Microsoft, also offer compute services that enable application developers to create and execute serverless functions. As mentioned Section 2.4, those serverless functions can be used to build business logic of a serverless system. Therefore, the serverless architecture is primarily function-oriented. A frontend/client application connects with serverless functions through APIs.

Microsoft, Google, and Amazon are among the most prevalent and highly regarded third-party providers of serverless technology. In the following sections, these cloud platforms will be discussed in more detail.

2.5.1 Microsoft Azure

Azure [7] is a platform for cloud computing services that was released by Microsoft in 2010. It offers IaaS, PaaS, and SaaS technologies and tools. With Azure, application developers are able to use Microsoft's own cloud services and resources without the need to manage internal infrastructure [2].

In the Microsoft Azure platform, Azure Function [9] and Azure API Management [8] services form the backbone of Azure Serverless design patterns. These services can also be used as gluing components to create backend services to provide specific business logic for an application [101].

Azure Function is a Microsoft computing service that enables the development of business solutions on the Azure platform. First of all, it employs a code-first approach [2], allowing application developers to execute small segments of code, referred to as Azure serverless functions, without concerning themselves with the deployment or management process. These functions, which can be written in C#, F#, Node.js, Python, PHP, batch, or bash, can trigger or be triggered by any APIs or services. Secondly, Azure functions can be integrated with various development or deployment tools, like Visual Studio Team, OneDrive, Dropbox, and Git [102]. Last but not least, Microsoft Azure also provides numerous templates for common problems as

well [101], which may assist developers in creating a more secure serverless pattern for their business problems.

Azure API Management is a Microsoft service that allows you to create and publish both internal and external APIs. APIs can play a key role in communication between services and provide a connection between front-end applications and back-end systems built with services provided by the Azure platform. As previously mentioned, Azure API Management and Azure functions form the skeleton of an Azure serverless solution.

2.5.2 Google Cloud

GCP [75] (abbreviated for Google Cloud Platform), provided by Google, comprises an extensive array of services that allow developers partial access to Google's internal infrastructure [49]. Among these services, two of them are especially crucial to building a business solution with serverless architecture in GCP, which are Google Cloud Functions [76] and Google Cloud Endpoints [74]. Their philosophy is similar to the aforementioned Azure functions and Azure API Management. Google Cloud Endpoints allow developers to develop, deploy, and manage APIs, while by using Google Cloud Functions, developers can build serverless functions in a GCP environment.

One of the specific features of GCP is its integration with Firebase [35], which is a BaaS (Backend as a Service) platform for mobile and web application developers. Firebase offers many services for development, while Google Cloud Functions can respond to events by triggering some Firebase elements or/and HTTPS requests [66].

2.5.3 Amazon Web Services

Amazon is one of the most prominent cloud platforms, offering serverless technology alongside a wide range of web services. The AWS (Amazon Web Services) platform, officially launched in 2006, initially provided three main services: Simple Storage Service (S3) [96] and EC2 (Elastic Compute Cloud) [92]. The number of web services offered by AWS has since grown exponentially, with 80 web services available in 2011, 280 in 2013, 1,017 in 2016, and 1,430 in 2017 [28]. This rapid pace of growth makes Amazon dominant in cloud technology. According to a report released by Synergy Group [45] in 2017, AWS constitutes 34% of the cloud market (IaaS, PaaS), whereas other platforms such as Microsoft and Google have 11% and 8%, respectively, in the cloud markets.

As mentioned before, a serverless architecture fundamentally relies on two components: serverless functions provided by a computing service and APIs. In the AWS context, serverless functions are created and managed by AWS Lambda [94], while APIs are deployed and managed by the AWS API Gateway [89].

In AWS serverless architecture, a frontend app mostly connects with backend services (AWS services) using restful APIs created and managed by the AWS API Gateway. Each API endpoint triggers a Lambda serverless function. After Lambda functions process the incoming data from API calls, these functions connect with an AWS service to perform the requested action, such as writing a record to a database.

2.6 Security and Access Controls in AWS Environments

Given our focus on AWS-based serverless systems, this section will provide detailed information about the architecture of AWS-based serverless systems, along with a comprehensive discussion on access control mechanisms within AWS environments.

2.6.1 AWS-Based Serverless Application Architecture

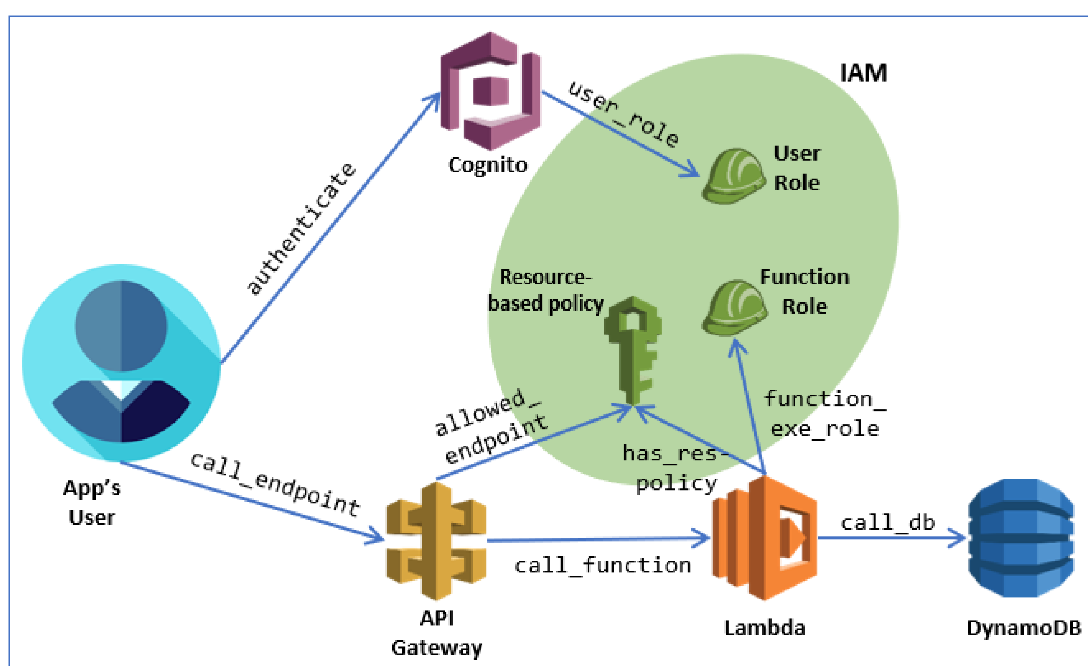


FIGURE 2.6: A Basic Structure of a Web App with Serverless Architecture in AWS Environment

Figure 2.6 illustrates the basic structure of an application with serverless architecture in an AWS environment. In AWS Serverless Architecture, firstly, Amazon Cognito provides an authentication process for application users. Moreover, Cognito can distribute IAM roles as user roles for app users to provide different access levels in the AWS cloud system. Then, an authenticated user connects to the system through API endpoints that are created and managed by the API Gateways service. An authorizer in the API Gateway checks the permission of the user's role to execute the requested endpoint or not. If it allows, the requested endpoint is executed, and the requested endpoint calls the related lambda function (a serverless function

in the AWS context). The called function will be executed in a container to fulfil the request. The function may need to connect to any other AWS service or a third-party external API. In the case of the Figure 2.6, it needs to make a request to Dynamo (an AWS service that offers a non-relational database) to fulfil the request. For instance, it could be to update the data in the database. A lambda function needs permission when it makes a request to other services. The function can provide this permission by using an IAM role as an execution role.

To understand the AWS architecture, the AWS services that are used in Figure 2.6 are explained in more detail.

2.6.1.1 Amazon Cognito

Amazon Cognito[90] is an identity provider service, offering authentication, authorization, and user management for web/mobile applications. Amazon Cognito consists of two components, which are the user pool and the identity pool. The user pool serves as a user directory to create and store users for applications, whereas the identity pool allows application developers to manage the permissions of users of their applications[115] on AWS resources.

Identities (users) can be provided in several ways, such as by fetching from social accounts, creating in the Cognito user pools, or fetching from any private identity providers. Then, the Cognito identity pool provides unique AWS credentials for each user. The permissions allowing access to AWS resources are also assigned to these credentials. Therefore, authorised users can access and use AWS resources.

2.6.1.2 AWS API Gateway

AWS API Gateway[89] is a web service that allows developers to create and manage RESTful APIs. The APIs, which are managed and deployed by API Gateway, make a bridge between frontend applications and backend AWS services.

In most serverless scenarios, API Gateway makes a connection between Lambda functions (serverless functions) and a frontend application. The app client communicates with the system in the AWS cloud environment through APIs provided by the API Gateway using CRUD operations (Create, Read, Update, Delete). Additionally, these APIs trigger lambda functions, which interact with other AWS services.

Moreover, permissions for different levels of access can also be created for APIs, thereby enhancing system security. API Gateway supports several mechanisms for controlling and managing access to your API, such as API Gateway resource-based policies and IAM-based authorizers. With an API Gateway Resource-based policy, one can specify a whitelist or blacklist for accessing the API resources, whereas an IAM-based authorizer grants access to application users to access API endpoints based on their respective IAM roles.

2.6.1.3 AWS Lambda

AWS Lambda[94] is an event—driven computing web service[103] that allows developers to run their code without concerning about the provision or management of servers. As mentioned before, serverless functions, or lambda functions in the AWS context, are a core concept for serverless architecture.

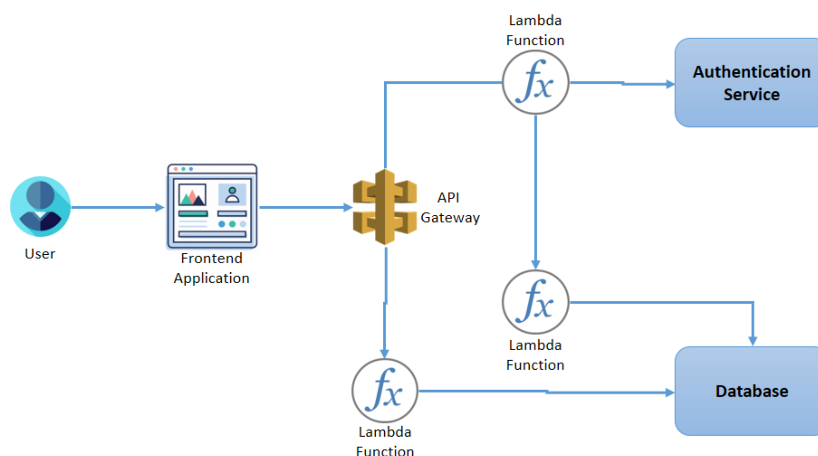


FIGURE 2.7: The structure of a serverless application

As shown in Figure 2.7, the backend of a serverless system seems to be mostly dominated by APIs and lambda functions. The lambda functions are written and executed as isolated, independent, stateless, and often granular[103].

Lastly, Lambda functions may make a request to an AWS service to provide inputs for its process. However, in AWS philosophy, no service has permission to perform an action on another service. As also shown in Figure 2.6, the required permissions can be granted to lambda functions via IAM roles.

To understand how a request is allowed or denied in the AWS cloud, we should have a closer look at the access control mechanism, which is mainly managed by the AWS IAM (Identity and Access Management) service.

2.6.1.4 AWS IAM

AWS Identity and Access Management (IAM)[93] is a fundamental web service defining and managing permissions in the AWS environment. The IAM service basically consists of two components, which are IAM identities and IAM policies. The IAM identities control the authentication, while the IAM policies determine the authorization of resource usage.

IAM policies are AWS JSON—based access control objects that make a connection between AWS identities and AWS resources by defining permissions.

Each AWS account has one root user who has full administrative rights. IAM identities allow the management of an AWS account by multiple users, if required. Moreover, the permissions for these identities are specified by the IAM policy provided. Therefore, different access levels for an AWS cloud account are provided. There are two different IAM identities: IAM users and IAM roles. The difference between an IAM user and an IAM role is that the former uses permanent credentials to access AWS cloud resources, whereas the latter uses temporary credentials. In some cases, the temporary credentials might be crucial, like a front-end application needing AWS credentials to connect with AWS resources. Moreover, the main aspect of the IAM role is the ability to share its permission with another object outside of IAM services.

2.6.2 Authorization and Access Control in AWS

The authorization and access control mechanisms in AWS are managed by the IAM service. As previously mentioned, the access control mechanism is managed by using IAM identities and IAM policies. With those components, restricted access to cloud resources can be designed.

IAM Policy is a JSON-based AWS object that defines the level of access for IAM identities or resources. In the AWS cloud environment, there are basically two fundamental types of policies: identity-based policy and resource-based policy, categorised according to their function in granting permissions. The identity-based policies specify what actions an IAM identity can perform on what resources, whereas the resource-based policies define who can access the related resource with what permissions. To put it another way, identity-based policy grants permissions to IAM identities to access resources in an AWS account, while resource-based policy is directly attached to a resource to restrict or grant access to the related resource[87]. Notably, not all AWS services support the resource-based policies. For instance, among the services in the structure depicted in Figure 2.6, only API Gateway and Lambda services (functions) support resource-based policies[98].

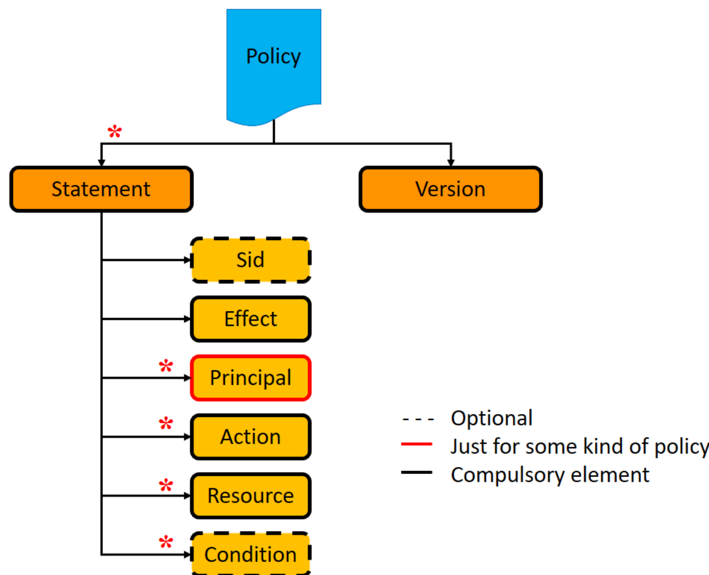


FIGURE 2.8: The structure of an AWS IAM Policy

Figure 2.8 illustrates the basic structure of IAM policy. A policy consists of a version part and a statement part. The version indicates the version of the policy language, while the statement can consist of an array of statements that define the permissions. Each statement may include the following elements:

- **Sid** : It is an optional statement ID to differentiate between your statements.
- **Effect** : It indicates whether the policy allows or denies access. Its value could be "Allow" or "Deny".
- **Principal** : It shows the actors (AWS root users, IAM users, IAM roles, or federated users) whose access is allowed or denied.
- **Action** : It includes a list of actions that the policy allows or denies.
- **Resource** : It specifies a list of resources to which the actions apply.
- **Condition** : It defines the circumstances under which the policy grants permission.

```
{ "Version": "2012-10-17",
  "Statement": [
    { "Sid": "72Bat44Sot",
      "Effect": "Allow",
      "Action": [ "dynamodb:PutItem" ],
      "Resource": [ "arn:aws:dynamodb:[region]:[account-id]:table/project" ,
                    "arn:aws:dynamodb:[region]:[account-id]:table/department" ,
                    "arn:aws:dynamodb:[region]:[account-id]:table/employee"
                  ]
    } }
```

The **statement** says that:
* **Adding a record to department, employee, or project table in DB is allowed.**

Action
Resource
Effect

FIGURE 2.9: An AWS IAM Policy Example

Figure 2.9 illustrates an IAM policy instance. Any IAM identity with the policy shown in Figure 2.9 is granted permission to write on *project*, *department*, and *employee* tables in DynamoDB database.

When a request is sent to a resource in the AWS cloud, the AWS framework aggregates all the statements that the requester has, along with all the statements of the requested resource if it has a resource-based policy. And then, over those statements, the logic of the authorization in AWS works like following :

- By default, all requests are denied. (implicit deny)
- If there is at least one "allow" statement in the context, and if there is no "deny" statement about the request, the request is allowed. (explicit allow)
- However, if there is at least one "deny" statement about the request, the request is denied because a "deny" statement always overwrites an "allow" statement. (explicit deny)

For instance, User1 wants to write on Table1 in the database. In the beginning, by default, there is no permission to perform the requested action (implicit deny). To grant permission to write on Table1 successfully, User1 must have a policy to allow writing on Table1 (explicit allow), but s/he must not have any policy to restrict this action (explicit deny).

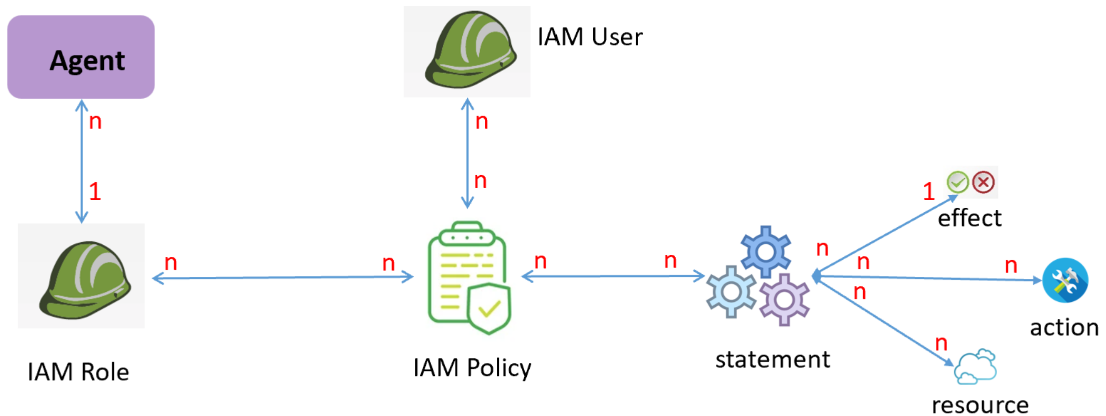


FIGURE 2.10: The Relation between IAM Identities and Policies

As depicted in Figure 2.10, IAM identities have a set of policies, while the permissions that are granted or restricted are defined with statements.

As shown in figure 2.10, the main object for permission is policy. A policy has an array of statements. Moreover, each statement grants or restricts a permission that is about to perform a set of actions on a set of resources. Therefore, statements in a policy define permissions that the policy provides for its related entities. These entities could be an IAM user or an IAM role. An IAM role can be used by an agent, which could be an app user or a function to access a resource in AWS.

To see how the authorization mechanism in the AWS environment works, take the scenario of writing data by an app user in a DB table in the system illustrated in figure 2.6 as an example. The following requirements should be satisfied to allow requests:

- The app user role must be allowed to execute the endpoint that is related to writing data in API Gateway.
- The related function must have a resource-based policy that allows the endpoint related to data writing to execute the function.
- The executed function must have an execution IAM role that allows it to write data to the requested table in the database.

As seen from this example, the authorization of a request that satisfies the business-level functionality of a system that is built with serverless architecture could be considerably complex. Therefore, the design of access control mechanisms for a serverless system becomes crucial.

2.7 Conclusion

Despite advances in distributed systems and cloud technologies that provide highly distributed and scalable applications, developing these types of applications introduces significant complexity, particularly in service composition and access control configurations. This increase in complexity heightens the likelihood of defects and security gaps, potentially leading to substantial losses. For instance, a poorly designed authorization mechanism may allow unauthorised access to system resources. Therefore, it is crucial for serverless application developers to carefully design service compositions and authorization mechanisms to prevent such adverse outcomes. Robust development methods and verified solutions are more important than ever, and our aim is to contribute to this goal.

Chapter 3

Formal Methods

As computer systems become more complex and their role in modern economics becomes increasingly critical, the presence of potential defects can be extremely hazardous. Therefore, detecting and fixing potential bugs before the application/system becomes operational is crucial for preventing losses and making the system more secure. Numerous studies have explored the impact of requirements analysis and architectural design in the software development life cycle. Take the Glass and Boehm studies as examples; both studies highlight these impacts. The Boehm first law states that [18, 33]:

> *"Errors are most frequent during the requirements and design activities and are the more expensive the later they are removed."*

whereas Glass's law said that [40]:

> *"Requirement deficiencies are the prime source for project failures."*

A report published by the Carnegie-Mellon Software Engineering Institute (SEI) in 2013 [34] shows that defects related to requirements analysis and architecture design can constitute about 70% of all system defects. However, as Butler highlighted [22], early identification of errors in development could be difficult by virtue of a lack of precision in formulating specifications and the high complexity of a system. This complexity can manifest in several ways, such as the complexity of requirements,the multifaceted nature of system environments, or the sophisticated design of a system itself. To deal with requirement ambiguities and system complexity, formal methods may enter the picture. With formal methods, defects related to design can be significantly reduced or eliminated during the early stages of the development life cycle [33].

3.1 Formal Specification

Formal methods are techniques to specify, develop, and verify a system using mathematical notation [19]. As Lamport stated [58], mathematics is a standard language developed over a couple of millennia to describe things precisely. By using this precision inherent in mathematical expression, formal methods create a more precise and consistent system design and specification than those founded on natural language based methods.

When it comes to challenges related to complexity, formal methods offer the concept of abstraction to get a clear understanding of a system. The abstraction concept helps to focus on the essential functionality of a system, thereby allowing for the temporary disregard of intricate details [22]. Such an approach ensures that the underlying complexity does not obstruct the analysis and comprehension of the system's core objectives and behaviour.

3.2 Formal Specification Languages

There is a wide array of formal method specification languages to implement formal modelling in a software development cycle. They are mostly based on techniques in mathematical and logical notation to build a rigorous model of a system. Such a formal model may enable the features of the system to be verifiable.

Formal specification languages can be categorised into model-based specification languages and algebraic (or axiomatic) specification languages.

3.2.1 Model-based Specification Languages

Model-based specification is a formal method technique that models a system using a system state model to describe its behaviours. In this approach, invariants that define constraints should be satisfied at all times [60], while transitions between states are detailed through pre-conditions and post-conditions [17]. Z [111], VDM (Vienna Development Method [51], and Event-B [3] are examples of a model-based approach.

3.2.2 Algebraic Specification Languages

In the algebraic approach, also known as the axiomatic approach, the behaviour of a system is modelled as axioms. The structure of algebraic languages basically consists of abstract data types, function signatures, and axioms that define what functions do. Notable examples of algebraic specification languages include OBJ [41] and CASL (Common Algebraic Specification Language) [111].

3.2.3 Formal Specification Languages used for Cloud Systems

Formal methods can be used by both cloud providers and app developers who develop systems in cloud environments. For instance, Amazon uses formal methods to enhance the security of its cloud environment [28, 70], whereas the formal method based tool Zelkova provided by Amazon is used by cloud customers to identify potential misconfigurations in their systems [12].

Moreover, languages such as TLA+ [112], Event-Calculus [54], and Event-B [3] are examples of formal methods languages for modelling cloud-native systems. These languages are commonly employed to model service-based systems or access control mechanisms. First of all, TLA+ is used by the Amazon Formal Verification team to detect design flaws in various AWS systems, including but not limited to DynamoDB, S3, and EC2 [69]. Then, Event-Calculus offers a formal approach to defining actions and their consequent effects. There are several studies, like [113] and [114], that use Event-Calculus to model access control mechanisms in the cloud. Lastly, Event-B allows developers to design their systems using an incremental approach [21]. This incremental approach lets you design a complex system very abstractly, followed by gradual and systematic development of the model. Such an approach is beneficial in dealing with system complexity, which is mostly high in service-based systems. There are also several studies, such as [56] and [11], that focus on modeling and verifying service-based systems with Event-B.

3.2.4 TLA+

TLA+ is a formal specification language rooted in TLA (Temporal Logic of Actions), first-order logic, and ZF set theory [59]. TLA is a logic language to describe and reason about the behaviour of concurrent or distributed systems [57]. TLA+ embodies TLA in a formal language by involving first-order logic and ZF set theory, which helps to form large and modular specifications [112]. A typical specification in the TLA+ language is:

$$Init \land \Box Next \land Liveness$$

Init describes all initial states in a system, while $\Box$*Next* specifies the next state relation. *Liveness* defines the liveness of a system as a conjunction of fairness conditions on actions [59]. Once a system has been formally modelled using TLA+, the correctness of the formal model can be assessed with TLC, a dedicated model checker for TLA+ specifications [112].

3.2.5 Event-Calculus

Event-Calculus is a logic-based formal language to describe actions and their consequential effects [99]. In Event-Calculus, the treatment of time is based on the notion of events, which

are represented in the form of Horn clauses[1], which can be executed as a logic program [54].

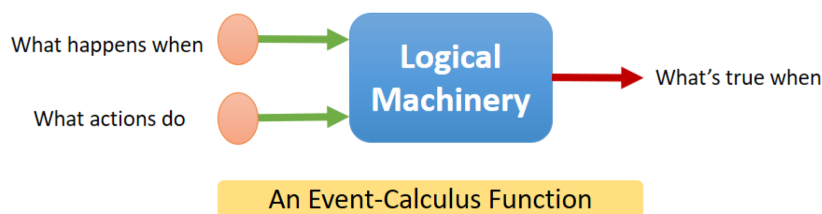


FIGURE 3.1: Roughly the mechanism in Event-Calculus, adapted from [99]

Figure 3.1 shows a basic structure of the event-calculus function. As shown in figure 3.1, Event-calculus represents a logical mechanism for inferring *"what is true when"* predicated on the inputs that define *"what happens when"* and *"what actions do"*. For example, supposing that "listening to music makes me relax" and "I listen to music at ten o'clock" are given to the Event-Calculus mechanism as inputs. Provided these inputs to the Event-Calculus mechanism, the mechanism would certificate the output "I am relaxing at 10:20". If nothing else breaks the event that makes me relax, the "be relax" event will continue to execute.

3.2.6 Event-B

Event-B [3] is a formal modelling language grounded in set theory and predicate logic, developed by Abrial. The key features of the Event-B language are abstraction and refinement, which help to manage system complexity [21]. In the Event-B approach, a system is initially modelled in an abstract form, and then the model is developed through refinement steps to build a comprehensive model that fully encapsulates the system's characteristics. This "incremental fashion" [3] brings Event-B to the fore. Although abstraction of a system is a common feature across various formal specification languages, the use of refinement to model incrementally is a typical feature of Event-B developments.

3.2.6.1 The Structure and Syntax

An Event-B model consists of two components, which are *context* and *machine*. The former represents the static part of the model, while the latter includes the dynamic components of the model. Contexts provide axiomatic properties of the model, whereas machines provide behavioural properties of the model [24].

[1] Horn clause: it is a subset of classical logic that is augmented with negation as failure [54]

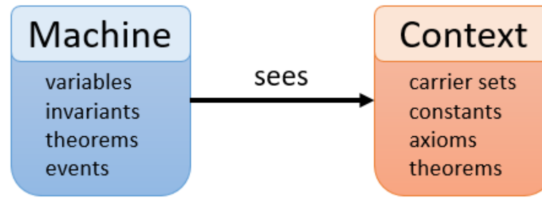


FIGURE 3.2: The structure of an Event-B model

Figure 3.2 illustrates the relationship between a machine and a context. The phrase *"machine_x sees context_x"* means that the static types formulated by *context_x* are accessible to *machine_x*.

In the *context* component, there can be four elements, which are *carrier sets*, *constants*, *axioms*, and *theorems*. *Carrier sets* represent user defined static types. *Axioms* define sets or constants, whereas *theorems* are features that *axioms* should follow [24].

When it comes to the *machine* component, it contains *variables*, *invariants*, *theorems*, and events. Here, *variables* define the states in the model, and *invariants* are predicates that must remain satisfied at all times. *Events*, on the other hand, are the statements that specify the change of state.

In Event-B modelling, a system is initially modelled at an abstract level. This abstract representation is then refined through subsequent steps, gradually shaping the model to correspond to the system's real-world configuration more closely. By dividing the development process into incremental steps, the refinement methodology permits developers to build the model in manageable stages. This iterative refinement approach helps in both comprehension and manipulation of complex systems, thereby illustrating the efficacy of Event-B in tackling complicated modelling tasks by breaking them down into more manageable pieces.

3.2.6.2 Refinement

Refinement in Event-B refers to the systematic process of developing an abstract model to more accurately represent the real system we want to build. The development process typically starts with an abstract model that focuses on only the most crucial functions [108]. After that, throughout the refinement steps, the model gradually developed by defining detailed design functions.

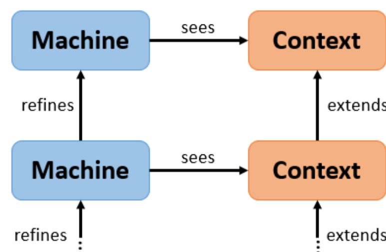


FIGURE 3.3: The follow of refinements in event-b Model

In the refinement mechanism, a machine can make an association, named *refines*, with another machine, while a context makes a connection, called *extends*, with another context. Crucially, there must be consistency between the refined machine and the machine being refined. Although the behaviour of the refined machine may differ, it must not conflict with the behaviour of the machine being refined [81]. This consistency is verified by using proofs [24].

Furthermore, there are two different approaches to refinement, which are vertical refinement and horizontal refinement [24]. In the *vertical refinement*, design details related to the existing functionality in the abstract model are added, whereas new functionalities that do not exist in the abstract model are defined in the *horizontal refinement*.

3.2.6.3 Proof Obligation

Proof Obligation (PO) is a key feature of Event-B formalism. POs are responsible for verifying properties of a model that need to be proved, such as the consistency between an abstract state and a concrete state. The outcome of these checks shows whether the feature has been formally verified. There are several different types of proof obligations, including *Well-Definedness (WD)*, *Invariant Preservation INV*, and *Guard Strengthening (GRD)*. Below is a brief description of these:

- *Well-definedness (WD):* WD POs assess whether the expressions/features in the model are well-defined or not.
- *Invariant Preservation (INV):* INVs are concerned with whether any *invariant* is violated when events run. Invariants are conditions that must hold true at all times, and this type of proof obligation ensures that these conditions are not violated during the state transition (event execution).
- *Guard Strengthening (GRD):* GRD POs ensure that a concrete guard is stronger than its abstract one. In operational terms, this means that the concrete event can only be enabled if the abstract event is also enabled. This helps maintain the alignment between the concrete and abstract aspects of the model.

By applying various proof obligations, the Event-B formalism guarantees that the model is rigorously verified at different levels, contributing to the overall integrity and reliability of the system.

3.2.6.4 Decomposition

As an Event-B model is elaborated through the refinement process, the complexity of the model inherently increases. In response to this escalating complexity, the *decomposition* approach

serves as a solution, allowing the division of an Event-B model into smaller sub-models. This division not only reduces the overall complexity of the model but also distributes Proof Obligations (POs) across the sub-models, which facilitate their discharge.

There are two styles for implementing the decomposition approach, which are *shared variable* and *shared event*. In the *shared-variable* style [24], an Event-B model is divided into sub-models based on variables, while *shared-event*, as proposed by Butler [20], offers to split the model with consideration of events. In the *event-shared* approach, an event is shared among sub-models, while variable sharing is not permitted.

3.2.6.5 Event-B Tool: Rodin

Rodin is an open-source tool that is developed on top of the Eclipse platform [31], supporting Event-B modelling, refinement, and mathematical proof steps [4]. It plays a critical role in using the implementation of Event-B formalism in software development.

Rodin has two key features, which are automation and extensibility. Firstly, the tool is built with the user's experience in mind. It automates the generation and discharge of proof obligations [23], which allows developers to focus on building and refining their models. Moreover, the Rodin tool platform allows for the seamless integration of additional development tools via plug-ins. This flexibility can help users adapt the Rodin platform to their specific needs, which may enhance modelling efficiency. For instance, the UML-B plug-in allows users to design the model as a flow chart using UML-like graphical notation and then generate a corresponding Event-B modelling script from it.

3.2.6.6 ProB (Animator and Model Checker for Event-B)

ProB is an animator and model-checking plug-in tool for the Rodin platform. It provides an animation of an Event-B model, allowing users to traverse between different states in the model. When a user traverses between states, ProB checks whether any invariant is violated or not [61]. Moreover, ProB will generate and display counter-examples when an invariant violation is discovered [5].

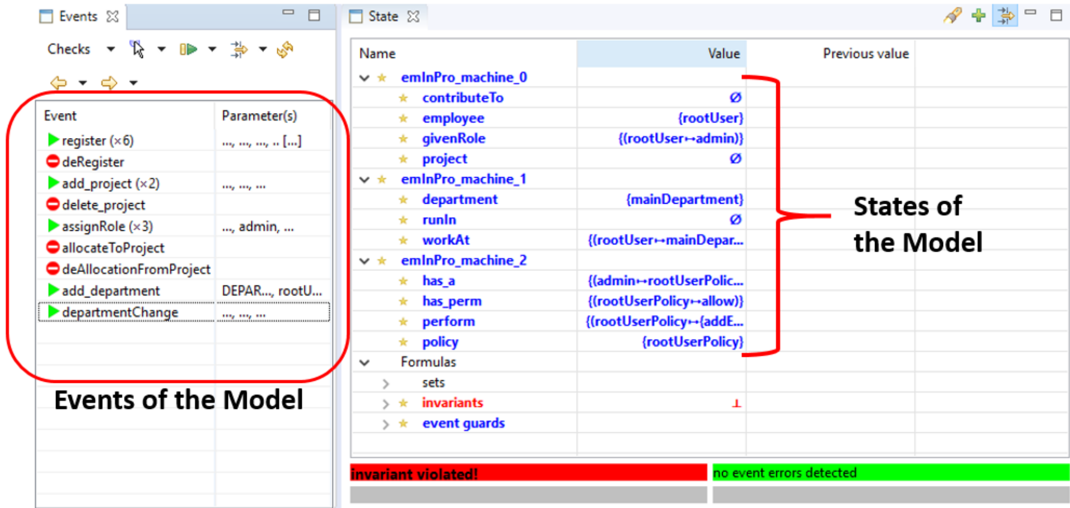


FIGURE 3.4: A view of ProB Model Checker

Figure 3.4 shows a screenshot of the ProB animator plugin in the Rodin Platform. On the left side of the figure, the events of the model are depicted, and users can run these events with a restricted set of parameters. Icons are used to denote the status of each event: a green icon means an event is available to run, while a red icon shows that it is disabled for execution. When an event is run, the values representing the current states of the model are illustrated in the top-right section of the screen. Meanwhile, the bottom-right side of the screen shows invariant violations and errors in events. For example, the current state in the Figure 3.4 states that there are no errors in events, but one or more invariant violations have occurred.

3.3 Related Work

3.3.1 Formal Methods in Web Services

Since web services and many resources serve as a service on cloud platforms, system complexity has escalated dramatically. Moreover, the pace of development on some cloud platforms has been exceedingly rapid. For instance, AWS released 80 services and features in 2011, whereas the number of its services and features saw 1430 in 2017 [28]. Amazon has used formal methods since 2011 to prevent the system from serious and subtle bugs related to system design [70]. In the paper explaining why Amazon uses formal methods [69], Newcombe stated that since Amazon has many sophisticated distributed systems and relentless rapid business growth, engineers at Amazon chose formal methods to assist them in solving challenging design problems in critical systems. In the context of the web services and cloud technologies industry, formal methods are used to seek solutions for problems primarily in two domains, which are *"security of cloud"* and *"security of customers in the cloud"*.

3.3.1.1 Security of Cloud

Firstly, the aim of the *"security of cloud"* tenet is to ensure that the cloud infrastructure remains secure following its launch or any modifications. The security team at AWS performs formal security reviews for all features and services [28]. Amazon also uses formal methods to prove that their initial boot code in the data centre is memory safe [27]. It's noteworthy that not just AWS but other cloud providers also use formal methods and techniques to provide a more secure environment in their cloud systems. For example, Microsoft Azure uses CloudSDV (Cloud Static Driver Verifier) [10] to enhance the security and reliability of software in cloud environments.

3.3.1.2 Security of Customers in The Cloud

The second domain perspective in which formal methods offer a solution is *"security of customers in the cloud"*. As mentioned in Section 2.6, access and use of resources in a cloud account are under the responsibility of cloud account owners. To help with this complex task, AWS offers a set of tools based on formal method techniques to customers to provide clarity about who can access their data and resources with what permission. For example, Zelkova [12], which is an SMT (Satisfiability Modulo Theories) based formal reasoning tool, can be used to analyse access control policies and determine their degree of openness in a cloud environment. There are also two different research studies on offering a formal reasoning solution to enhance the security of access control policies. The [113] study offers a formal attribute-based access control model developed with Event-Calculus that is able to model and verify AWS IAM policies, whereas the [114] study addresses the problem related to conflicts in policies in multi-cloud environments. In the [114] study, Zahoor specified and classified the policy conflicts and offered an Event-Calculus based model to reason about them.

3.3.2 Web Service Composition

Web services may connect to each other to create business logic. Services communicate with each other via messages. In the intricate process of designing service composition, subtle errors may arise that can undermine the functionality of the system. At this point, formal methods provide a critical advantage, as many of them come equipped with tools to verify the correctness of service compositions. For instance, the study run by Lahouij et al. in 2018 [56] offers an Event-B based formal approach to prove the correctness of cloud composite services, whereas the framework offered by [82] support formal modelling for data-centric web services, aiming to verify the correctness properties for service compositions. Moreover, Abbassi et al. have proposed an incremental design approach by using Event-B to model and verify the dynamic reconfiguration of web service compositions [1].

3.3.3 Access Control

Formal methods are also useful techniques to verify access control mechanisms. The study by Vistbakka [108] specifies and verifies a dynamic role-based access control[2] using Event-B, whereas Gouglidis proposes a formal definition for the ABAC (Attribute-Based Access Control) model to formally verify resilience specifications in a set of ABAC policies [44]. Moreover, Khayat and Abdallah have provided a formal state-based model for Flat Role-Based Access Control (FRBAC), the specification of which is described in Z notation [53]. Finally, Tarkhanov offers a policy algebra-based method to solve the problems related to rights restrictions in enterprise document management systems [104].

3.3.4 Other Relevant Works

There are many studies to depict that formal methods make a significant contribution to the development of web-based or distributed systems. To begin with, an approach developed by Rauf et al. verifies the correctness of a system built with REST services by using Event-B and its refinement technique [78]. Moreover, [50] formalises the Web Services Atomic Transaction(WS-AT) protocol in TLA^+ and uses the TLC model checker to detect unexpected behaviors. Finally, the research by Rezazadeh [79] states the contribution of formal methods to web-based application development. As shown in Rezazadeh's research, the refinement and decomposition features of Event-B are so beneficial when designing distributed / web-based systems.

3.4 Conclusion

To conclude, formal methods offer substantial benefits for detecting subtle bugs in design or making a design more secure, especially in complex or distributed systems. Formal modelling languages, like Event-Calculus, TLA+, and Event-B, draw an abstract formal model of a system, which gives a comprehensive understanding of the system. However, two features of Event-B, which are iterative refinement strategies, decomposition, and having a powerful tool, make Event-B a more efficient solution in the domain of complex and distributed systems. The refinement approach allows designers to conceptualise a system at a highly abstract level, progressively developing the model through several refinement steps, whereas the decomposition approach lets designers divide the model into several sub-models if the model is too big to handle. These methodologies not only facilitate the handling of complex systems but also contribute to a more robust and accurate design process.

[2] A Role-based dynamic access control for a reporting management system

Chapter 4

Formal Patterns for Serverless App

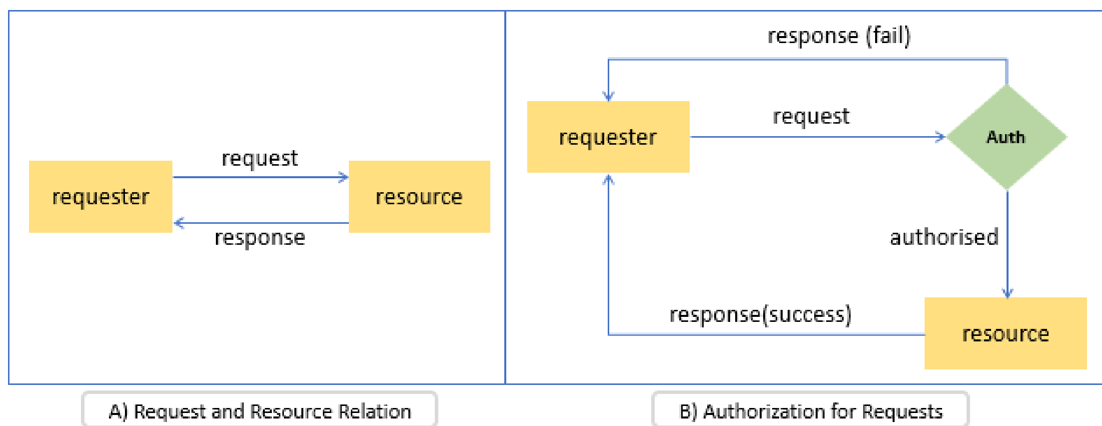


FIGURE 4.1: Access to Resources

In cloud-based systems, as illustrated in Figure 4.1A, everything is considered a resource that can be accessed by sending/processing a request. However, to successfully fulfil the request, the requester must have the appropriate permissions to access the requested resource (Figure 4.1B). Therefore, request execution and the corresponding authorization mechanisms become crucial for effective and secure resource management in cloud-based systems.

In this chapter, our objective is to develop formal model patterns to assist cloud-native application developers in modelling their systems. We introduce two distinct types of formal patterns, which are the Request Handling Pattern (RHP) and the Request Ordering Pattern (ROP). Moreover, in RHP pattern, we modelled the authorization mechanism at a high level, introducing it non-deterministically. In Chapter 5, this abstract formal representation of authorization mechanisms will be developed through stepwise refinement to fit the concrete system.

The Request Handling Pattern (RHP) represents the behaviour of a request in a generalised context. It outlines the lifecycle of a request, providing insights into how the request is processed and managed within a cloud-native environment. A detailed explanation of how the RHP pattern is formulated is provided in Section 4.1.

On the other hand, the Request Ordering Pattern (ROP) helps in the modelling of specific functionalities that are fulfilled by a single request execution or a set of requests that are executed in specific orders. The ROP patterns focus on a set of requests that must be executed in a particular sequence to achieve a desired outcome. Section 4.2 delves into how ROP patterns are developed and shows the complexity of request sequencing to fulfil a specific functionality.

4.1 Request Handling Pattern (RHP)

In a cloud system, all entities are considered as resources. Moreover, API calls play a pivotal role in communication and interaction between resources and services. For instance, in the AWS cloud environment, each service has its own specific set of API endpoints, which are URLs that a client can use to send their API requests. In order to access the desired resources, clients can send requests to those specific endpoints. In the AWS cloud environment, authorization is required for each API request before it can be processed. The authorization is managed by an evaluation of IAM policies related to the requester.

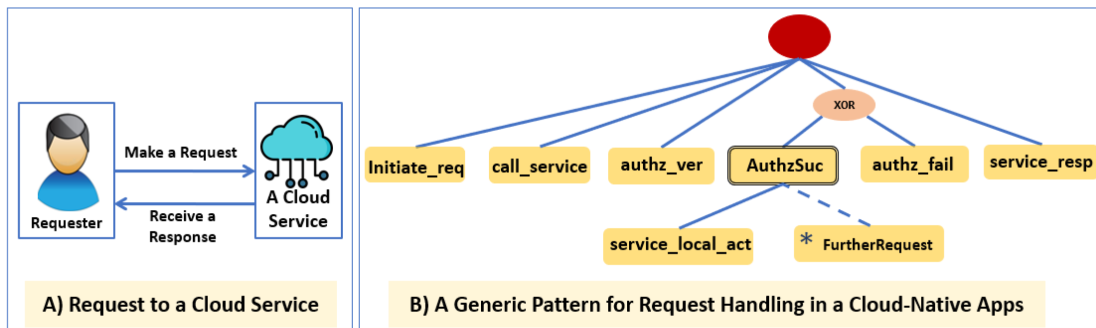


Figure 4.2: A Generic Pattern for Request Handling in a Cloud-Native Apps

Therefore, as shown in Figure 4.2A, a service exposes its functionality through API requests made by a requester client. If the requester is authorised, s/he can access the desired resources through the API request made. More specifically, at first, a request is initiated by a requester user before calling the corresponding service. Based on whether the requester is authorised or not, the execution of the requested action is succeeded or failed. Lastly, the service makes a response to the requester. As a result, the life cycle of a request can be introduced as a combination of processes.

The tree-like diagram in Figure 4.2B is the Request Handling Pattern (RHP), which illustrates the detailed behavioural aspects of a request life cycle. Drawing inspiration from Fathabadi's

work [83], our representation diverges by focusing on the sequence and interrelations of requests for a specific functionality or scenario fulfilment, rather than detailing a refinement strategy. The components or processes in the RHP in Figure 4.2B are :

- *initate_req* represents to initialise a request.
- *call_service* represents to call a service with the initialised request.
- *authz_ver* represents an authorizer to check whether the requester is authorised to perform the called request or not.
- *XOR* represents a choice that only one of its branches could be enabled at a time.
- *AuthzSuc* and *authz_fail* represent the authorization of a request execution. The former shows the success case, while the latter depicts the failure case. Throughout the life cycle of a request, only one of them can be enabled. This choice is determined based on the output of the authorizer, which is shown as *authz_ver* in the diagram.
- If the execution of a request succeeds (*AuthzSuc*), the required actions for the requested service are processed (*service_local_act*). If data or a process from any other service is needed to fulfil the request, a further request will be made that replicates a new RHP pattern again.
- *service_resp* represents a response to a request.

The order of execution of RHP components in Figure 4.2B are from left to right. The rectangular leaves without frame represent single task events whereas ones with double-line frame represent a process. Process means an abstraction of one or multiple events. The rectangular component with a star represents a new further request handling pattern while the dashed line shows that this component can be skipped in some cases.

Furthermore, a textual representation of RHP could be like (4.1) which may make pattern representation more readable in some cases. As shown in textual representation in(4.1), events showing the behaviour aspects of a request are the parameters for the RHP formal pattern.Those parameters are; ir (*initate_req*), cs (*call_service*), av (*authz_ver*), sla (*service_local_act*), FR (*Further Request*), af (*authz_fail*), and sr (*service_resp*). Therefore, the textual representation of the RHP can be shown as

$$\boxed{\begin{array}{l}\textbf{RHP}(\text{ir, cs, av, sla, FR, af, sr}) = \\ \text{ir} >> \text{cs} >> \text{av} >> (\text{AS } XOR \text{ af}) >> \text{sr} \\ \text{AS} = \text{sla} >> [\text{FR}]\end{array}} \tag{4.1}$$

The $>>$ operator in the textual representation in(4.1) shows the sequencing between the two parameter components, while XOR shows the choice between two parameter components.

Moreover, FR (Further Request) shows a new Request Handling Pattern for further requests. Therefore, the workflow of the pattern in 4.1 starts with *initate_req* and the execution of this process enables the execution of the next process, *call_service*, by representing the $>>$ operator. After the called request is checked whether it is authorised or not (*authz_ver*), the authorization will be succeeded ($AuthzSuc$) or failed ($authz_fail$). The choice is made by using the *XOR* operator. Then, if the authorization is succeeded, the local action is performed (*service_local_act*). And then, if a further request is required to fulfil the prior request, the optional further request is made, which creates a new RHP pattern for the new request. Lastly, after all required requests are fulfilled, the cloud system makes a response to the requester (*service_resp*).

4.2 Request Order Patterns

RHP is a generic pattern to model the detailed behaviour of a request life cycle when the request is processed. However, to fulfil a functionality, either a single request or multiple requests in the cloud system could be executed. Therefore, the order of requests' executions becomes crucial when multiple requests' executions are required to fulfil a functionality. For each functionality, the number and order of the required requests can vary. In this context, we introduce four request ordering patterns that are likely to be commonly used in designing a functionality. Those request ordering patterns are *Single Service Request* (SSR), *Linear Service Request* (LSR), *Branching Service Request* (BSR), and *Chained Branching Service Request* (CBSR). Moreover, each request in ROP patterns is defined by the RHP pattern. If there are multiple requests in a ROP pattern, the tracking among requests is satisfied by extra flags and relations, such as a relation that keeps the preceding and subsequent requests. Briefly, the RHP pattern introduces the detailed behaviour of a single request execution, while ROP patterns introduce a functionality that may require multiple request executions in a particular order. In other words, ROP patterns consist of multiple RHP patterns in a specific order.

4.2.1 Single Service Request (SSR)

In this structure, as shown in Figure 4.3A, a functionality is fulfilled by making a request to a cloud service, and the requested service can fulfil the request without necessitating for any data or process from any other services to fulfil the requested functionality.

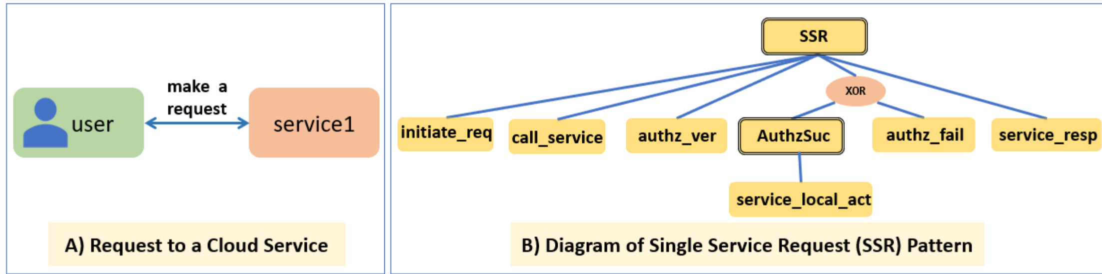


FIGURE 4.3: Single Service Request Pattern

Figure 4.3A shows the tree-like diagram of the SSR pattern. As shown in the diagram, if a request is authorised successfully, *service_local_act* event will fulfil the request. There is no need for further requests. Therefore, the textual representation of SSR is :

$$\begin{array}{|l|}\hline \textbf{SSR}(\text{ir, cs, av, sla, af, sr}) = \\ \textbf{RHP}(\text{ir, cs, av, sla, SKIP, af, sr}) = \\ \\ \text{ir} >> \text{cs} >> \text{av} >> (\text{AS } XOR \text{ af}) >> \text{sr} \\ \text{AS} = \text{sla} \\ \hline \end{array} \tag{4.2}$$

The textual representation 4.2 defines the SSR pattern in RHP. However, because there is no further request, the parameter *FR* (Further Request) in RHP is replaced with *SKIP*, meaning that there is no further request. Because of that, we introduce the RHP and SSR patterns together in the same section.

4.2.1.1 Representation RHP and SSR Patterns in Event-B

As the SSR pattern introduces a single request execution, the module includes a single RHP pattern. Because there are no further requests in a functionality fulfilment process, there is no need for tracking between requests.

```
context context01
sets
REQUEST
STATUS
EFFECT //The Outcome of Authorisation
constants
Initiated      Called
Succeded       Failed

Allow   Deny
axioms
@axm0_1 partition(STATUS,{Initiated},{Called},{Succeded},{Failed})
@axm0_2 partition(EFFECT,{Allow},{Deny})
end
```

FIGURE 4.4: Entities for Request Handling Pattern (Context01)

To model the RHP pattern in Event-B, we introduce REQUEST, STATUS, and EFFECT *sets* as basic types in the context of the abstract model, as detailed in Figure 4.4. STATUS helps us to track a request throughout its execution life-cycle, whereas EFFECT helps us to flag a request allowed or denied as their authorization result.

The *request* variable in *inv0_1* shows the registered requests that are already while *processed_reqs*(*inv0_2*) illustrates the request that are processed and respond.

invariants
@inv0_1 request ⊆ REQUEST
@inv0_2 processed_reqs ⊆ request
@inv0_3 : req_status ∈ request → STATUS
@inv0_4 : req_authz ∈ request ⇸ EFFECT

The status and the authorization result of a request are defined as variables in the machine since they could be changed during the execution life cycle. The variable *req_status* in *inv0_3* invariant says that each request has only one status, while *req_authz* in *inv0_4* says that a request could have at most one authorization outcome.

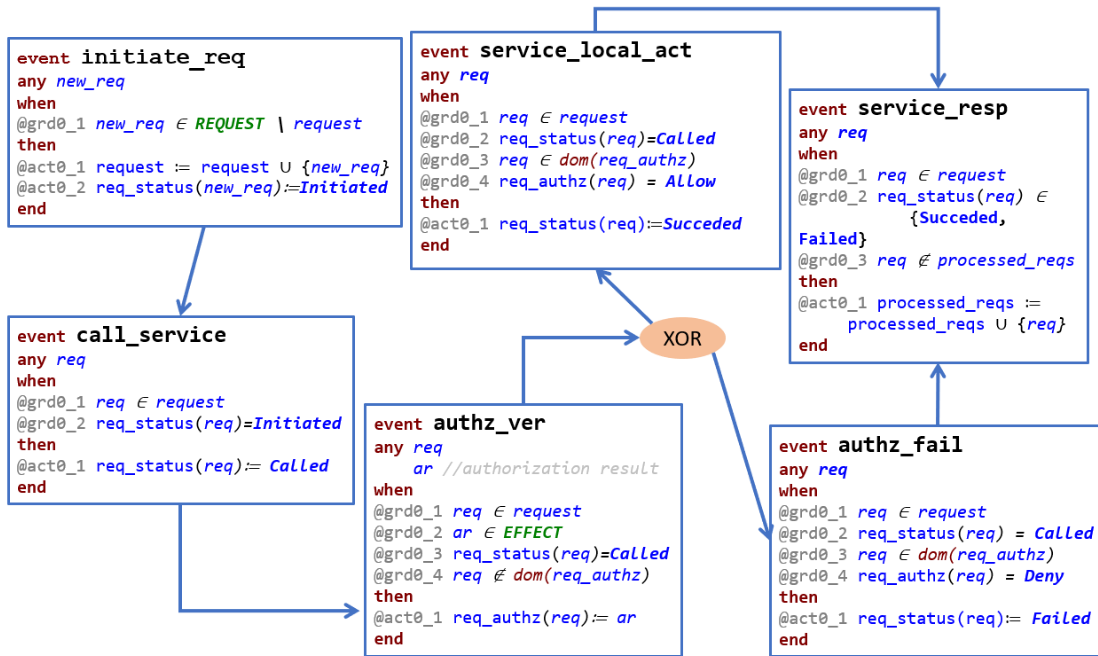


FIGURE 4.5: Event-B model of Request Handling Pattern

Each parameter in the RHP pattern in Equation (4.1) represents an Event-B event in our formal model. Moreover, as shown in the SSR pattern (Textual Representation 4.2), we assume that there are no further requests when the requester is authorised. The relationship and sequencing between those events are shown in Figure 4.5. Figure 4.5 shows the Event-B model of the RHP pattern, which is the abstraction of the SSR pattern at the same time. Before a request is made, only *initiate_req* is enabled. The *initiate_req* event represents the initiation of a new

request (*new_req*) by adding it to the **request** variable (@*act*0_1 in *initiate_req* event) and its status set to **Initiated** which enables the *call_service* event. Therefore, the sequencing between the events is achieved by updating the status of the request variable specified using the *req_status*. The *events* presented in Fig. 4.5 are a direct representation of Fig.4.3B, or the RHP pattern which is textually shown in (4.2). Moreover, the choice between the *service_local_act* and *authz_fail* events are nondeterministic at this level of abstraction.

```
context context02 extends context01
 sets
  RESOURCE
  ACTION
end
```

Moreover, in the refinement step, we need to introduce resources and actions to represent the requested resources and actions. Therefore, in the extended context, RESOURCE and ACTION are defined as carrier sets.

```
invariants
 @inv1_1 req_res ∈ request → RESOURCE
 @inv1_2 req_act ∈ request → ACTION
```

Additionally, in the refined machine, the requested resources and actions are introduced as invariants *inv1_1* and *inv1_2*, respectively. Therefore, when a request is initiated, the requested resource and the requested action must also be defined, as shown in the refined *init_req* event by *act1_1* and *act1_2*, respectively.

```
event init_req extends init_req
  any res  act
  where
    @grd1_1 res ∈ RESOURCE
    @grd1_2 act ∈ ACTION
  then
    @act1_1 req_res(new_req) := res
    @act1_2 req_act(new_req) := act
end
```

However, when a client user submits a service request, the requested service may rely on data or processes from other services in order to effectively meet the client's requirements.

4.2.2 Linear Service Request (LSR)

FIGURE 4.6: The structure of multiple requests which have LSR pattern

Typically, to fulfil a functionality, the execution of multiple requests in a specific order is required. Therefore, when a client user initiates a request for a service, the requested service may necessitate the utilisation of data or processes from other services to fulfil the client's request. The basic pattern could be a linear chain for requests' executions, as shown in Figure 4.6. In the structure of this pattern, each individual request may require, at most, a singular supplementary request to other services to adequately fulfil its own task. Take the pattern in Figure 4.6 as an example; it represents a functionality that can be satisfied when requests submitted to the services in the figure are fulfilled in the ordered way. To satisfy the functionality visualised in Figure 4.6, a client user sends a *request 1* to *service 1*, but the *service 1* needs a data or process from *service 2* to fulfil *request 1*. Therefore, *service 1* sends *request 2* to *service 2* to get the required data or processes. However, *service 2* also requires *request 3* for *service 3* to fulfil *request 2*. In summary, the fulfilment of *request 1* depends on *request 2*, which depends on the successful fulfilment of *request 3*.

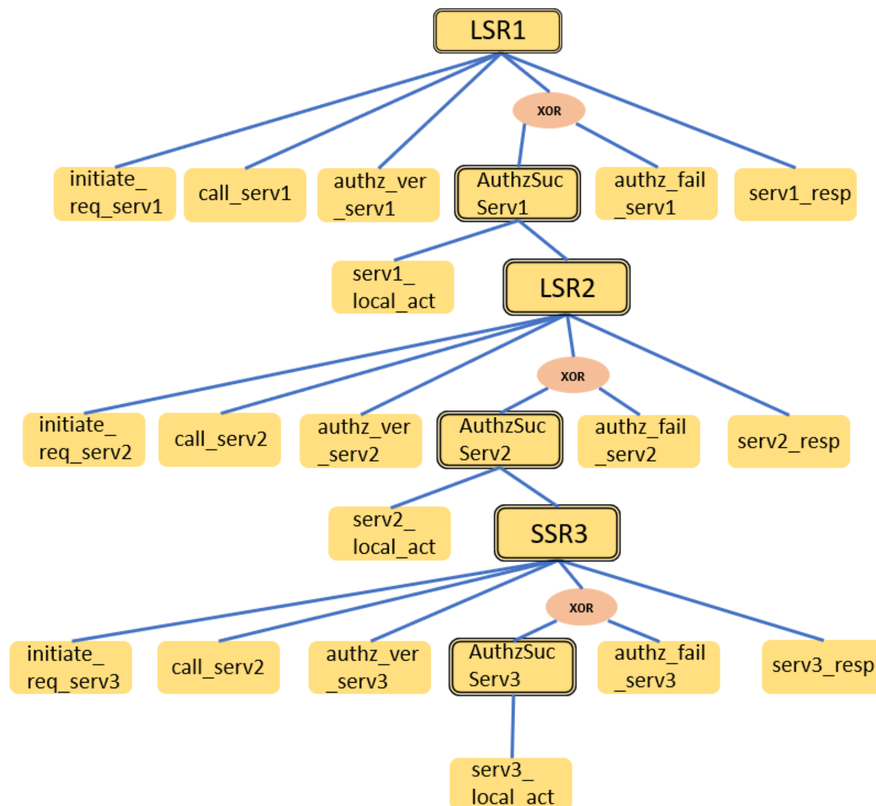


FIGURE 4.7: The tree-like diagram of a LSR pattern

Figure 4.7 is a tree-like representation of the LSR pattern in Figure 4.6. Each level in the tree introduces a RHP pattern for a request by starting with *request 1* for service 1. As seen in the diagram, the execution of a request depends on the authorization result of its predecessor request.

LSR1 =
RHP(irs1, cs1, av1, sla1, LSR2, af1, sr1)
LSR2 =
RHP(irs2, cs2, av2, sla2, SSR3, af2, sr2)
SSR3 =
RHP(irs3, cs3, av3, sla3,SKIP, af3, sr3)

(4.3)

Textual Pattern 4.3 shows the textual representation of the LSR pattern in Figure 4.7. As shown in both the graphical (Fig.4.7) and textual (4.3) representations of the LSR pattern, the last required request (*request 3*) has the SSR pattern because no further request is needed to fulfil *request 3*.

4.2.2.1 Representation LSR Pattern in Event-B

To model the LSR pattern, it is essential to identify and differentiate the target resources or services. Therefore, we refine the RHP pattern by introducing features to define to track different request in a process and define the requested resource and action action.

As a solution, we first introduce *RESOURCE* as a new type to represent various resources in a cloud account. Moreover, as depicted in *axm1_1*, we separate resources into six different services to define different resources at a high level.

```
sets
 RESOURCE
constants
 Service1   Service2   Service3
 Service4   Service5   Service6
axioms
 @axm1_1 partition(RESOURCE, Service1, Service2, Service3,
                   Service4, Service5, Service6)
```

In the SSR pattern, to track a request in its execution life cycle, we introduce the *req_status* variable. No additional tracking flag is needed due to the nature of the pattern, where a single request is sufficient to fulfil a functionality. However, in the LSR pattern, where a set of requests in sequential order is required to satisfy a functionality, it becomes essential to keep track

of requests as well during the process of functionality fulfilment. Therefore, we introduce *subsequent_reqs* as in *inv1_3* to keep track of requests whose outcomes influence one another. The domain of *subsequent_reqs* shows prior requests, while requests in the range represent subsequent/complementary requests. Moreover, *inv1_4* says a complement request can have at most one prior request.

invariants

@inv1_1 req_res $\in$ request $\rightarrow$ RESOURCE
@inv1_2 req_act $\in$ request $\rightarrow$ ACTION
@inv1_3 subsequent_reqs $\in$ request $\leftrightarrow$ request
@inv1_4 subsequent_reqs^{-1} $\in$ request $\pfun$ request
@inv1_5 has_further_req $\in$ request $\pfun$ BOOL

As previously mentioned, the last request in a sequence of the LSR pattern has an SSR pattern. This can be observed in *request 3*, as shown in Figure 4.6, where no further request is needed to fulfil *request 3*. Moreover, the *has_further_req* variable in *inv1_5* illustrates whether a request is the last request in a chain or not. Each request has a boolean value associated with it. Requests that map to the value of *FALSE* are considered the last request in their chain, indicating that no further request is required to fulfil the current functionality. Lastly, likewise in the SSR pattern, the *req_res* variable (*inv1_5*) shows the association between each request and the corresponding resource, such as services in Figure 4.6, whereas the *req_act* (*inv1_5*) variable shows the requested action.

The determination of the number of services and their interconnections to achieve the fulfilment of a functionality can be designed in the application set-up phase by the developer(s) of the application. We show our LSR pattern based on the design in Figure 4.6, where three specific services in a specific order are required to fulfil a functionality. The number of services does not impact the structure of our pattern; it can be applied to all designs where the necessary services follow a linear, sequential order.

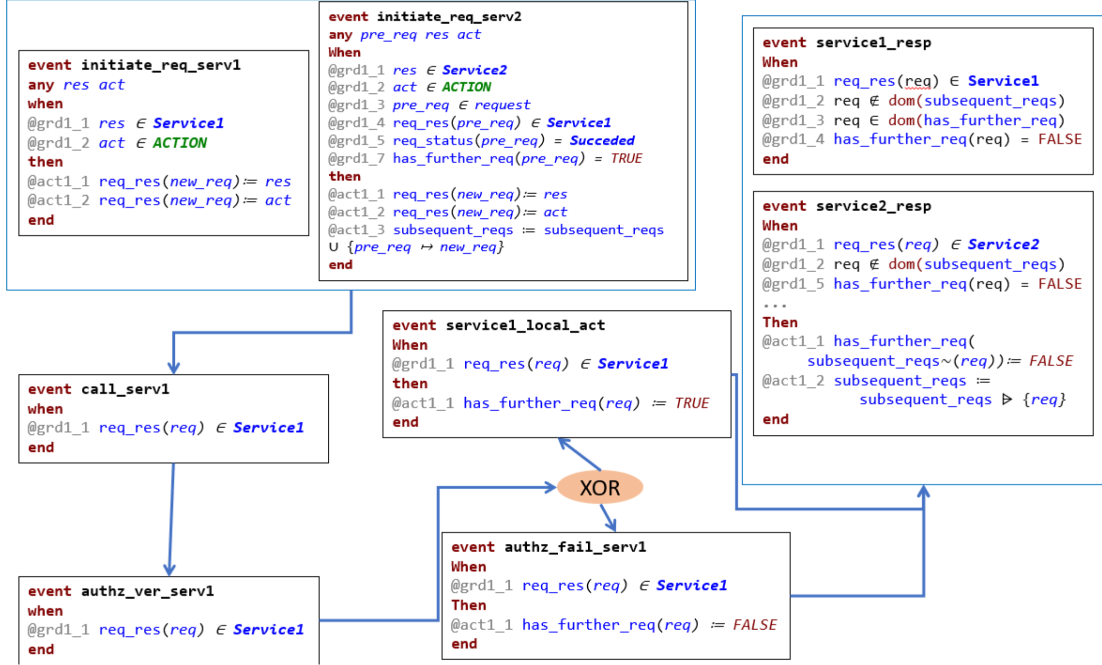


FIGURE 4.8: The Event-B events in a LSR pattern

Figure 4.8 shows some events in the refined machine where the features of the LSR pattern are introduced. The events in Figure 4.8 include all features of their equivalent events in the SSR pattern in Figure 4.5. For instance, the event of *call_serv1* in Figure 4.8 also involves all parameters, guards, and actions of *call_service* in Figure 4.5. Figure 4.8 illustrates the additional features that aid in clearly differentiating between the two patterns, thereby facilitating a more comprehensive understanding of their distinctions.

As shown both in the graphical and textual representation of the LSR pattern, a request is, firstly, initiated. To initiate a request, the authorization of its predecessor request is required, except for the first request in the pattern. Therefore, the first request in the pattern does not have a predecessor request, whereas others have a predecessor request, necessitating a check in the request initialization event. As shown in Figure 4.8, the *initiate_req_serv1* event represents the initiation of a request for *service1*, which is the first request in the pattern. *grd1_1* in *initiate_req_serv1* ensures that the request initiated for *service1*. *act1_1* makes an association between the requested resource (*res*) and the new request, showing that the *new_req* request is initiated for the resource of the *res*. Additionally, the parameter *act* defines the requested action.

Moreover, if the request is not the first request in the pattern, the predecessor request should also be taken into consideration. Take the *initiate_req_serv2* event in Figure 4.8 as an example; the parameter of *pre_req* represents the predecessor request. The *grd1_3* guard enforces the order of services as defined in the pattern, so to initiate a request for service 2, the predecessor request should be submitted to *service1*. *grd1_5* guarantees that the predecessor request was successfully executed, whereas *grd1_7* says that the predecessor request requires a further

request to be fulfilled. Lastly, an association between the predecessor request and the new request is created (*act1_3*).

Finally, in order to generate a response for a request, all required further requests must be fulfilled; for instance, *grd1_2* and *grd1_4* guards in *service1_resp* enforce that there must be no further request required for service 1 to fulfil its request. Moreover, when a request is responded, if the responded request is not the first request in the pattern, the *has_further_req* value of its predecessor request is updated as "FALSE" (*act1_1* in *service2_resp*), meaning that the predecessor request is ready to be fulfilled. Then the responded request is also removed from the *subsequent_reqs* variable (*act1_2* in *service2_resp*).

The Linear Request Service (LSR) Pattern may help us to model a functionality if the required requests follow a linear order. In this pattern, each request in the model requires, at most, one additional request to fulfil its task. However, the LSR pattern is not suitable for cases where any of the requests require more than one further request to be fulfilled. For instance, a service may require data from two different services to fulfil an incoming request. To address this limitation, our proposed Branching Service Request (BSR) pattern offers a solution to effectively model such more complex cases.

4.2.3 Branching Service Request (BSR)

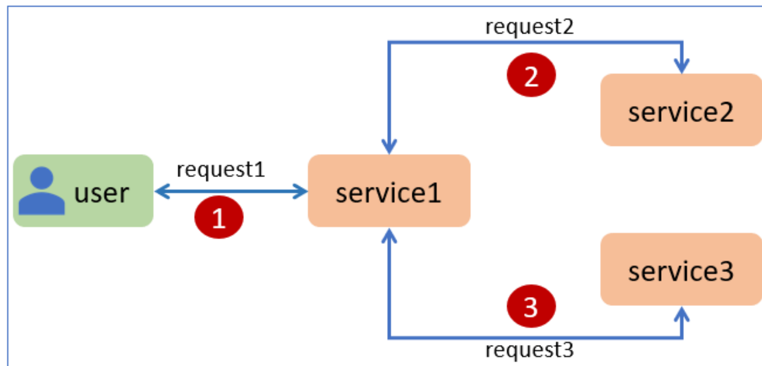


FIGURE 4.9: The structure of multiple requests which have BSR pattern

To satisfy a functionality, the required requests may need to follow a non-linear order. In such cases, a service that processes a request within the order might require input or involvement from multiple distinct services, resulting in branching structures, as exemplified by the request 1 to service 1 in Figure 4.9. To successfully fulfil request 1, service 1 must initiate request 2 to service 2, followed by request 3 to service 3.

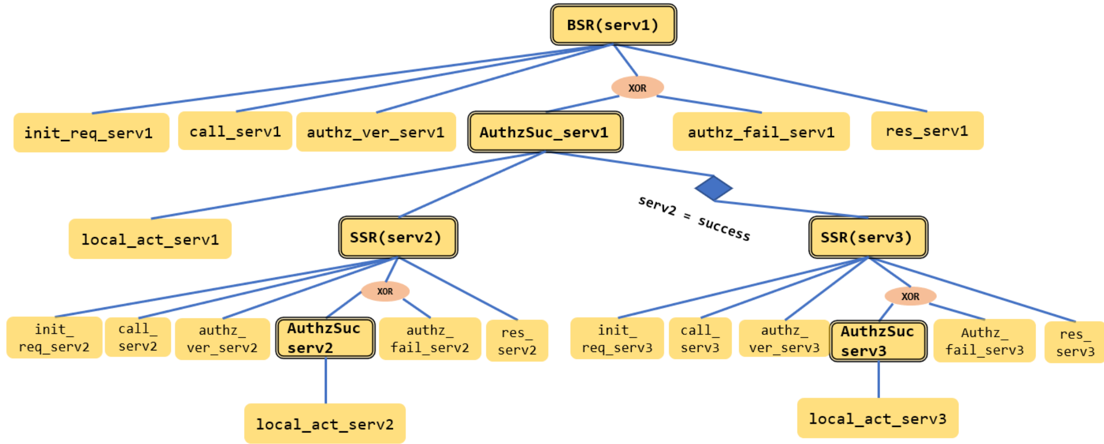


FIGURE 4.10: The Tree-like diagram of BSR Pattern

In order to effectively model this type of branching request order design, our proposed Branching Service Request (BSR) pattern offers a suitable approach. The textual representation 4.4 and the tree-like diagram in Figure fig:BSRTree illustrate the BSR pattern for the design in Figure 4.9. As clearly seen in Textual Representation 4.4, after request 1 is successfully authorised (*av1*), the sequence of events under *BSR2* is initiated, encapsulating the whole process conducted by service 1. To effectively execute Request 1, Service 1 necessitates data or process inputs from two different services.

BSR1 (irs1, cs1, av1, sla1, BSR2, af1, sr1) =
BSR2 = **SSR2** <> **SSR3**

SSR2 =
RHP(irs2, cs2, av2, sla2, SKIP, af2, sr2)

SSR3 =
RHP(irs3, cs3, av3, sla3, SKIP, af3, sr3)

(4.4)

BSR2 is defined in Textual Representation 4.4 as follows:
$BSR2 = SSR2 <> SSR3$

The "<>" symbol signifies that the initiation of the process on the right (*SSR3*) is dependent on the successful execution of the left process (*SSR2*). Then, both *SSR2* and *SSR3* processes follow the Single Service Request (SSR) pattern, as mentioned in Section 4.2.1. Moreover, in the graphical representation of the pattern (Figure 4.10), this condition is shown by the blue diamond operator.

4.2.3.1 Representation BSR Pattern in Event-B

Structures following the BSR pattern require a set of requests in a specific order to fulfil a functionality. However, unlike the LSR pattern, a request may require multiple further requests to be fulfilled. Therefore, the significant features of the BSR pattern are that we need to track all the requests made by the same service, and we should go into further steps only if all the requests made are successfully executed.

invariants

@inv1_6 : proc_further_req ∈ request ⇸ request

Therefore, we introduce the *proc_further_req* variable to track subsequent requests that have the same predecessor request. It basically shows the processed further/subsequent request. When a further request is fulfilled, it is removed from *subsequent_req* and added to *proc_further_req*. For example, assuming that *req1* requires *req2* request to be fulfilled, making *req1* and *req2* the predecessor request and the subsequent request, respectively. Therefore, the pair of *{req2 ↦ req1}* is added to *proc_further_req*, indicating that req2, which is required for req1 fulfilment, is fulfilled.

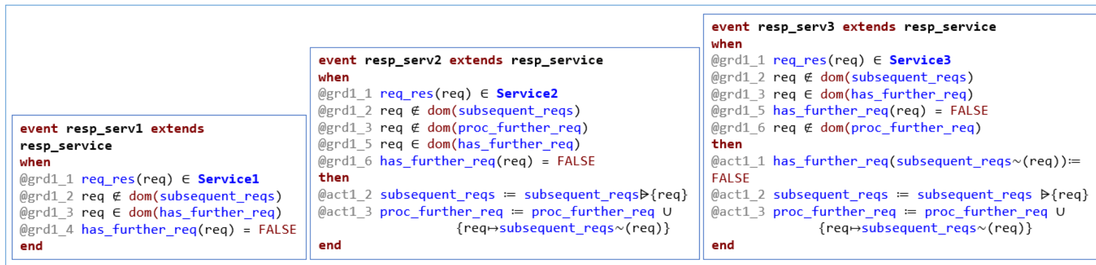


FIGURE 4.11: Response Events of subsequent request in BSR Pattern

The process of adding the executed subsequent request to *proc_further_req* will be done in *service_resp* events of subsequent requests. Figure 4.11 illustrates response events for service 1, service 2, and service 3. Action of *act1_3* in response events of service 2 and service 3 show the pair of processed request and its predecessor request are added to *proc_further_req*. The reason why we specify the *proc_further_req* variable as the reverse of *subsequent_reqs* is because it makes it easier to check the executed subsequent requests when a new branch is created, such as *grd1_9* in the *initiate_req_serv3* event in Figure 4.12.

```
event init_req_serv3 extends init_req
any pre_req  res   act
when
@grd1_1 res ∈ Service2
@grd1_2 act ∈ ACTION
@grd1_3 pre_req ∈ request
@grd1_4 req_res(pre_req) ∈ Service1
@grd1_5 req_status(pre_req) = Succeded
@grd1_6 pre_req ∈ dom(has_further_req)
@grd1_7 has_further_req(pre_req) = TRUE
@grd1_8 pre_req ∈ ran(proc_further_req)
@grd1_9 ∀r·r∈request ∧ r∈dom(proc_further_req) ∧
proc_further_req(r)= pre_req ⇒ r∈processed_reqs ∧ req_status(r) = Succeded
then
@act1_1 req_res(new_req)≔ res
@act1_2 req_act(new_req)≔ act
@act1_3 subsequent_reqs ≔ subsequent_reqs ∪ {pre_req↦new_req}
end
```

FIGURE 4.12: Initiate Request to Service3 Events in BSR Pattern

initiate_req_serv3 event in Figure 4.12 model a request initiation for Service 3 of the structure in Figure 4.9. Therefore, to initiate a request to service 3 (*irs3* of *SSR3* in textual representation 4.4), the request 2 that is made to service 2 needs to be successfully executed. The condition shown with the symbol "<>" in BSR (4.4) maps to *grd1_8* and *grd1_9* guards. *grd1_8* enforces that there is at least a fulfilled subsequent request of *pre_req* parameter request. Moreover, *grd1_9* guarantees that all the processed subsequent requests whose predecessor is*pre_req* parameter request are successfully executed.

In summary, our proposed BSR pattern offers an effective way to model a functionality whose process includes a request that requires data from multiple different services. In the BSR pattern, a single service might depend on inputs from several other services to fulfil incoming requests. However, this pattern reaches its limitations when the input-providing services form a sequential chain services, leading to complexities that the BSR pattern cannot efficiently address. For such cases, the Chained Branching Service Request (CBSR) pattern emerges as a more suitable solution, offering an advanced approach to handle the modeling more complex functionalities

4.2.4 Chained Branching Service Request (CBSR)

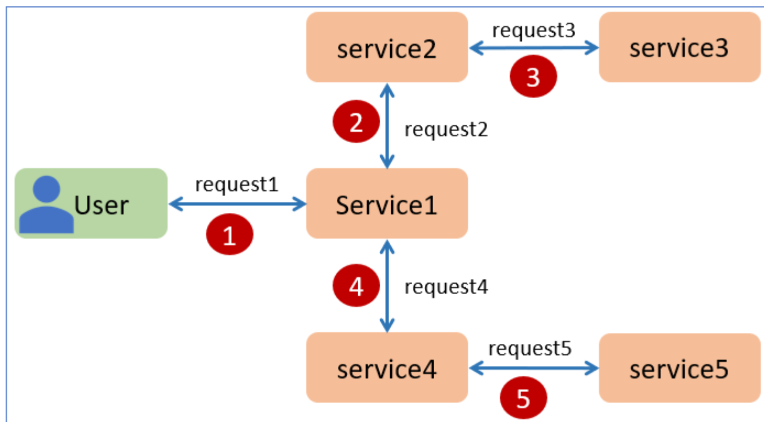


FIGURE 4.13: The structure of multiple requests which have CBSR pattern

For the fulfilment of a functionality, a cloud platform may require data or processing from a set of different services. As depicted in Figure 4.13, the order of those services can be quite complicated. There is a branching in the order of requests in Figure 4.13 akin to functionalities following a BSR pattern. However, unlike those adhering to a BSR pattern, Figure 4.13 illustrates branches comprising a series of distinct requests in a specified order, resembling a chain. To model such complex functionalities, we propose the Chained Branching Service Request (CBSR) pattern. In the CBSR pattern, each branch may constitute a chain of requests. Moreover, the failure of any request within a chain results in the omission of further request chains. For instance, in Figure 4.13, if any request in the chain starting with request 2 fails, the whole chain starting with request 4 will be skipped.

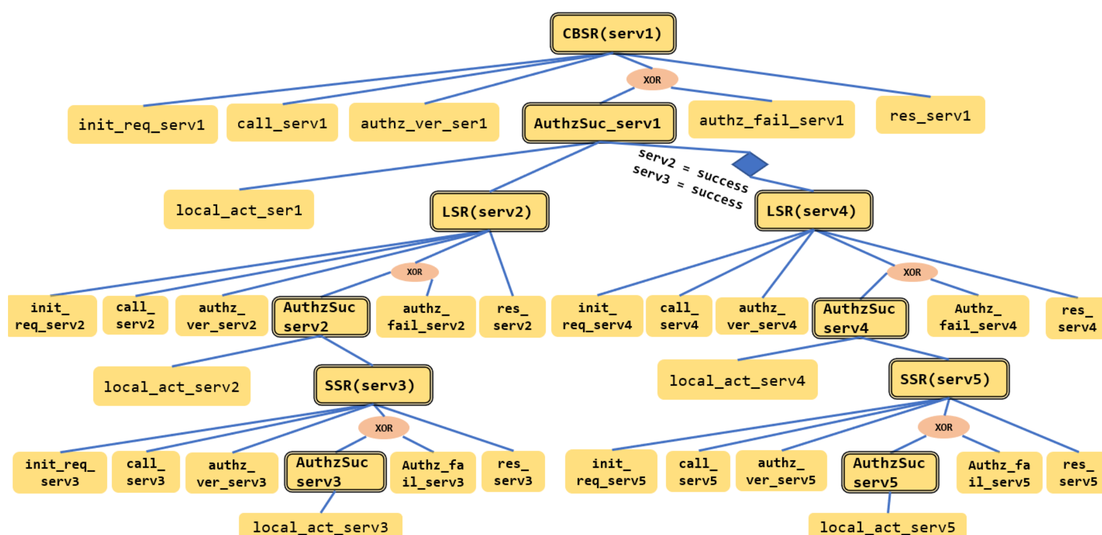


FIGURE 4.14: The Tree-like diagram of CBSR Pattern

Figure 4.14 shows a graphical representation of the *CBSR* pattern. As clearly seen from the diagram, for the fulfilment of request 1, both *LSR(serv2)* and *LSR(serv4)* branches must be successfully fulfilled.

Textual Representation 4.5 illustrates the Chained Branching Service Request (CBSR) pattern. If the first request's (request 1) authorization result (*av1*) is "Allowed", the process of *CBSR2*, which includes all further requests' processes for fulfilling request 1, is enabled.

CBSR1 (irs1, cs1, av1, sla1, CBSR2, af1, sr1) =

CBSR2 = **LSR2** <> **LSR4**

LSR2 =
RHP(irs2, cs2, av2, sla2, SSR3, af2, sr2)
SSR3 =
RHP(irs3, cs3, av3, sla3, SKIP, af3, sr3)

LSR4 =
RHP(irs4, cs4, av4, sla4, SSR5, af4, sr4)
SSR5 =
RHP(irs5, cs5, av5, sla5, SKIP, af5, sr5)

(4.5)

In Textual Representation 4.5, *CBSR* is defined as :
CBSR2 = **LSR2** <> **LSR4**
This implies that *CBSR2* is composed of two sub-processes, *LSR2* and *LSR4*, both of which abide by the Linear Service Request (LSR) pattern. Moreover, the <> symbol interposed between the sub-processes signifies that *LSR4* can only be initiated following the successful execution of all requests within the *LSR2* process. Therefore, within the context of the Textual Representation 4.5, request 4 can only be initiated after request 1, request 2, and request 3 are successfully executed, respectively.

4.2.4.1 Representation CBSR Pattern in Event-B

In the *CBSR* pattern, the execution of a request may require data or processing from multiple different services. Those data/processesing could be produced after a set of requests are executed in a specific order. For instance, service 1 can fulfil request 1 only after both *LSR2* and *LSR4* have been successfully executed, respectively. Therefore, to effectively track whether a request in a process fails or succeeds, we introduce "chain" and "chain status" into the model.

A chain defines a set of requests within the same process. Moreover, a new chain is defined whenever a new branching occurs, as demonstrated in Figure 4.15.

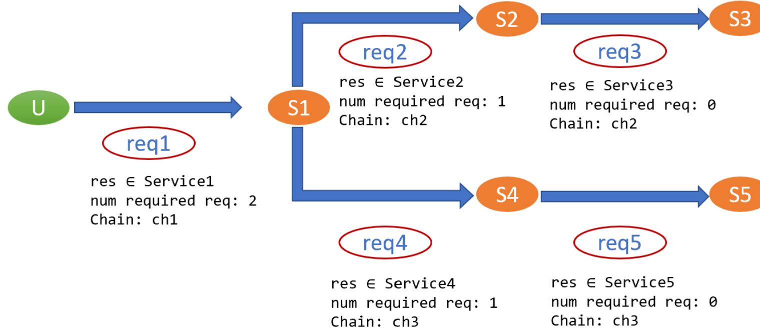


FIGURE 4.15: The relation between chain number and branching

Therefore, to model the CBSR pattern in Event-B, like other ROP patterns, we refine an RHP pattern. Besides all variables in the Event-B representation of the BSR pattern, we also introduce the chain concept for the request in the same branching. Firstly, we define CHAIN as a carrier set in the context to model request's chains. Moreover, in the refined machine, *inv1_7* illustrates the existing chains. *req_chain* in *inv1_8* depicts the association between requests and chains, whereas *chain_status* in *inv1_9* shows the status of chains.

```
invariants
  @inv1_7 chain ⊆ CHAIN
  @inv1_8 req_chain ∈ request ⇸ chain
  @inv1_9 chain_status ∈ chain → STATUS
```

The status of a chain could be either *Succeeded* or *Failed*. The chain status is updated to *Failed* if at least one of the requests within the chain fails. Conversely, the chain status could be *Succeeded* if only all the requests within the chain are successfully executed.

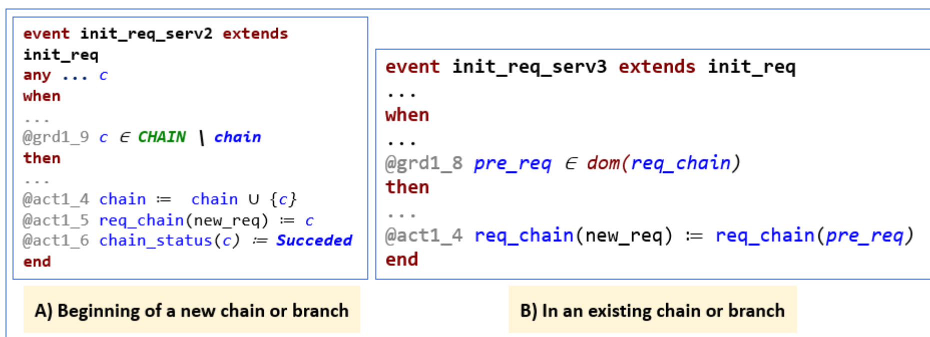


FIGURE 4.16: Guards and Actions About Chain in Initiate Request Events

If the initiated request is either the first request of an entire scenario or a new branching within the scenario, as in Figure 4.16A, a new chain is created (*act1_4*) and an association between the new request and the new chain is created (*act1_5*). The status of the chain is defined as *Succeded*

(*act1_6*). In other cases, the initiated request is in an existing branch, like both req3 and req5 requests in Figure 4.15, the newly initiated request is added to the chain of the predecessor request of the new request (*act1_4*), as shown in Figure 4.16B.

```
event authz_fail_... extends authz_fail
when
...
@grd1_2 req ∈ dom(req_chain)
then
@act1_1 has_further_req(req) ≔ FALSE
@act1_2 chain_status(req_chain(req)) ≔ Failed
end
```

FIGURE 4.17: Updating The Status of a Chain when a Request fails

The status of a chain is updated to *Failed* when at least one of the requests in the chain fails. As depicted in Figure 4.17, when a request fails, the status of its chain is updated to *Failed* in *act1_2*.

```
event init_req_serv4 extends init_req
any ...    c
when
...
@grd1_8 pre_req ∈ ran(proc_further_req)
@grd1_9 ∀r·r∈request ∧ r∈dom(proc_further_req) ∧
                    proc_further_req(r)=pre_req ⇒
   r ∈ processed_reqs ∧ req_status(r) = Succeded
@grd1_10 c ∈ CHAIN \ chain
then
...
@act1_4 chain ≔ chain ∪ {c}
@act1_5 req_chain(new_req) ≔ c
@act1_6 chain_status(c) ≔  Succeded
end
```

FIGURE 4.18: To Represent <> Condition in Event-B Model

Lastly, a service can make a second or more further requests if only all its previous requests and their corresponding subsequent requests are successfully executed. This is represented by the <> condition in the CBSR pattern (Textual Representation 4.5). To accommodate this condition, we introduce the <> condition with the guards of *grd1_8* and *grd1_9*, as shown in Figure 4.18.

The *grd1_8* ensures that the predecessor request (*pre_req*) has at least one processed subsequent request. Recall from Figure 4.12, *pre_req* is the predecessor of the newly initiated request as well. Moreover, the *grd1_9* guard ensures the authorization results of all processed subsequent requests of *pre_req* (*req_status*) and the status of their chain (*chain_status*) must be *Succeded*.

Thus, when request 4 is initiated, which is the beginning of *LSR4*, *grd1_9* guarantees that all requests within *LSR2* are successfully executed.

4.2.5 Comparison of ROP Patterns

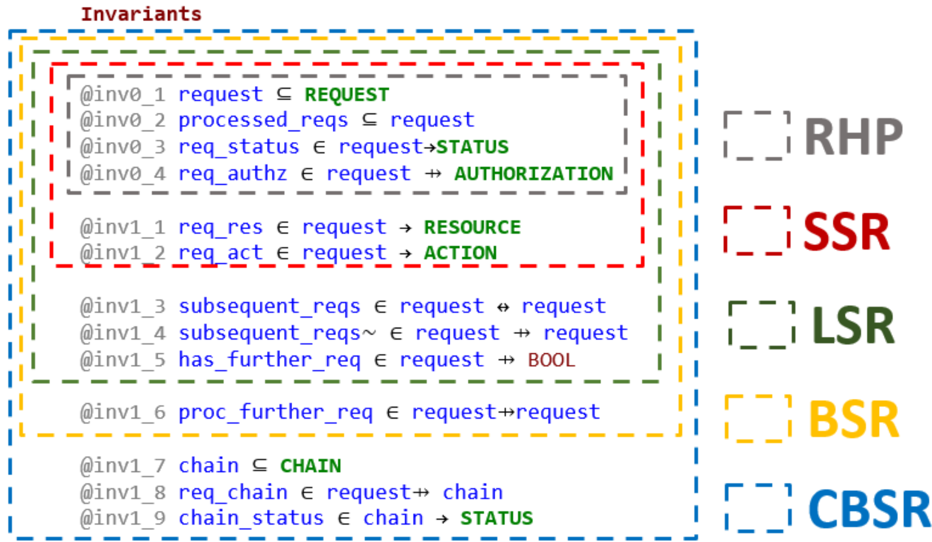


FIGURE 4.19: Invariants in RHP and ROP Pattern Model

Depending on the structure of the cloud components of a functionality, our proposed patterns offer an effective approach to model the functionality in Event-B. Figure 4.19 shows invariants in the RHP and ROP patterns.

Firstly, the Request Handling Pattern (RHP) represents the behaviour of a single request life cycle. In this pattern, the phase of a request during its execution life-cycle (*req_status* in *inv0_3*) and its authorization outcome (*req_authz* in *inv0_4*) are introduced. By refining the RHP patterns, ROP patterns are developed to model the functionalities of a serverless system.

Moreover, the Single Service Request (SSR) pattern, our most basic request ordering pattern, is effective for modelling functionalities that are met by the execution of a single request. Therefore, it is sufficient to track the phases of a request execution during its life cycle that are already introduced in abstraction (RHP). Because ROP patterns represent functionalities, the requested resource and the requested action are also introduced in *inv1_1* and *inv1_2*, respectively.

However, if the functionality requires multiple requests across different services, the order of these requests also requires tracking. In cases requiring multiple requests, the most basic structure could be a linear order of requests. To be fulfilled, each request may require input from only one further request to another service. For such structures where requests are ordered linearly, we developed the Linear Service Request (LSR) pattern. In the LSR Pattern, we track request execution to ensure the proper order of the requests (*subsequent_reqs* in *inv1_3*). A flag

is also associated with each request to identify whether it requires input from other services or not. (*has_further_req* in *inv1_5*).

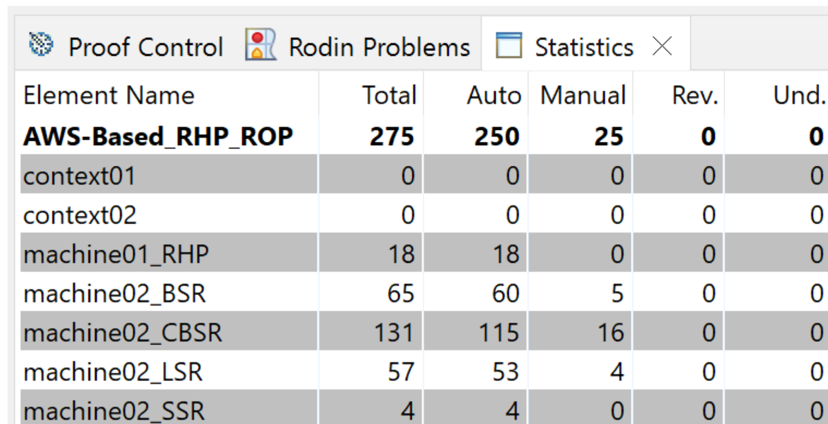
Moreover, if any request within a functionality requires inputs from multiple services, we propose the BSR pattern to model the functionality. In the BSR pattern, processed subsequent requests are kept (*proc_further_req* in *inv1_6*) to verify that all processed subsequent requests are successfully executed when initiating a new one.

Finally, for more complex cases where a request needs inputs from multiple services that also require inputs from other services, we propose the Chained Branching Service Request (CBSR) pattern. This pattern introduces the concepts of a chain (*req_chain* in *inv1_8*) and chain status (*chain_status* in *inv1_9*). The former concept defines chains for requests in different branches, while the latter defines the status of those chains. By determining the status of a chain, it can be ascertained whether a request in a chain (branch) fails or not.

4.3 POs of RHP / ROP Patterns

Proof obligations are logical condition that must be proved to ensure the correctness of the model. In this section, the statistics about discharging og POs in RHP and ROP patterns.

The table in Figure 4.20 shows proof statistics of RHP and ROP patterns. In the development of RHP pattern (machine01_RHP), 18 POs were generated by the Rodin tool, while in ROP patterns development, 4, 57, 65, and 131 more POs were generated for SSR(machine02_SSR), LSR(machine02_LSR), BSR(machine02_BSR), and CBSR (machine02_CBSR) pattern, respectively.


Proof Control | Rodin Problems | Statistics

Element Name	Total	Auto	Manual	Rev.	Und.
AWS-Based_RHP_ROP	**275**	**250**	**25**	**0**	**0**
context01	0	0	0	0	0
context02	0	0	0	0	0
machine01_RHP	18	18	0	0	0
machine02_BSR	65	60	5	0	0
machine02_CBSR	131	115	16	0	0
machine02_LSR	57	53	4	0	0
machine02_SSR	4	4	0	0	0

FIGURE 4.20: POs of RHP and ROP Patterns

All POs in RHP pattern were automatically proved. Because ROP formal patterns are developed by refining RHP formal pattern, POs in RHP development were inherited to ROP formal patterns development.

Therefore, as illustrated in Figure 4.20, Besides RHP POs in abstraction, all generated four POs automatically were proved in SSR development. Moreover, in LSR pattern development, 53 out of 57 POs were automatically proved by tool, while 60 out of 65 were automatically proved. Lastly, of 131 generated POs, 115 were automatically proved by ROdin tool, whereas 16 POs manually proved.

Chapter 5

Formal Patterns for Authorization Mechanism

In a cloud system, including a serverless system, to access a specific resource, a request must be made. The cloud system's acceptance or rejection of this request is contingent on whether the requester is authenticated and possesses the appropriate permissions. Subsequently, the cloud system will issue a response indicating either success or failure.

When a request is received by a cloud system, authentication is first checked. After the requester's identity has been authenticated, AWS checks for authorization to determine whether the authenticated entity has the necessary permissions to perform the requested action on the requested resources. However, as authentication is beyond the scope of our current model, we assume that requests that are sent by registered requesters, like existing app users or functions in the system, are already authenticated.

In this research, we focus on the authorization mechanism. Each request made in cloud a environment must go through the authorization mechanism, and only those requests with proper authorization are accepted. This makes the authorization mechanism crucial for ensuring the proper and secure functioning of a cloud-native system.

Several different cloud providers exist in the market, each with distinct offerings and no standardisation in terms of the services provided or the way to design a system with those services. In our research, we specifically focus on AWS services because they have the lion's share of the cloud computing market and there are more documents compared to other providers.

Moreover, a comprehensive explanation of the architecture of AWS-based serverless applications and the functioning of the authorization mechanism in AWS environment are provided in Section 2.6.

In this chapter, we introduce an Event-B formal modelling approach tailored for the authorization and access control mechanisms in an AWS environment.

5.1 Formal Patterns for Authorisation Mechanism

In this section, it shows how we model the authorization mechanism at a high abstract level and then refine our model to represent the authorization mechanism on the AWS cloud platform.

5.1.1 A Non-Deterministic Authorization Mechanism

The authorization mechanism in the AWS platform is very complex. We use the abstraction and refinement features of Event-B to manage the complexity of the authorization process. At first, we focus on the outcome of an authorization process, which is either to accept or reject a request.

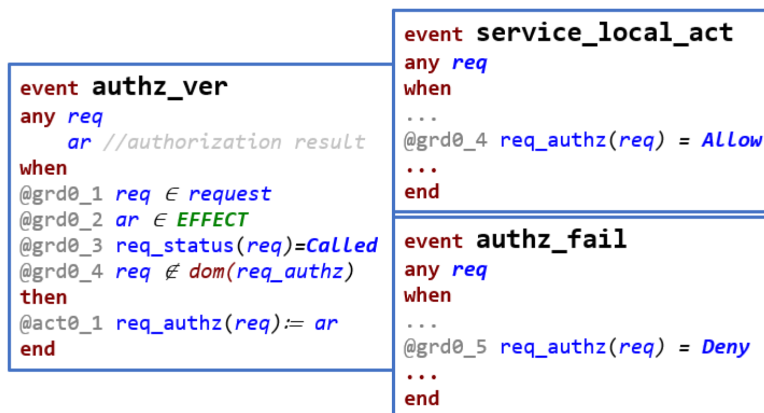


FIGURE 5.1: A Non-Deterministic Authorization Mechanism

In the abstract model, as represented in Chapter 4, the authorization mechanism is introduced in a non-deterministic way. *authz_ver* event in Figure 5.1 represents the authorization mechanism. In this event, the authorization outcome for a request is assigned non-deterministically by using the value of the parameter: *ar*. The type of *ar* is defined by *grd0_2* as EFFECT, a binary set whose value could be either Allow or Deny. The output of the *authz_ver* event will determine which event is enabled in the next stage. The success case (*service_local_act*) is enabled when the authorization result is *Allow*, whereas the result of *Deny* makes the failure case (*authz_fail*) enable.

$$@inv0_4 : req_authz \in request \rightarrowtail STATUS$$

The variable *req_authz* in *inv0_4* displays the results of the authorization. The relationship is a partial function because a request could only have an authorization result after passing through the authorization mechanism.

5.1.2 First Refinement: Introducing Sub-typing Generalization

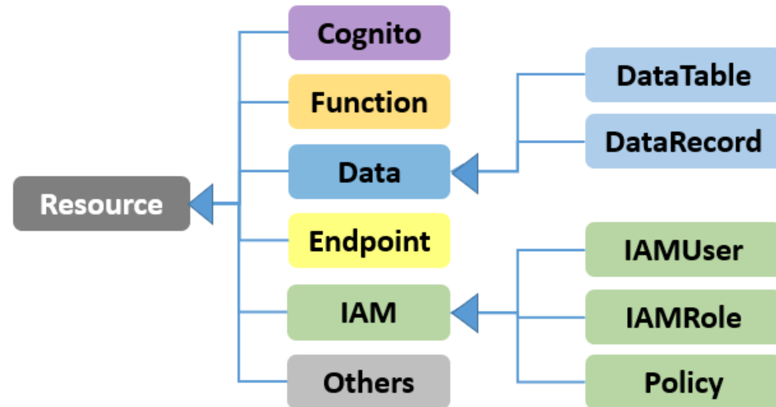


FIGURE 5.2: Resource and Resource Types

Everything within a cloud platform constitutes a resource, and a request is made for a resource. Therefore, to model the request processing uniformly, as shown in Figure 5.1, we introduce an abstract RESOURCE type. In the next stage, we use a sub-typing mechanism to introduce different resource types. This sub-typing approach allows us to specialise in the request processing associated with various resource sub-types. The *inv1_1* invariant defines the link between requests and the abstract resource type:

@inv1_1 : req_res $\in$ request $\rightarrow$ RESOURCE

To implement our subtyping approach of Figure 5.1 in Event-B, we use a partition operator. The *axm1_1* axiom states that Cognito, Function, Data, IAM, EndPoint, and OtherRes sets are subsets of the RESOURCE set. We incorporate the OtherRes set to enhance the model's flexibility, enabling the introduction of a new resource type as a subset of the OtherRes set. Then, in *axm2_2* and *axm5_1* axioms, we refine the Data and IAM subset further to represent the relevant subtypes.

axioms
@axm1_1 : partition (RESOURCE , Cognito , Function , Data , IAM , EndPoint , OtherRes)
@axm2_2 : partition (Data , DataTable , DataRec)
@axm5_1 : partition (IAM , IAMUser , IAMRole , IAMPolicy)

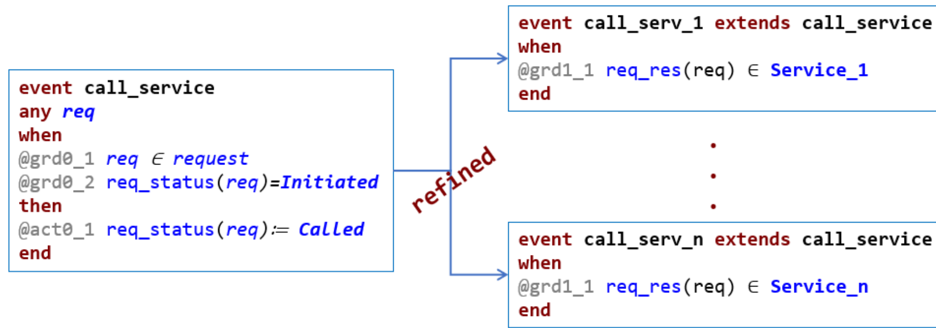


Figure 5.3: Refining Request Execution Based on Resource Types

The introduction of the sub-typing mechanism discussed earlier allows us to refine the abstract model to introduce more details about the operations of a serverless app. For example, in Figure 5.3, we see how the abstract service *call_service* is refined to include two new functionalities: calling service 1 and calling service 2 in a serverless app. The operations of a serverless app are not restricted to these two cases, but the aim here is to demonstrate the effectiveness of our approach to abstractly specifying the operations of a serverless app and refining it later to introduce more details.

Instead of subtyping using the partition operator, an alternative approach would have been to define each resource class as a distinct type. This approach would distinguish between requests made for each type of resource and specify their relations separately. Consequently, some concepts that we define with a single invariant, like *inv1_1* ($req_res \in request \rightarrow RESOURCE$), would have to be replaced with multiple invariants for different kinds of resources, such as data, function, and so on. This approach would also increase the complexity of the specification of request handling, as we would need to introduce a separate set of events for each request type. For example:

@inv3_4a : req_res_data ∈ request → DATA
@inv3_4aa : req_res_fun ∈ request → FUNCTION
...

5.1.3 Second Refinement: Replacing the Non-deterministic Authorizer with a Deterministic Authorizer

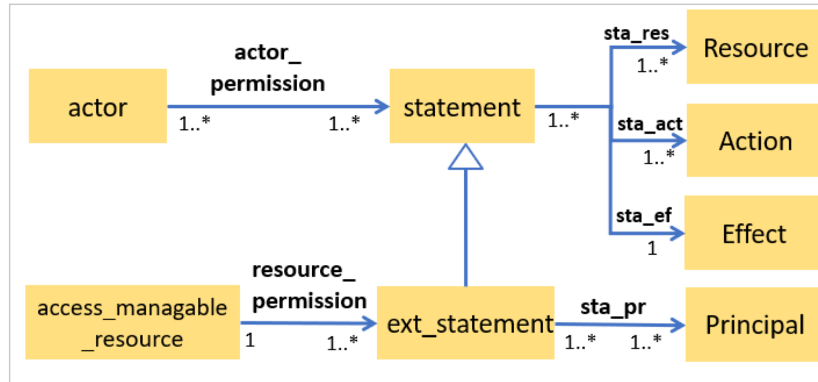


Figure 5.4: An abstraction of permission mechanism in AWS environment

To fulfil a request successfully, the requester must have permission to perform the requested access. As outlined in Section 2.6.2, the authorization mechanism in AWS is considerably complex. To manage this complexity, we devise an abstract Event-B specification of this access control and then refine it gradually to introduce all necessary details. The core idea of this abstraction is to define a permission relation that directly relates the actor/requester to the statement. This approach allows us to bypass some intermediary entities, such as the IAM role, described in Section 2.6.2, thereby simplifying our model. Figure 5.4 illustrates our abstraction of the permission mechanism in the AWS environment. It is worth noting that we have maintained the core idea of actor and resource-based permissions as described in Section 2.6.2.

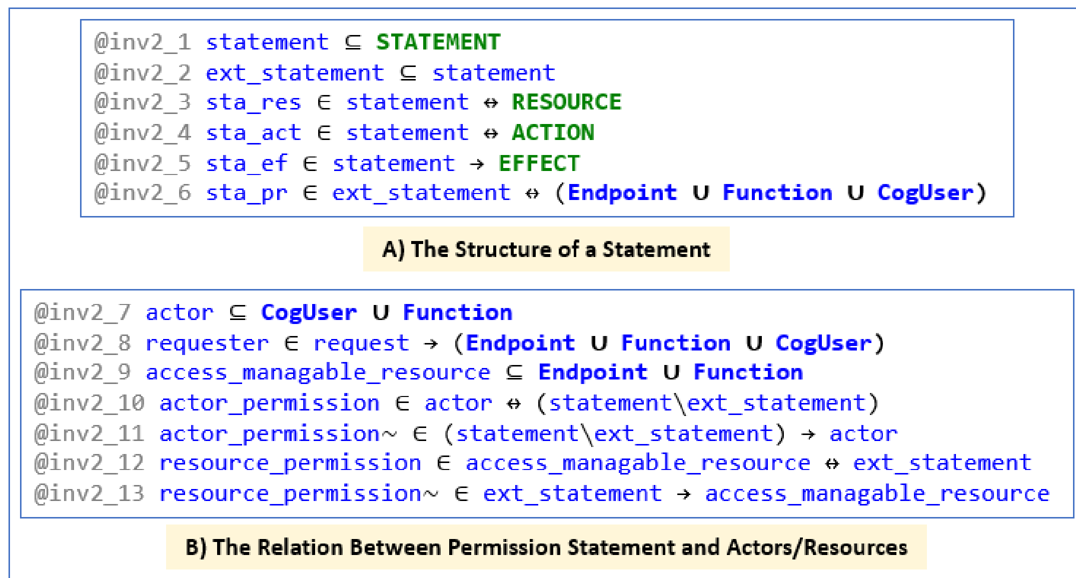


Figure 5.5: Event-B Invariants Representing Abstract Permission Mechanism

The formal specification of variables in Figure 5.4 is provided in the Event-B excerpt in Figure 5.5. These definitions reflect the structure of AWS entities we described earlier in Section 2.6.1. The structure of statements and resource-based statements are presented in Figure 5.5A. A statement in *inv2_1* introduces the existing permissions in the system. In the definition of a statement, what actions (*inv2_4*) are allowed or denied (*inv2_5*) on what resources (*inv2_3*) are introduced. If the statement is a resource-based statement (*inv2_2*), *sta_pr* shows the entities who are granted or restricted access permission in the statement.

Moreover, the relationships between various AWS entities and these statements are defined in Figure 5.5B. We used the *actor* variable to represent all AWS entities where they can be associated with a permission object. Additionally, we have a subset of resources with which a resource-based policy can be associated. We represent this subset by the *access_managable_resouce* variable. It should be noted that the *function* entity can be part of both sets. That means that a function can utilise both actor and resource-based permissions.

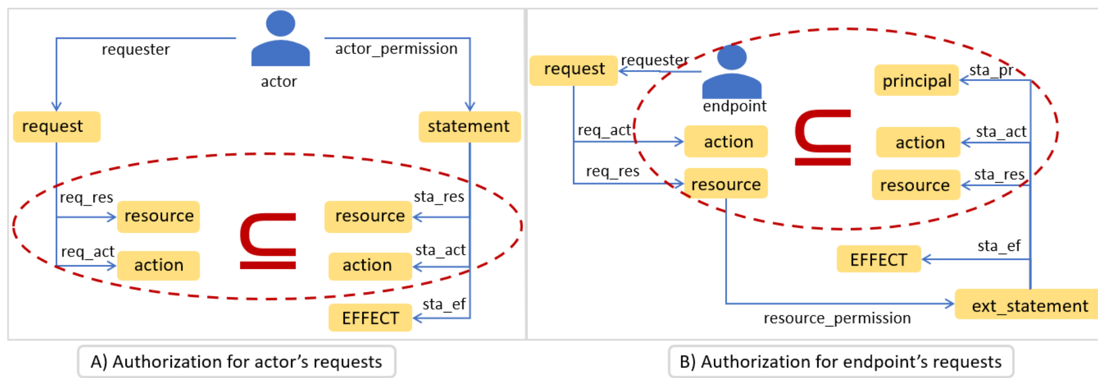


FIGURE 5.6: Authorization of a request

Introducing concrete variables in the access control system paves the way to refine the authorization rules. A graphical illustration of the authorization mechanism is provided in Figure 5.6. Each request is associated with a requester, and its abstract form is about performing an action on a resource. A request will only be allowed if the associated requester actor has permission to carry out the requested action on the requested resource. In this regard, the actions-resources pairs in the request (*req_act* , *req_res*) must be a subset of permitted resources-actions pairs in the permission object (*sta_act* , *sta_res*).

The permission can either be actor-based permission, directly associated with the *actor* captured by *actor_permission*, or it can be resource-based permission, which defines which actor is allowed to perform the requested action on the resource and is represented by the *resource_permission*. These two cases are presented in Figure 5.6A and Figure 5.6B. An example of a resource-based policy is an API Gateway endpoint that triggers a function execution by sending a request to the Lambda service. In this case, the requested function must possess a resource-based statement that explicitly allows the endpoint to execute this function.

```
@inv2_14 ∀r·r∈request ∧ req_authz(r)=Allow ∧ requester(r)∈ actor ⇒
        (∃s·s∈statement ∧  requester(r)↦s ∈ actor_permission ∧
             s↦req_res(r) ∈ sta_res ∧
             s↦req_act(r) ∈ sta_act ∧
             sta_ef(s) = Allow)

@inv2_15 ∀r·r∈request ∧ req_authz(r)=Allow ∧ requester(r)∈ endpoint ⇒
        (∃s·s∈statement ∧ req_res(r)↦s ∈ resource_permission ∧
            s↦req_res(r) ∈ sta_res ∧
            s↦req_act(r) ∈ sta_act ∧
            s↦requester(r)∈ sta_pr ∧
            sta_ef(s) = Allow)
```

FIGURE 5.7: Event-B Encoding of the Request Authorization

The "EFFECT" element in both types of permission objects is a mechanism to define the permission in positive or negative terms and, therefore, can take the value of allow or deny, respectively. The Event-B encoding of the conditions we have illustrated graphically in figures 5.6A and 5.6B are presented in *inv2_14* and *inv2_15* in Figure 5.7, respectively.

After introducing the necessary variables and invariants to represent the authorization constraints, now is the right time to refine the abstract authorizer of Section 5.1.1. The aim is to replace the non-deterministic version of the authorizer presented in Figure 5.1 with a more deterministic one.

As previously discussed, the initiator of a request could be either an actor or an endpoint. The authorization process must distinguish between these cases because different types of permissions are used for each case. For instance, resource-based permissions should be used when the initiator is an endpoint. The authorization process can consist of several cases depending on the type of request initiator, the presence or absence of the permission object, and the possible values of the "EFFECT" element within the permission. These cases might include:

1. The requester is an *actor* and the actor has an *allow* statement for the request.
2. The requester is an *actor* and the actor has a *deny* statement for the request.
3. The requester is an *actor* and the actor does not have any statement for the request.
4. The requester is an *endpoint* and the requested resource has an *allow* statement to allow the endpoint to make a request.
5. The requester is an *endpoint* and the requested resource has a *deny* statement to reject the endpoint to make a request.
6. The requester is an *endpoint* and the requested resource does not have any statement about the endpoint to make a request.

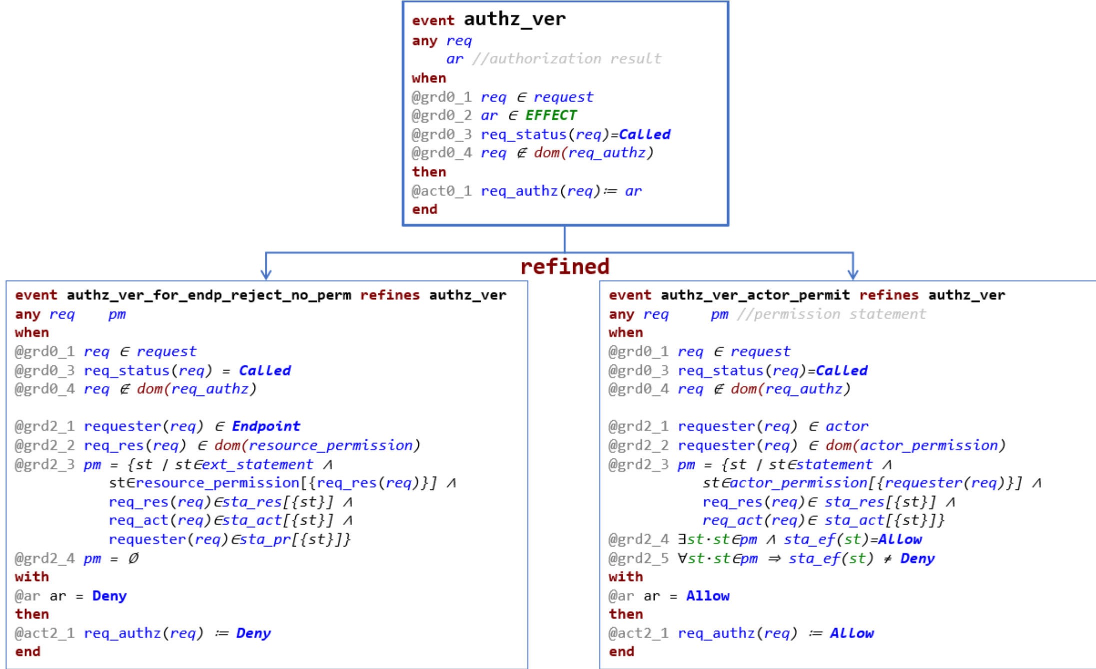


FIGURE 5.8: Deterministic Authorizer Events

In Figure 5.8, we cover case 1, where the requester is an actor having an appropriate statement, and case 6, where the requester is an endpoint having no related statement. Due to space constraints, we do not cover the model representations of other cases in this text.

The *authz_ver_actor_permit* event detailed in Figure 5.8 specifies the case that the requester is an actor having an *allow* value in his/her statement. Guard *grd2_1* in the *authz_ver_actor_permit* event ensures that the requester is an actor, while *grd2_3* states that *pm* comprises a set of statements that includes all the requester's statements that are related to the requested resource and action. Moreover, *grd2_4* ensures the presence of at least one "Allow" statement in the *pm*, authorising the performance of the requested action on the requested resource. Simultaneously, *grd2_5* ensures that there are no "Deny" statements in the *pm* that would prohibit performing the requested action on the requested resource.

The*authz_ver_for_endp_reject_no_perm* event represents the case where the requester is an endpoint and there is no related statement, resulting in the request being rejected. Among the statements (resource-based statements) associated with the requested resource, guard *grd2_3* in this event puts all statements that include the requester, the requested resource, and action together in the *pm* set. Simultaneously, *grd2_4* ensures that the *pm* set is empty.

The witness in events illustrated in Figure 5.8 is the mechanism enabling us to transform the abstract, non-deterministic authorization system into a deterministic one. In these refined events, the abstract parameter *ar*, which makes the authorizer non-deterministic, is replaced with an authorization result based on permissions associated with the requester.

5.1.4 Third Refinement: The complete Access Control in AWS

As mentioned in Section 2.6.2, the AWS authorization mechanism uses IAM users, IAM roles, and IAM policies to manage access to the cloud resources. Policies incorporate an array of statements that dictate the execution of each issued request. In this refinement to provide a complete representation of the AWS authorization mechanism, we refined our previous model by introducing IAM roles, IAM policies, and their relations with other entities.

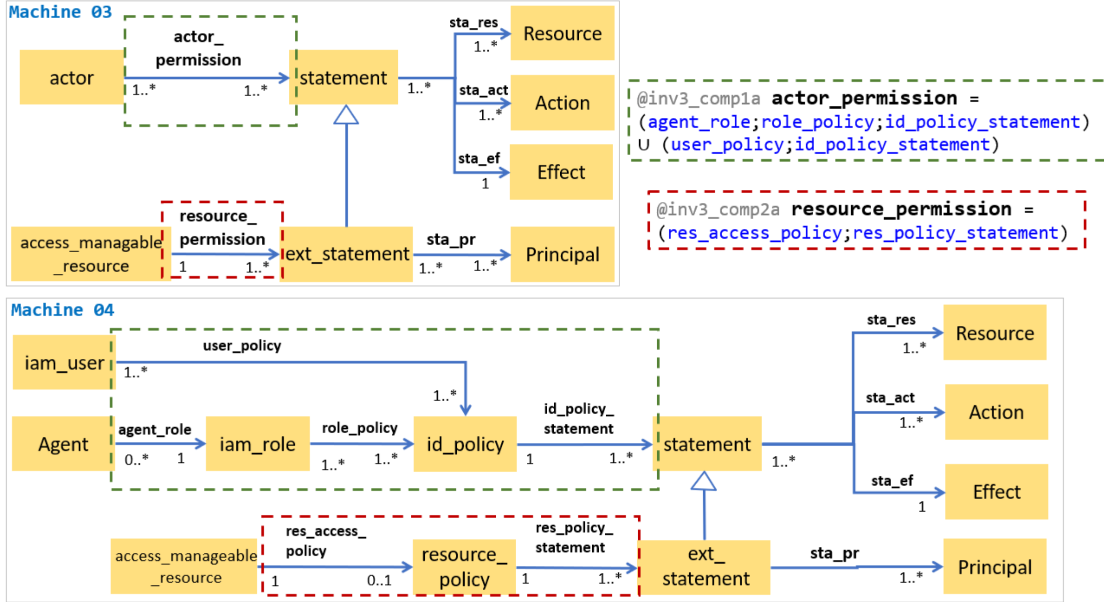


Figure 5.9: Introducing IAM user, IAM role and IAM policy

The core idea in this refinement is to replace the *actor_permission* and *resource_permission* relations with more concrete versions that encompass AWS entities such as IAM users and IAM roles. Figure 5.9 provides a graphical illustration of the abstract and the concrete relationships. The provided *inv3_comp1a* and *inv3_comp2a* define the composition relation between the abstract and refined ones.

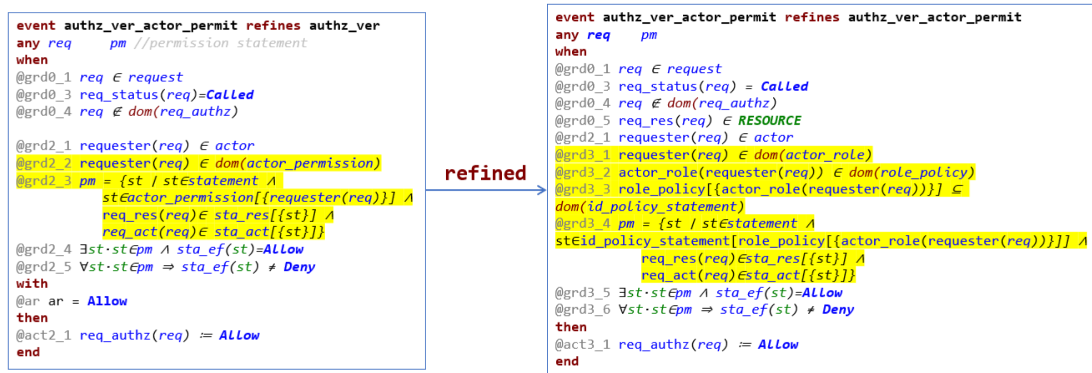
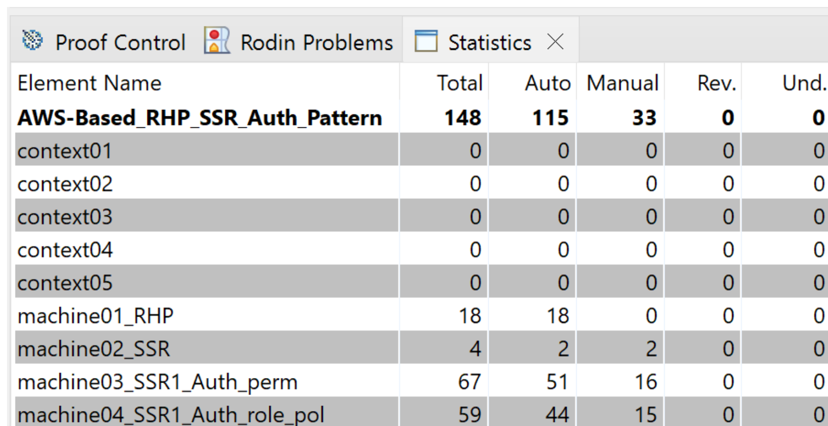


Figure 5.10: The Effect of New Features on Authorization Events

Refining the abstract authorization mechanism and introducing the concrete features, as presented in Figure 5.9 also affect authorizers' events. For instance, Figure 5.10 shows how introducing role and policy affect the *authz_ver_actor_permit* event in the refinement step. An actor has a role that is linked to policies defining its access level. Therefore, as can be seen in Figure 5.10, *grd2_3* guard in the abstract event, which represents the requester's permission statements related to the requested resource action, is refined into *grd3_4* in the refined event by using new concrete features like role and policy. Therefore, *grd3_4* guard in the refined event ensures that *pm* includes all statements of policies of the actor's role that are related to the requested resource and action.

5.2 POs of Authorization Mechanism Model

The authorization mechanism is introduced in two refinement steps by refining *authz_ver* event in RHP and ROP patterns. Moreover, ROP patterns are developed by refining the RHP pattern. To ensure the correctness of the authorization model and the consistency between refinement steps, all proof obligations (POs) that are logic conditions must be discharged.


Proof Control | Rodin Problems | Statistics

Element Name	Total	Auto	Manual	Rev.	Und.
AWS-Based_RHP_SSR_Auth_Pattern	**148**	**115**	**33**	**0**	**0**
context01	0	0	0	0	0
context02	0	0	0	0	0
context03	0	0	0	0	0
context04	0	0	0	0	0
context05	0	0	0	0	0
machine01_RHP	18	18	0	0	0
machine02_SSR	4	2	2	0	0
machine03_SSR1_Auth_perm	67	51	16	0	0
machine04_SSR1_Auth_role_pol	59	44	15	0	0

FIGURE 5.11: POs of Authorization Mechanism's Model

Figure 5.11 illustrates the PO statistics in the development of a refinement strategy for the model that introduces the authorization mechanism in the AWS cloud environment. As detailed in previous sections, to show the refinement strategy for formal modelling of the authorization mechanism, the SSR pattern is used to refine. As illustrated in Figure 5.11, 148 POs were generated, 115 of which were automatically proved, while 33 of which were manually proved.

5.3 Conclusion

To conclude, cloud-based systems predominantly allow users to access resources over the network. To safeguard resources and ensure secure access, it is essential to clearly define the identities and permissions of those identities attempting to access the system. Otherwise, unauthorised access may cause significant loss in the system. Therefore, authorzation, which mainly manages user actions on a set of resources in a system [44], is a key mechanism to make a system more secure in any cloud concept, including serverless architectures.

To design and configure access control mechanisms properly, formal methods may help cloud native app developers. Moreover, in the formal modelling pattern proposed in this section, we develop invariants and guards to identify any conflicts in the access control mechanism and rectify them.

Chapter 6

Case Studies

In the previous chapters, we presented our formal patterns for modelling a functionality of a serverless system (Chapter 4) and the incremental introduction of the authorization mechanism specific to serverless applications in the AWS environment (Chapter 5). These patterns aim to manage the complexity of the modelling process and enhance the robustness of the authorization mechanism in the final model.

In this chapter, by using proposed patterns (Chapter 4 and 5), we modelled three different case study scenarios from two different domains. The first two scenarios are from the 'Project Management System' case study, while the third scenario is from the 'Learning Management' case study.

To model these scenarios, we first selected the appropriate ROP pattern. Then scenario-specific features were integrated into the pattern to adapt it to the particular scenario. Lastly, the authorization mechanism is introduced, as detailed in Chapter 5, in the model.

The aim is to demonstrate the usability of our patterns in modeling various scenarios within the same domain as well as different scenarios across different domains.

6.1 Case Study: Project Management System

Project Management System is a system to enable a companies or institution to create and manage projects. The system is characterised by employee, project, department, and role. Each employee is introduced by an application user, whereas roles define the access level of a user to the system. Moreover, user roles are admin, project manager, department manager, and developer.

General requirements for the Project Management System are shown in Table 6.1, whereas requirements for admins, department manager, project manager, and developer are illustrated in Table 6.2, Table 6.3, Table 6.4, and Table 6.5, respectively.

R1	Each employee work at only one department.
R2	Each project is run under one department.
R3	Each project has a status to define its completion level.
R4	Status of a project can be changed by its project manager.
R5	Each user has a role that determines his/her access level in the system.
R6	Those roles are: admin, project manager, department manager, and developer.

Table 6.1: General Requirements for Learning Management System

R7	An admin can add/delete a user to/from the system.
R8	An admin can add/delete a project to/from the system.
R9	An admin can add/delete a department to/from the system.
R10	An admin can add/delete a role to/from the system.
R11	An admin can assign a user to a department as a department manager.
R12	An admin can view the personal information of any user.

Table 6.2: Requirements of Admins

R13	A Department Manager can add/delete a user to/from his/her Department.
R14	A Department Manager can allocate a employee to a project in his/her Department.
R15	A Department Manager can allocate a employee to a project as Project Manager in his/her Department.
R16	A Department Manager can list department, employee, and project information in his/her department.

Table 6.3: Requirements of Department Managers

R17	A Project Manager can allocate a developer to his/her project.
R18	A Project Manager can remove a developer from his/her project.
R19	A Project Manager can update the information of his/her project.
R20	A Project Manager can list all developer's information in his/her project.

Table 6.4: Requirements of Project Managers

R21	A Developer can update his/her information.
R22	A Developer can list his/her information.
R23	A Developer can list the project s/he attend.

Table 6.5: Requirements of Developers

The project management system that is characterised by employee, project, department, and role entities is built in the AWS environment. Figure 6.1 depicts the structure of the system in the AWS cloud environment.

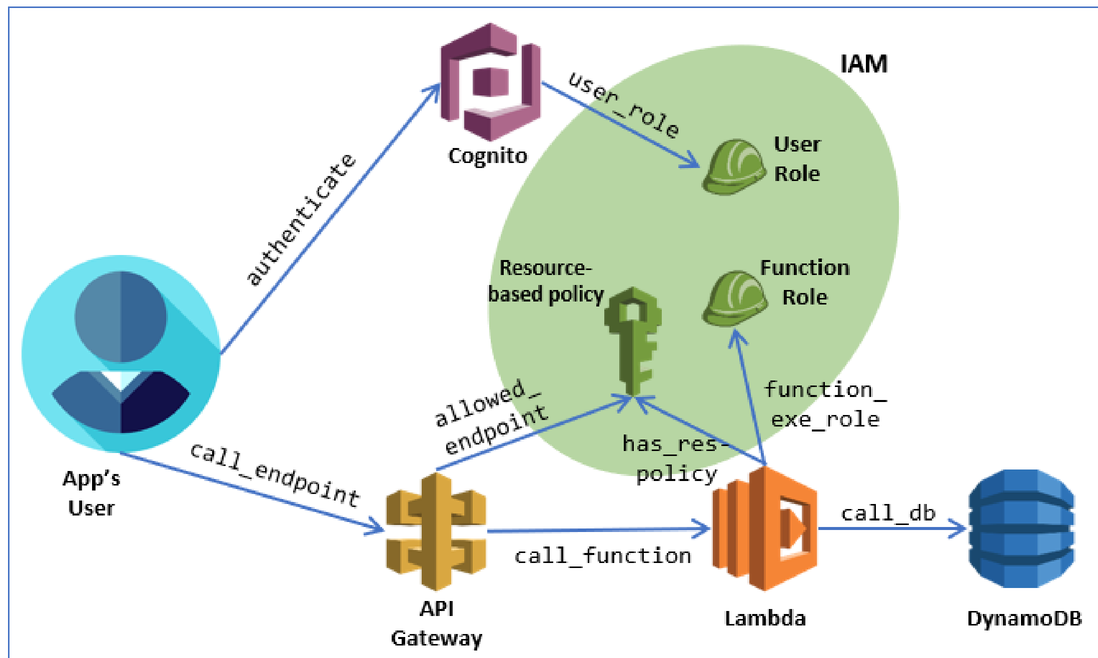


FIGURE 6.1: A Basic Structure of a Web App with Serverless Architecture in AWS Environment

The functionalities that we have chosen are from a *"project management system"* built on the AWS environment, as shown in Figure 6.1. The system manages application users through the Cognito service, associating each user with an IAM role as a user role, defining their access level. These access levels introduce which users can execute which API endpoints in the API Gateway. Each of these API endpoints is designed to satisfy a distinct functionality of the *"project management system"*. One such endpoint, *EPPromoteUser*, when executed by an authorised user, triggers the corresponding Lambda function (a serverless function in the AWS context) to fulfil the request. To execute a lambda function, the requester must either have an IAM role granting the necessary permissions or be granted permission by the function's resource-based policy. If the requester is an API endpoint in the API Gateway, permission to execute the function must be granted by the function's resource-based policy. This is because an API endpoint cannot use an IAM user, requiring permission to be granted directly to the requester endpoint. When the function is executed, it may need to access resources from different services, such as database tables in DynamoDB. Each lambda function is linked to an IAM role known as the function execution role, which defines the access level of the associated function. Therefore, a function can access those resources that are allowed by its execution role.

6.1.1 Scenario 1: Updating Project Status

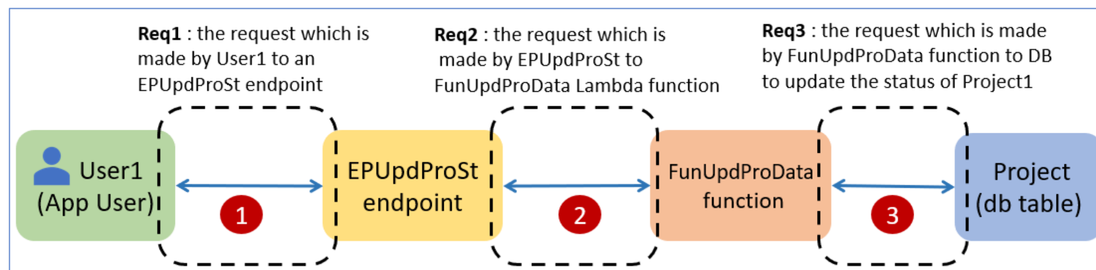


FIGURE 6.2: The Structure of "Updating Project Status" Functionality

The scenario: User1, with a Project Manager role, wants to update the status of Project1, a project that he manages in the project management system. The structure of the "updating project status" functionality in the AWS environment is illustrated in Figure 6.2. The following steps are executed to fulfil this functionality:

1. User1 initiates a request to update the status of Project1.
2. The client app sends User1's request to the API Gateway service.
3. The API Gateway receives the request.
4. The authorizer of the EPUpdProSt endpoint determines whether User1 has permission to execute EPUpdProSt endpoint, the endpoint associated with "updating project status" functionality.
5. If User1 has the proper permission, the EPUpdProSt endpoint is executed.
6. When the requested endpoint is executed successfully, the endpoint calls the FunUpdProData lambda function.
7. The authorizer of the FunUpdProData function determines whether the EPUpdProSt endpoint is allowed to execute the function in the function's resource-based policy.
8. The FunUpdProData function is executed if the EPUpdProSt endpoint is permitted.
9. The executed FunUpdProData lambda function sends a request to DynamoDB to update the status of Project1.
10. The authorizer of Project table determines whether the FunUpdProData function has permission to update the requested table.
11. The status of Project1 is updated if the FunUpdProData function is allowed.

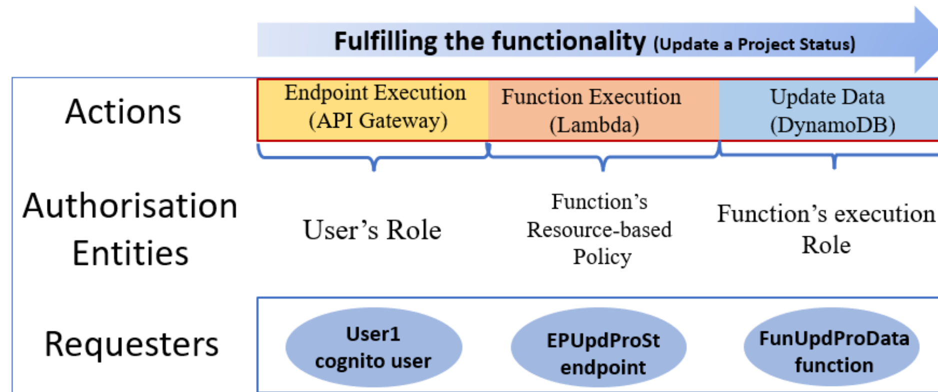


FIGURE 6.3: The Authorisation entities in "Updating Project Status" Functionality

Furthermore, Figure 6.3 clearly depicts the authorization entities involved, such as IAM roles or policies, and their respective responsibilities during the fulfilment of the "updating project status" functionality.

6.1.1.1 Model "updating project status" Functionality

As illustrated in Figure 6.2, the structure of the "updating project status" functionality in the AWS environment is well-suited for implementing our LSR pattern, as explained in Section 4.2.2, to model in Event-B.

To fulfil "updating project status" functionality, three distinct requests must be executed in linear order. Figure 6.4 illustrates the implementation of the LSR pattern to model the "updating project status" functionality as a case study. As shown in the figure, LSR(ue)[1], LSR(uf)[2], and SSR(pt)[3] introduce requests sent to the EPUpdProSt endpoint, the FunUpdProData function, and the Project database table, respectively. Therefore, as depicted in Figure 6.4, to update Project1's status, the FunUpdProData lambda function, whose execution depends on the successful execution of the EPUpdProSt endpoint, must be successfully executed.

[1] ue : EPUpdProSt endpoint
[2] uf : FunUpdProData function
[3] pt : Project database table

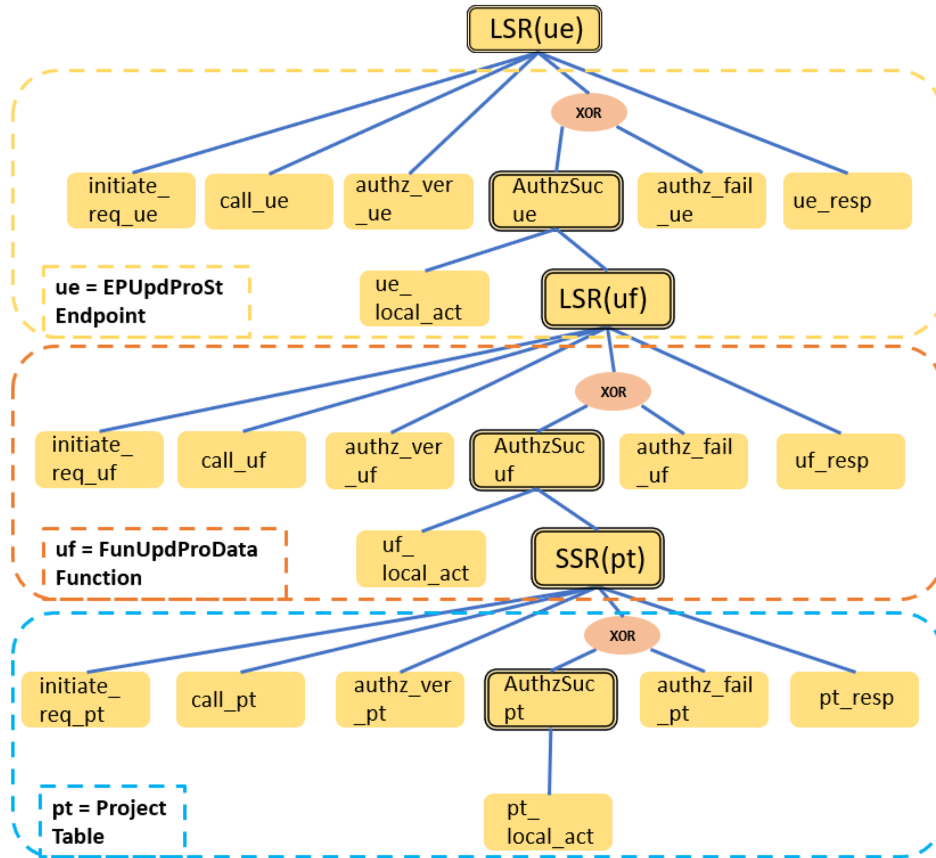


FIGURE 6.4: The tree-like representation of "updating project status" functionality

Textual Representation 6.1 illustrates the implementation of the LSR pattern for modelling the "updating project status" scenario in textual form. The textual representation helps us in analysing the model in Figure 6.4 in terms of an RHP pattern.

LSR(ue) =
RHP(ir_ue, c_ue, av_ue, la_ue, LSR(uf), af_ue, r_ue)
LSR(uf) =
RHP(ir_uf, c_uf, av_uf, la_uf, SSR(pt), af_uf, r_uf)
SSR(pt) =
RHP(ir_pt, c_pt, av_pt, la_pt, SKIP, af_pt, r_pt)

(6.1)

The abbreviation in the Textual Representation 6.1 means the following :

ir_ = initiate request to
c_ = call
av_ = authorisation verification of

la_ = local action in
af_ = authorisation failure of
r_ = response of

Therefore, the abbreviations of events in the RHP pattern of *LSR(eu)* mean :

ir_ue = initiate request to EPUpdProSt endpoint
c_ue = call EPUpdProSt endpoint
av_ue = authorization verification of EPUpdProSt endpoint
la_ue = local action in EPUpdProSt endpoint
af_ue = authorization failure of EPUpdProSt endpoint
r_ue = response of EPUpdProSt endpoint

Finally, the requests are required for fulfilment of the "updating project status" functionality map to entities in the pattern. Table 6.6 offers a lucid mapping between the requests and their associated entities within the pattern and the system:

In Pattern	In The System	Request
LSR(ue)	The request that is sent EPUpdProSt Endpoint to execute	req1
LSR(uf)	The request that is sent FunUpdProData Function to execute	req2
SSR(pt)	The request that is sent DynamoDB to update project table (project status)	req3

Table 6.6: Requests During the Process of Scenario 1

6.1.1.2 Event-B Model of The Scenario

First of all, in the first context (context01), the features that are required for the RHP pattern (request, request status, request authorization), as detailed in Figure 4.4, are defined. And then, to model "updating project status" functionality, as illustrated in Figure 6.2, users, endpoints, functions, and data, which are specific to the AWS system, must be introduced as different resources in the context, as shown in Figure 6.5.

Moreover, actions are also defined as carrier set in the context, as illustrated in Figure 6.5. Therefore, the requested action could be represented. The *axm1_2* axiom defines some specific actions that are required for case study functionality fulfilment. *InvokeApi* represents the action of an API execution, while *ExecFun* shows the action of a function execution. Similarly, *UpdItemDB* depicts the action of updating an existing item in a database table.

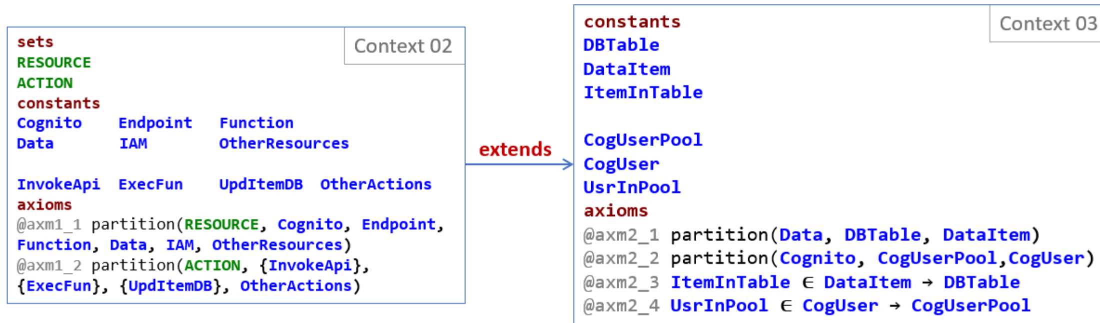


FIGURE 6.5: Features Specific to AWS system

The context 03 in Figure 6.5 introduces and details resources in Cognito and a database. In a database, we have database tables and data records (*axm2_1*) and each data record belongs to a database table (*axm2_3*). Likewise, Cognito, which is a restricted database service for user management, authentication, and authorization, comprises cognito user pools and cognito users. Cognito users represent application users in the application domain (*axm2_2*), each cognito user (app user) belongs to a cognito user pool (*axm2_4*).

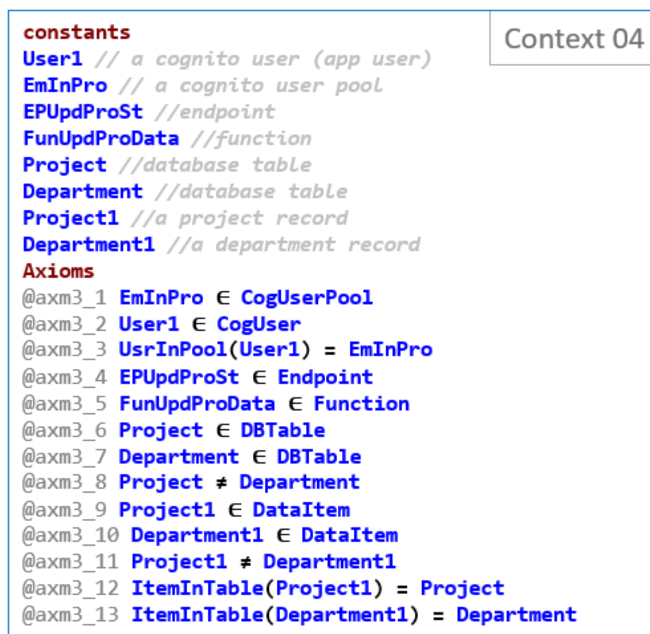


FIGURE 6.6: Features Specific to Case Study System and Scenario

Furthermore, Context 04 context in Figure 6.6 introduces features that are related to our case study system (project management system), specifically focusing on the 'updating project status' scenario.

Figure 6.7 shows the invariants in the abstract machine of the model for "updating project status functionality. Invariants in Figure 6.7A are coming from the RHP pattern, detailed in Section 4.1. Figure 6.7B shows invariants related to the LSR pattern that is developed by refining the RHP pattern, as explained in Section 4.2.2.

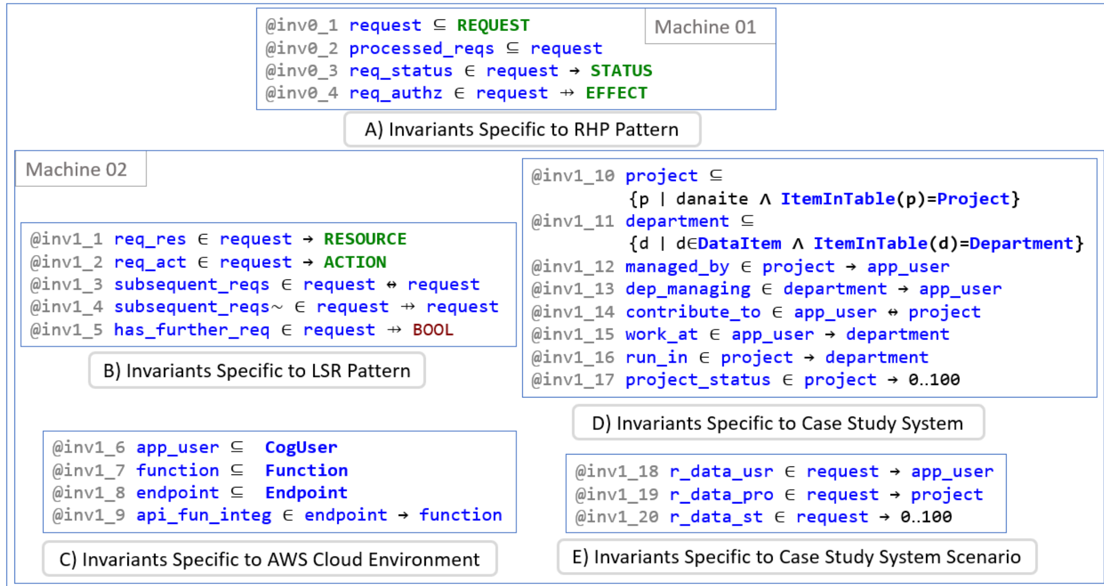


FIGURE 6.7: Invariants of Abstraction for Scenario 1

Additionally, Figure 6.7C shows invariants to introduce features specific to the AWS cloud environment. For instance, *inv1_9* states that each API Gateway endpoint is associated with a lambda function that can be triggered when the endpoint is executed. The invariants in Figure 6.7D mainly introduce application-specific features relevant to our case study system. For example, invariants *inv1_10* and *inv1_11* define the application specific entities, which are "projects" and "departments". The user is defined in *inv1_6*. Moreover, *inv1_14* depicts that multiple users can work on various projects, whereas *inv1_12* clarifies that each project is managed by a single user. Each project has a status, showing its completion percentage (*inv1_17*). Lastly, a user or a project is associated with only one department (*inv1_15* and *inv1_16*, respectively).

Furthermore, the invariants in Figure 6.7E represent data in the payload of a request. The r_data_usr variable (*inv1_18*) shows the application user initiating a request to update a project's status, whereas the r_data_pro variable (*inv1_19*) depicts the targeted project. Moreover, r_data_st (*inv1_20*) shows the new updated status value.

The model also includes certain setup features specific to our case study system. These features are assumed to be implemented during the application setup stage. Notably, the setup configurations, such as the association of each endpoint with a lambda function (*act1_8*), are introduced in the 'Initialisation' event illustrated in Figure 6.8.

```
event INITIALISATION
then
...
@act1_5 app_user ≔ {User1}
@act1_6 endpoint ≔ {EPUpdProSt}
@act1_7 function ≔ {FunUpdProData}
@act1_8 api_fun_integ ≔ {EPUpdProSt↦FunUpdProData}
@act1_9 project ≔ {Project1}
@act1_10 department ≔ {Department1}
@act1_11 managed_by ≔ {Project1↦User1}
@act1_12 dep_managing ≔ {Department1↦User1}
@act1_13 contribute_to ≔ {User1↦Project1}
@act1_14 work_at ≔ {User1↦Department1}
@act1_15 run_in ≔ {Project1↦Department1}
@act1_16 project_status ≔ {Project1↦0}
...
end
```

FIGURE 6.8: Initialisation for Scenario 1 Abstraction

In addition to the overall system setup, the case study scenario requires particular configurations. These are also detailed in the Initialisation event in Figure 6.8. For instance, *act1_5* introduces the registration of User1, while *act1_11* allocates User1 to Project1 as a project manager. Significantly, in Figure 6.8, the configurations highlighted in yellow pertain to the necessary configurations for the case study system (*Project Management System*). In contrast, the remaining configurations are specific to the case study scenario, which is the "updating project status" functionality.

```
extends init_req_ue extends
init_req
any res    act
    rdu    rdp    rds
when
@grd1_1 res = EPUpdProSt
@grd1_2 act ∈ ACTION
@grd1_3 rdu ∈ app_user
@grd1_4 rdp ∈ project
@grd1_5 rds ∈ 0..100
then
@act1_1 req_res(new_req) ≔ res
@act1_2 req_act(new_req)≔ act
@act1_3 r_data_usr(new_req) ≔ rdu
@act1_4 r_data_pro(new_req) ≔ rdp
@act1_5 r_data_st(new_req) ≔ rds
end
```

A) Initiate Request to EPUpdProSt Endpoint

```
event init_req_uf extends init_req
any pre_req    res  act
when
@grd1_1 res = FunUpdProData
@grd1_2 act ∈ ACTION
@grd1_3 pre_req ∈ request
@grd1_4 req_res(pre_req) = EPUpdProSt
@grd1_5 req_status(pre_req) = Succeded
@grd1_6 pre_req ∈ dom(has_further_req)
@grd1_7 has_further_req(pre_req) = TRUE
@grd1_8 req_res(pre_req)↦res ∈ api_fun_integ
then
@act1_1 req_res(new_req) ≔ res
@act1_2 req_act(new_req)≔ act
@act1_3 subsequent_reqs ≔ subsequent_reqs ∪
                        {pre_req↦new_req}
@act1_4 r_data_usr(new_req)≔r_data_usr(pre_req)
@act1_5 r_data_pro(new_req)≔r_data_pro(pre_req)
@act1_6 r_data_st(new_req)≔ r_data_st(pre_req)
end
```

B) Initiate Request to FunUpdProData Function

```
event init_req_pt extends init_req
any pre_req   res   act
when
@grd1_1 res = Project
@grd1_2 act ∈ ACTION
@grd1_3 pre_req ∈ request
@grd1_4 req_res(pre_req) = FunUpdProData
@grd1_5 req_status(pre_req) = Succeded
@grd1_6 pre_req ∈ dom(has_further_req)
@grd1_7 has_further_req(pre_req) = TRUE
then
@act1_1 req_res(new_req) ≔ res
@act1_2 req_act(new_req)≔ act
@act1_3 subsequent_reqs ≔subsequent_reqs ∪
                        {pre_req↦new_req}
@act1_4 r_data_usr(new_req)≔r_data_usr(pre_req)
@act1_5 r_data_pro(new_req)≔r_data_pro(pre_req)
@act1_6 r_data_st(new_req)≔ r_data_st(pre_req)
end
```

C) Initiate Request to Project DB Table

FIGURE 6.9: Request Initiation Events for Scenario 1 Abstraction

Figure 6.9 illustrates how the generic initiate request events in the RHP pattern are refined into specific request initiation events for a given functionality in the LSR pattern format. Events in Figure 6.9A, Figure 6.9B, and Figure 6.9C represent the initiation of a request to the EPUpdProSt endpoint, the FunUpdProData lambda function, and the Project DB table, respectively. The *grd1_1* in *init_req_ue*, *init_req_uf*, and *init_req_pt* specify the target resource of requests. For example, the *grd1_1* in *init_req_ue* says the request is initiated for the EPUpdProSt endpoint.

As shown in Textual Representation 6.1, the process to fulfil the "updating project status" functionality begins with initiating a request to the EPUpdProSt endpoint (*init_req_ue*). In the

init_req_ue event, the details about the functionality, including the user attempting the update (*rdu*), the project targeted for update (*rdp*), and the new status value (*rds*), are put in the request payload (request body) (*act1_3*, *act1_4*, and *act1_5*, respectively). Then, when a new request is initiated during the process, this information is subsequently transferred to the new request, as seen in the *init_req_uf* and *init_req_pt* events (*act1_4*, *act1_5*, and *act1_6*).

Additionally, as each endpoint is associated with a lambda function, the lambda function can be executed when the associated endpoint is successfully executed. This association is captured in the *grd1_8* guard in Figure 6.9B. The *grd1_8* guard ensures that the requested function (*res*) is associated with the requester endpoint (*req_res(pre_req)*).

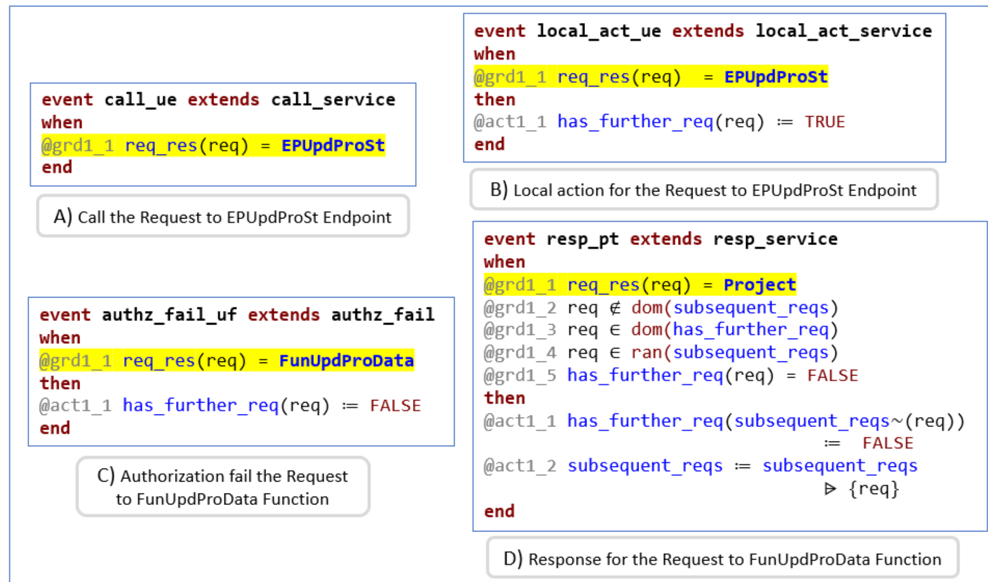


FIGURE 6.10: Some Events in Scenario 1 Abstraction

Figure 6.10 illustrates various events in the Event-B model of "updating project status" functionality. As depicted in Figure 6.10, the events in each RHP pattern of different requests can be distinguished by determining the value of *req_res(req)*, which shows the target resource of a request. The guards that are highlighted in yellow in Figure 6.10 define the target resource of the request. For example, the *grd1_1* guard in Figure 6.10A distinguishes *call_ue* from other calling service events, saying that one in Figure 6.10A is related to the request targeting the EPUpdProSt endpoint. In each event, the *grd1_1* guard specifies its corresponding request, which can be seen in other events in Figure 6.10.

```
event local_act_pt extends
                       local_act_service
any prj //project
    st //status
when
@grd1_1 req_res(req) = Project
@grd1_2 prj = r_data_pro(req)
@grd1_3 st = r_data_st(req)
@grd1_4 r_data_usr(req) = managed_by(prj)
then
@act1_1 has_further_req(req) ≔ FALSE
@act1_2 project_status(prj) ≔ st
end
```

FIGURE 6.11: Local Action Event Project DB Table

After successful execution of the requests to both the EPUpdProSt endpoint and the FunUpdProData function, the FunUpdProData function makes a request to the Project DB table to update the status of the requested project. The *local_act_pt* event in Figure 6.11 represents the process performed in the DynamoDB database following acceptance of the request to the project DB table.

Therefore, when the *local_act_pt* event is executed, the requested project's status is updated (*act1_2*). This update relies on data from preceding requests, specifically *r_data_pro(req)* (*grd1_2*) and *r_data_st(req)* (*grd1_3*). Moreover, *grd1_4* ensures that the user requesting the status update is the project manager of the relevant project.

Implementing Deterministic Authorization Mechanism in Scenario 1 :

As mentioned before, in AWS philosophy, no service does not inherently have the right to perform any actions on the resources of other AWS services. Reflecting this principle, specific permissions must be granted for each request in the system. In the context of our scenario, where the scenario of "updating project status" is being fulfilled, several permissions must be carefully configured.

Firstly, the application user (User1) must be granted permission to execute a specific endpoint (the EPUpdProSt endpoint) in the API Gateway. Following this, the executed endpoint itself requires permission to initiate the corresponding Lambda function (the FunUpdProData Lambda function). Lastly, the executed function must have the requisite permission to write to the requested database table (in this case, the Project DynamoDB table).

To achieve this layered authorization, Lambda functions and application users use IAM roles, while the permissions required for API Gateway endpoints can be granted through the Lambda function's resource-based policy. This configuration illustrates the granular control required in the AWS environment, showcasing the necessity of precision in designing and implementing access controls.

Firstly, we introduce the entities related to permission that are required for case study representation.

```
context context05 extends context04                Context 05
sets
STATEMENT
constants
Sta1 Sta2 Sta3 OtherStatements
axioms
@axm4_1 partition(STATEMENT,{Sta1},{Sta2},
                            {Sta3},OtherStatements)
end
```

FIGURE 6.12: Permission Entities for Scenario 1

As detailed in Section 5.1, permissions are structured through the use of statements, each comprising mainly three attributes: resource, action, and effect. These statements provide an explicit specification of which actions are allowed or denied on particular resources.

As shown in Figure 6.12, STATEMENT is defined in context as carrier set. In the associated axiom, statements that is essential for the fulfilment of the case study functionality are introduced (*axm4_1*).

```
event INITIALISATION extends INITIALISATION
then
@act2_1 requester ≔ ∅
@act2_2 statement ≔ {Sta1,Sta2,Sta3}
@act2_3 ext_statement ≔ {Sta2}
@act2_4 sta_res ≔ {Sta1↦EPUpdProSt,Sta2↦FunUpdProData,Sta3↦Project}
@act2_5 sta_act ≔ {Sta1↦InvokeApi,Sta2↦ExecFun,Sta3↦UpdItemDB}
@act2_6 sta_pr ≔ {Sta2↦EPUpdProSt}
@act2_7 sta_ef ≔ {Sta1↦Allow,Sta2↦Allow,Sta3↦Allow}
@act2_8 actor ≔ {User1,FunUpdProData}
@act2_9 access_managable_resource ≔ {FunUpdProData}
@act2_10 actor_permission ≔ {User1↦Sta1,FunUpdProData↦Sta3}
@act2_11 resource_permission ≔ {FunUpdProData↦Sta2}
end
```

FIGURE 6.13: Ref 1: Initialisation Event for Scenario 1

The *Initialisation* event in Figure 6.13 illustrates necessary setup configurations for the fulfilment of the case study functionality. Actions from *act2_2* to *act2_7* show the configuration of permission statements, whereas *act2_10* and *act1_11* make associations between permission statements and their corresponding entities.

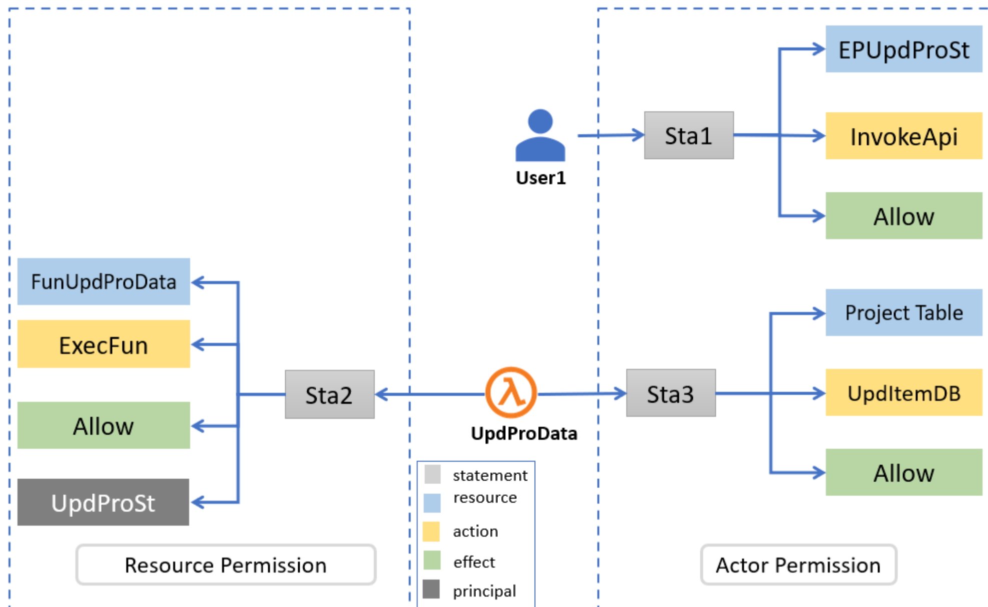


FIGURE 6.14: Ref 1: Visual of the Required Permissions for Fulfilment of "updating project status" functionality

Figure 6.14 visualises configurations that are done in the *INITIALISATION* event, as detailed in Figure 6.13. Therefore, the *sta1* statement grants permission to *User1* to execute the *EPUpdProSt* API endpoint, while *sta3* gives permission to the *FunUpdDataPro* function to update the *Project* database table. Additionally, the *sta2* statement allows the *EPUpdProSt* endpoint to execute the *FunUpdDataPro* function.

```
event init_req_ue extends init_req_ue
  any rq
  when
    @grd2_1 rq ∈ (Endpoint ∪ Function ∪ CogUser)
  then
    @act2_1 requester(new_req) := rq
  end
```

In this refinement level, requester is defined so each request is made by a requester that could an app user, a function or an endpoint. (requester $\in$ request $\rightarrow$ (Endpoint $\bigcup$ Function $\bigcup$ CogUser)). Moreover, as shown in *init_req_ue* Event-B event above, when a new request is initiated, a relation between the new request and the requester entity is created (*act2_1*).

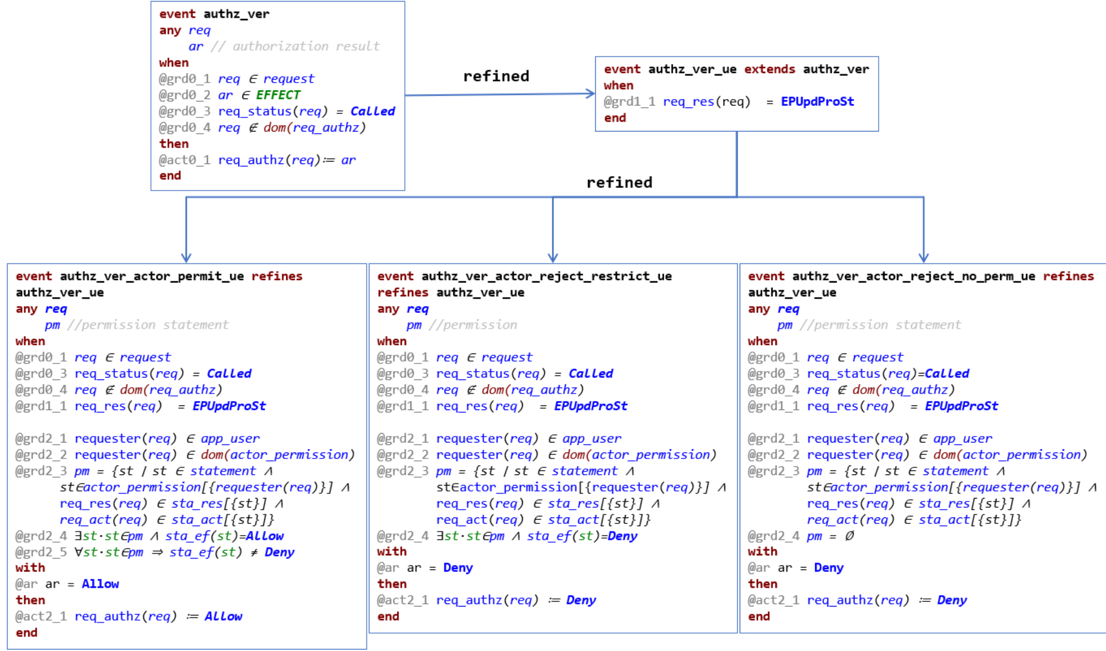


FIGURE 6.15: Refining Non-Deterministic Authorisation into Deterministic for the *EPUpdProSt* endpoint

In Figure 6.15, the refinement process of the authorization mechanism for requests directed at the *EPUpdProSt* endpoint is depicted. We follow the methodology that we propose in Section 5.1.3 to refine the non-deterministic authorization mechanism into a deterministic authorization mechanism. Whenever an app user sends a request to an API Gateway endpoint, the requester app user must have proper permission.

Therefore, we consider just the cases pertaining to actor permissions (as explored in Section 5.1.3). The abstract authorizer event (*authz_ver_ue*) is refined into three distinct events, capturing three different cases of authorization: the case that the requester has proper permission (*authz_ver_actor_permit_ue*), the case that the requester has a reject about the requested action (*authz_ver_actor_reject_restrict_ue*), and the case that the requester does not have any associated permission (*authz_ver_actor_reject_no_perm_ue*). Moreover, the *grd1_1* and *grd2_1* guards are pivotal in defining the request, defining that the target resource must be the EPUpdProSt endpoint (*grd1_1*) and the requester should be an app user (*grd2_1*).

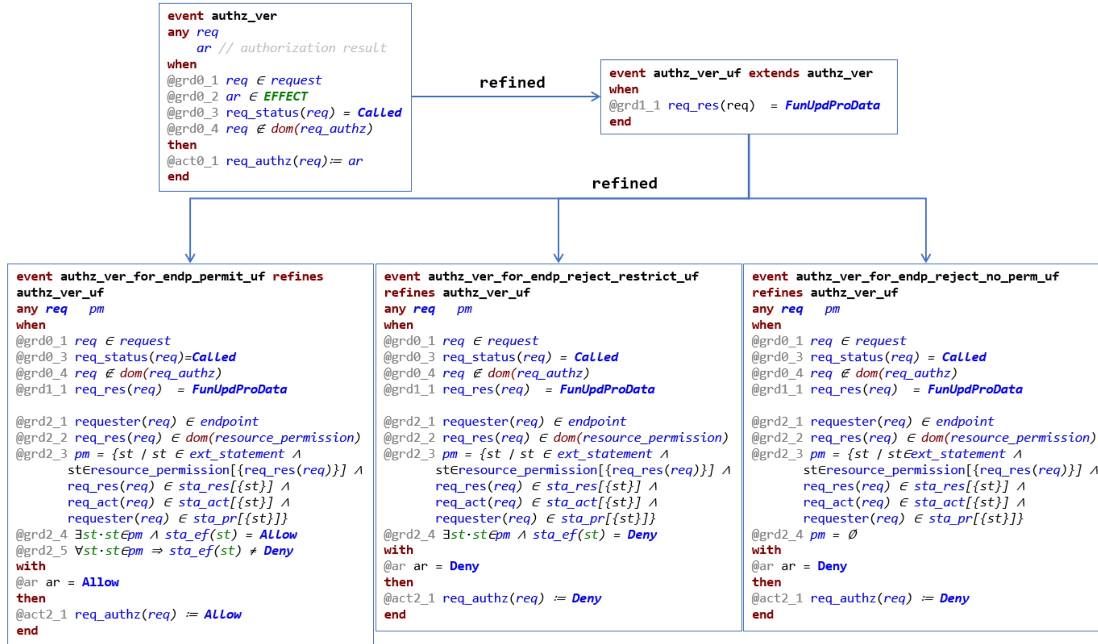


FIGURE 6.16: Refining Non-Deterministic Authorisation into Deterministic for the *FunUpdPro-Data Function*

Moreover, Figure 6.16 illustrates the refinement of the authorization mechanism concerning requests to the *FunUpdDataPro* function. Given that this function is triggered by an API Gateway endpoint, and given the inherent limitations of the API Gateway endpoint in the AWS environment, which lacks the ability to act as a permissions-bearing entity, resource permissions linked to the *FunUpdDataPro* function must be explicitly granted the permission of function execution to the inquiring endpoint. Therefore, as shown in Figure 6.16, three of the cases detailed in Section 5.1.1 could be possible for refinement of the *FunUpdDataPro* function authorization mechanism. These cases are: the case where the resource permission of the function allows the requester endpoint (*authz_ver_for_endp_permit_uf*), the case where the resource permission of the function bans the requester endpoint (*authz_ver_for_endp_reject_restrict_uf*), and the case where there is no statement in the resource permission of the function for the requester endpoint (*authz_ver_for_endp_reject_no_perm_uf*).

The refinement of the authorization mechanism pertaining to requests targeted at the *project* database table closely mirrors the structure seen in Figure 6.15, given the *FunUpdDataPro* function's reliance on an IAM role for directing requests to the *Project* database table.

However, it is vital to highlight that in the AWS context, permissions are granted to specific identities, like an application user or function, through IAM roles and IAM policies. In the further refinement stage, as detailed in the refinement strategy proposed in Chapter 5, these AWS specific nuances will be introduced in the model.

Introducing AWS-based Authorization mechanism specific features in Scenario 1:

As mentioned in Section 5.1.4, in this refinement step, the middle-ware abstractions *actor_permission* and *resource_permission* are replaced with AWS specific authorization entities, like IAM roles and IAM policies.

```
context context06 extends context05                                  Context 06
constants
IamUser IamRole IamPolicy

RoleAppProjMan RoleFunUpdProData OtherRoles

Pol1 Pol2 Pol3 OtherPols
axioms
@axm5_1 partition(IAM, IamUser, IamRole, IamPolicy)
@axm5_2 partition(IamRole, {RoleAppProjMan}, {RoleFunUpdProData}, OtherRoles)
@axm5_3 partition(IamPolicy, {Pol1}, {Pol2}, {Pol3}, OtherPols)
end
```

FIGURE 6.17: Application specific features: Roles, policies

As depicted in Figure 6.17, application specific features like roles and policies that are required for the execution of the case study scenario are created in the extended context. *RoleAppProjMan* defines the project manager application user role, while *RoleFunUpdProData* shows an IAM execution role for the *FunUpdDataPro* lambda function. Together, these roles, accompanied by their corresponding policies, are required to model the authorization mechanism for the fulfilment of "updating project status" functionality.

```
invariants
@inv3_1 iamrole ⊆ IamRole
@inv3_2 policy ⊆ IamPolicy
@inv3_3 partition(policy, id_policy, res_policy)
@inv3_4 actor_role ∈ actor → iamrole
@inv3_5 role_policy ∈ iamrole ↔ id_policy
@inv3_6 has_access_policy ∈ access_managable_resource ⇸ res_policy
@inv3_7 has_access_policy∼ ∈ res_policy → access_managable_resource
@inv3_8 id_policy_statement ∈ id_policy ↔ (statement\ext_statement)
@inv3_9 id_policy_statement∼ ∈ (statement\ext_statement) → id_policy
@inv3_10 res_policy_statement ∈ res_policy ↔ ext_statement
@inv3_11 res_policy_statement∼ ∈ ext_statement → res_policy

@inv3_comp1a actor_permission = (actor_role;role_policy;id_policy_statement)
@inv3_comp2a resource_permission = (has_access_policy;res_policy_statement)
```

FIGURE 6.18: Invariants Introducing Roles, policies

Invariants in Figure 6.18 represent role and policy features in the refined machine.

Moreover, the *inv3_comp1a* and *inv3_comp2a* invariants in Figure 6.18 are gluing invariants to illustrate the relationship between the abstract permission-sharing invariants and their refined counterparts. Therefore, *actor_permission*, which says that each actor has a set of permissions, embodies in a composition of states in the refined machine, illustrating that every

actor has a role linked to a set of policies, each encompassing a set of statements. Similarly, *resource_permission*, which says that a resource may have a set of permissions, is replaced with a composition of states stating that a resource may have a policy containing a set of statements.

```
event INITIALISATION
then
...
@act2_1 requester ≔ ∅
@act2_2 statement ≔ {Sta1,Sta2,Sta3}
@act2_3 ext_statement ≔ {Sta2}
@act2_4 sta_res ≔ {Sta1↦EPUpdProSt,Sta2↦FunUpdProData,Sta3↦Project}
@act2_5 sta_act ≔ {Sta1↦InvokeApi,Sta2↦ExecFun,Sta3↦UpdItemDB}
@act2_6 sta_ef ≔ {Sta1↦Allow,Sta2↦Allow,Sta3↦Allow}
@act2_7 actor ≔ {User1,FunUpdProData}
@act2_8 access_managable_resource ≔ {FunUpdProData}
@act2_11 sta_pr ≔ {Sta2↦EPUpdProSt}

@act3_1 iamrole ≔ {RoleAppProjMan,RoleFunUpdProData}
@act3_2 policy ≔ {Pol1,Pol2,Pol3}
@act3_3 id_policy ≔ {Pol1,Pol3}
@act3_4 res_policy ≔ {Pol2}
@act3_5 actor_role ≔ {User1↦RoleAppProjMan,FunUpdProData↦RoleFunUpdProData}
@act3_6 role_policy ≔ {RoleAppProjMan↦Pol1,RoleFunUpdProData↦Pol3}
@act3_7 has_access_policy ≔ {FunUpdProData↦Pol2}
@act3_8 id_policy_statement ≔ {Pol1↦Sta1,Pol3↦Sta3}
@act3_9 res_policy_statement ≔ {Pol2↦Sta2}
end
```

FIGURE 6.19: Ref 2: Initialisation Event for Case Study Scenario 1

Figure 6.19 shows the setup features tied to roles and policies. As the abstract states *actor_permission* and *resource_permission* are replaced with the composition of states, those abstract states are replaced with the refined details in the *INITIALISATION* event and in the rest of the model.

As illustrated in Figure 6.19, *act2_1* sets the registered roles that are required for the execution of the case study scenario. The actions *act2_2*, *act2_3*, and *act2_4* define the policies and their types, while the actions *act2_8* and *act2_9* define the permission statements that those policies have. *act2_5* makes relationships between roles and their corresponding actors, which are application users or functions, whereas the *act2_6* and *act2_7* actions create associations between policies and their related to roles or resources.

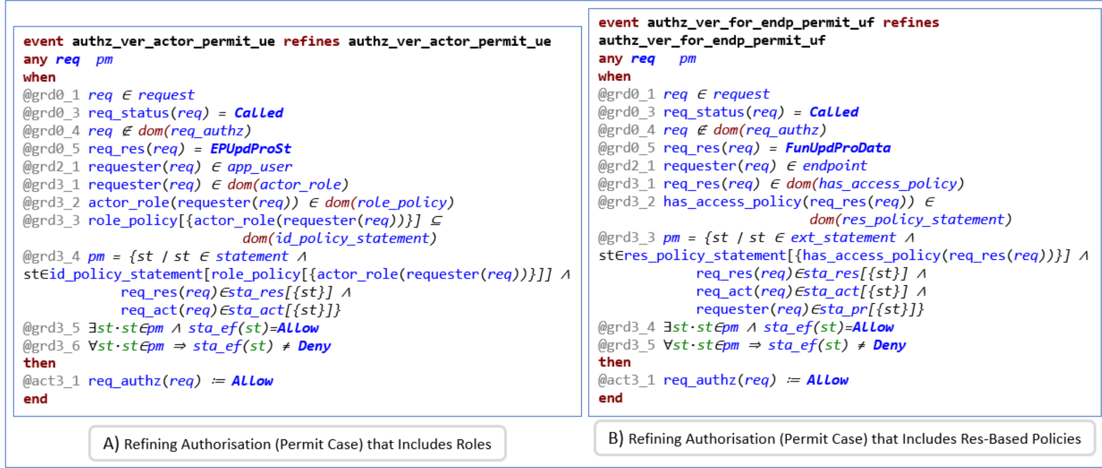


FIGURE 6.20: Ref 2: Authorisation Verification events Permit Cases

Introducing roles and policies and the stepwise evolution of abstract states like *actor_permission* and *resource_permission* affects authorisation verification events. Due to space limitations, Figure 6.20 selectively shows the effects on refinement of *authz_ver_actor_permit_ue* and *authz_ver_for_endp_permit_uf*. The *authz_ver_actor_permit_ue* event (shown in Figure 6.20A) illustrates the permit case of authorisation mechanism for requests targeting the *EPUpdProSt* endpoint. The guard *grd3_4* states that *pm* includes all statements that belong to the requester role's policies and that are related to the requested resource. Moreover, the guard *grd3_5* ensures that there is at least one statement that allows the requested action in the statements in *pm*, while *grd3_6* ensures that there is no statement that restricts the requested action in the statements in *pm*. Furthermore, *authz_ver_for_endp_permit_uf* (elaborated in Figure 6.20B) represents the permit case of authorization mechanism for requests targeting the *FunUpdProData* function. In this case, the resource-based policy of the *FunUpdProData* function must include a statement to grant permission to the inquiring endpoint to execute the *FunUpdProData* function (*grd3_3*, *grd3_4*, *grd3_5* in Figure 6.20B).

6.1.2 Scenario 2: "Promoting a User as Project Manager"

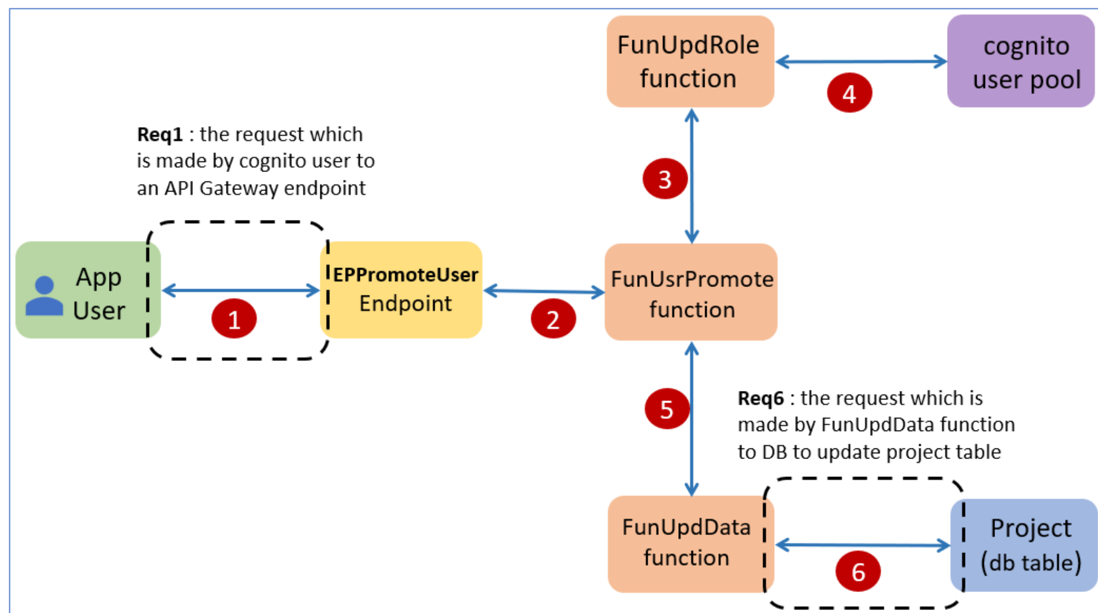


FIGURE 6.21: The Structure of "Promoting a User as Project Manager" Functionality

The scenario: User1, with a Department Manager role, wants to allocate User2 to Project1 as project manager. Figure 6.21 illustrates the structure of the "Promoting a User as Project Manager" functionality. In system design, we assume that the developer of the case study system follows the single responsibility principle, resulting in each function having a single task to do. For instance, the only task of the *UpdData* function is to update the DynamoDB table. To fulfil the "Promoting a User as Project Manager" functionality, the following steps must be satisfied:

1. User1 initiates a request (req1) to promote User2 as project manager of Project1.
2. The client app sends User1's request (req1) to the API Gateway service.
3. The API Gateway receives the request.
4. The authorizer of the *EPPromoteUser* endpoint determines whether User1 has permission to execute the *EPPromoteUser* endpoint, the endpoint associated with ”Promoting a User to Project Manager” functionality.
5. If User1 has the proper permission, the *EPPromoteUser* endpoint is executed.
6. When the requested endpoint is executed successfully, the endpoint calls the *FunUsrPromote* lambda function (req2).
7. The authorizer of the *FunUsrPromote* function determines whether the *EPPromoteUser* endpoint is allowed to execute the function in the function's resource-based policy.

8. The *FunUsrPromote* function is executed if the *EPPromoteUser* endpoint is permitted.

9. When the *FunUsrPromote* function is executed, it first makes a request to the *FunUpdRole* function (req3), which is responsible for updating the role of an application user, to execute.

10. The authorizer of the *FunUpdRole* function determines whether the *FunUsrPromote* function has proper permission to execute the function.

11. The *FunUpdRole* function is executed if the *FunUsrPromote* function is allowed.

12. The *FunUpdRole* function sends a request to the cognito user pool (req4) to update the role of User2.

13. The authorizer in the Cognito checks whether the *FunUpdRole* function has permission to update the role of User2.

14. The user role of User2 is updated as "Project Manager" after the *FunUpdRole* function request (req4) is authorised.

15. The *FunUsrPromote* function initiates a new request (req5) to the *FunUpdData* function, which is responsible for updating the database table (project table).

16. The *FunUsrPromote* function sends the request (req5) to the *FunUpdData* function after both the request to the *FunUpdRole* function (req3) and the request to the cognito user pool (req4) are successfully executed.

17. The *FunUpdData* function is executed if the *FunUsrPromote* function has proper permission.

18. The *FunUpdData* function sends a request to the *project* DynamoDB database (req6) to allocate User2 to Project1 as project manager.

19. User2 allocates to Project1 as project manager if the *FunUpdData* function has permission to do so.

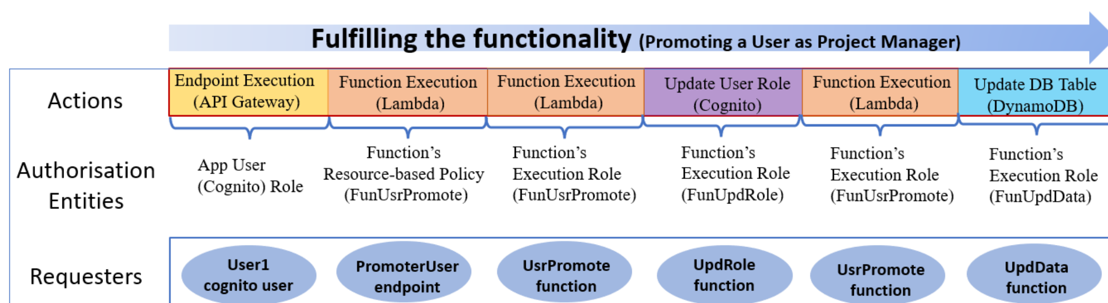


FIGURE 6.22: The Authorisation entities in "Promoting a User as Project Manager" Functionality

Additionally, Figure 6.22 provides a comprehensive illustration of authorization elements, including IAM roles and IAM policies, and their corresponding responsibilities during the fulfilment of the "updating project status" functionality. For instance, for fulfilment of the functionality, the user role of the requester must have permission to execute the requesting API Gateway endpoint, while the resource-based policy of the *FunUsrPromote* function must allow the requester endpoint to execute the *FunUsrPromote* function.

6.1.2.1 Model "Promoting a User as Project Manager" Functionality

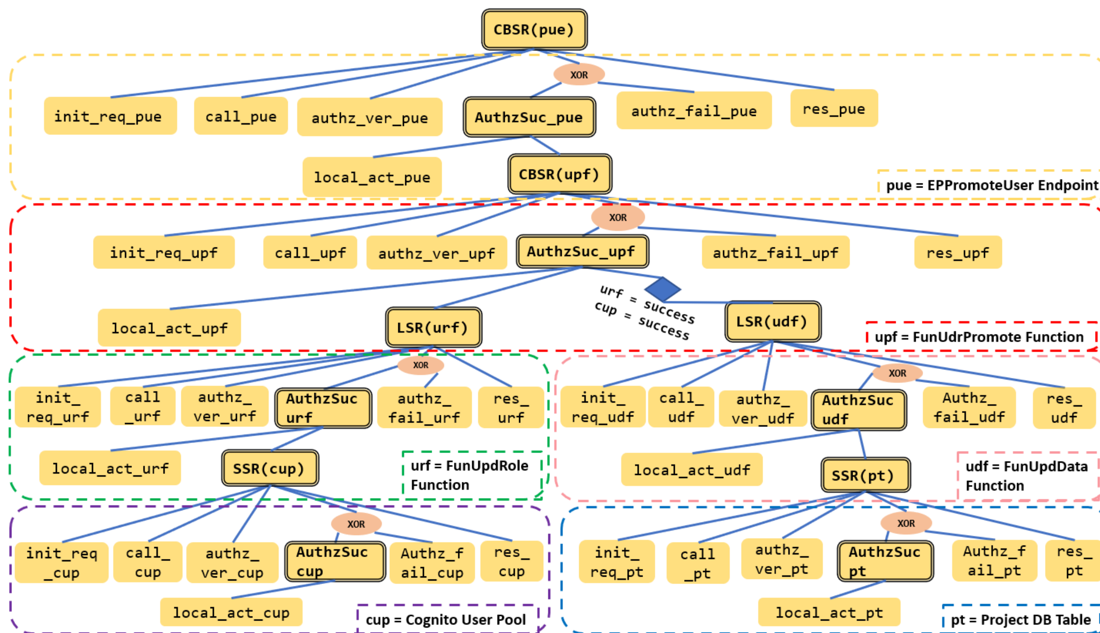


FIGURE 6.23: The tree-like representation of ”Promoting a User as Project Manager” functionality

In Figure 6.21, the architectural design of the "Promoting a User as Project Manager" functionality built in the AWS environment is shown. Using our CBSR pattern, as elaborated upon in Section 4.2.4, could offer an effective approach to modelling the functionality in Event-B.

To fulfil the "Promoting a User as Project Manager" functionality, six distinct requests, as detailed in Figure 6.21, must be successfully executed. The tree-like diagram in Figure 6.23 depicts requests, their orders, and relations between them during the process of the functionality of ”Promoting a User as Project Manager”. Each dashed and coloured square represents a request handling pattern. Moreover, the blue diamond represents a set of conditions and the branch under the diamond can be enabled only if the conditions of the diamond are satisfied. The conditions represented by diamond are the successful execution of all requests at the same level and their subsequent requests. For example, as depicted in Figure 6.23, to make a request *FunUpdData* function to execute (*LSR(udf)*), the request targeting *FunUpdRole* function (*LSR(urf)*), and the request targeting cognito user pool (*LSR(cup)*) must be successfully executed.

Textual Representation 6.2 illustrates the implementation of CBSR pattern to model the "Promoting a User as Project Manager" scenario that is detailed in Figure 6.23.

CBSR(pe) =
RHP(ir_pe, cs_pe, av_pe, la_pe, **BSR(upf)**, af_pe, r_pe)

CBSR(upf) = **LSR(urf)** $<>$ **LSR(udf)**

LSR(urf) =
RHP(ir_urf, cs_urf, av_urf, la_urf, **SSR(cup)**, af_urf, r_urf)
SSR(cup) =
RHP(ir_cup, cs_cup, av_cup, la_cup, SKIP, af_cup, r_cup)

LSR(udf) =
RHP(ir_udf, cs_udf, av_udf, la_udf, **SSR(pt)**, af_udf, r_udf)
SSR(pt) =
RHP(ir_pt, cs_pt, av_pt, la_pt, SKIP, af_pt, r_pt)

(6.2)

The abbreviation in the Textual Representation 6.2 means the following :

_pe = EPPromoteUser Endpoint
_upf = FunUsrPromote Function
_urf = FunUpdRole Function
_udf = FunUpdData Function
_cup = Cognito User Pool
_pt = Project DB Table

The requests that are required for the fulfilment of the functionality map to the entities in Textual Representation 6.2. Table 6.7 provides a clear representation of how these requests correspond to entities in both the pattern and the system:

In Pattern	In The System	Request
BSR(pe)	The request that is sent EPPromoteUser Endpoint to execute	req1
BSR(upf)	The request that is sent FunUsrPromote Function to execute	req2
LSR(urf)	The request that is sent FunUpdRole Function to execute	req3
SSR(cup)	The request that is sent Cognito User Pool to update a app user role	req4
LSR(udf)	The request that is sent FunUpdData Function to execute	req5
SSR(pt)	The request that is sent DynamoDB to update project table	req6

TABLE 6.7: Requests During The Process of Scenario 2

6.1.2.2 Event-B model of Scenario 2

Likewise, the "updating project status" functionality scenario, features specific to the AWS environment, features related to the case study system, and features that are required for fulfilment of the "Promoting a User as Project Manager" functionality are required in contexts and machines' initialisation events in the model. The setup configuration related AWS environment and case study system are the same as the model of scenario in Section 6.1.1.1.

```
Context 04
constants
User1 // a cognito user (app user)
User2 // a cognito user (app user)
OtherAppUsers
CupEmInPro // a cognito user pool that represent whole app user in the case study system
EPPromoteUser // endpoint that maps to "Promote a User" functionality
FunUsrPromote //lambda function that satisfies "update Project Status" functionality
FunUpdRole //function that update a cognito user's role
FunUpdData //function that update an DB table
OtherFunctions

Project // Project object in the system
Department //Department object in the system
Project1 //project 1
Department1 //department 1
axioms
@axm3_1 CupEmInPro ∈ CogUserPool
@axm3_2 partition(CogUser, {User1},{User2},OtherAppUsers)
@axm3_3 UsrInPool(User1) = CupEmInPro
@axm3_4 UsrInPool(User2) = CupEmInPro
@axm3_5 EPPromoteUser ∈ Endpoint
@axm3_6 partition(Function, {FunUsrPromote},{FunUpdRole},{FunUpdData},OtherFunctions)
@axm3_7 Project ∈ DBTable
@axm3_8 Department ∈ DBTable
@axm3_9 Project ≠ Department
@axm3_10 Project1 ∈ DataItem
@axm3_11 Department1 ∈ DataItem
@axm3_12 Project1 ≠ Department1
@axm3_13 ItemInTable(Project1) = Project
@axm3_14 ItemInTable(Department1) = Department
```

FIGURE 6.24: The Features Related to Scenario 2 Configuration

Figure 6.24 illustrates some features and configurations that are required for scenario 2. *User1* and *User2* are Cognito users (*axm3_2*), which are app users in application domain. Moreover, *axm3_3*, and *axm3_4* ensures those users belong to the same cognito User Pool, which is *CupEmInPro*. We assume that *CupEmInPro* Cognito user pool defines the cognito user pool that connects with the case study application.

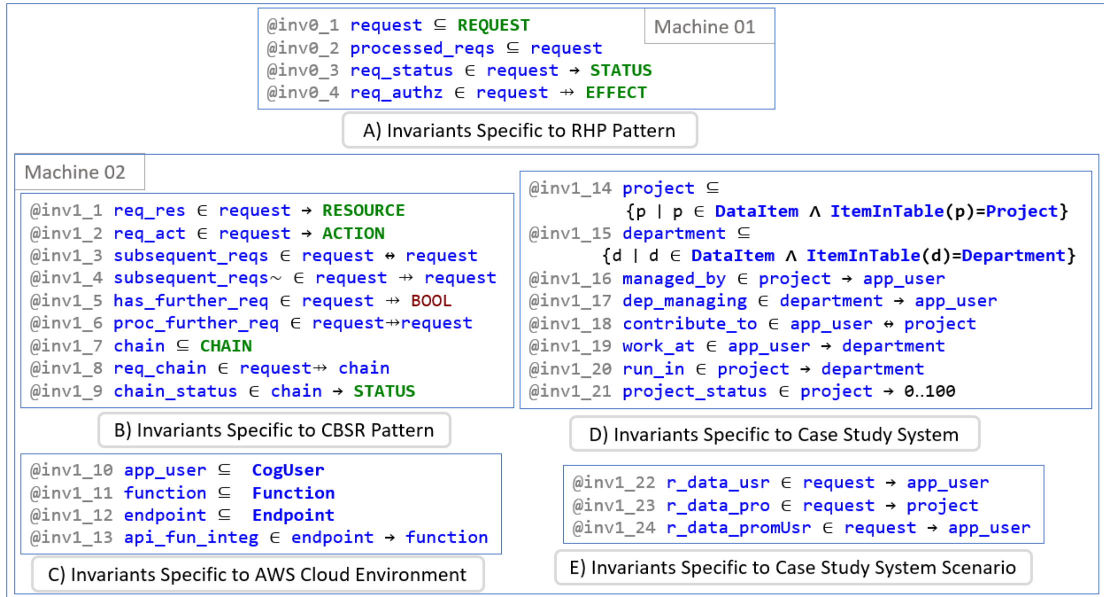


FIGURE 6.25: Abstract Invariants for Scenario 2

Figure 6.25 shows the invariants of the abstract of the model of scenario 2. First of all, Figure 6.25A shows invariants about to the RHP pattern. And then, this pattern is refined into a CBSR pattern to model the case study system scenario. Moreover, the invariants in Figure 6.25B are invariants to introduce features of the CBSR pattern, whereas the ones in Figure 6.25C are invariants specific to the AWS cloud environment. Furthermore, Figure 6.25D defines the features specific to the case study system, while invariants in Figure 6.25C introduce the structure of request payload pertaining to the scenario. Therefore, in the request's payload, information about the user who attempts to update a user role (*r_data_usr*), the user whose role is updated (*r_data_promUsr*), and the project to which a user is allocated as project manager must be kept (*r_data_pro*).

```
event INITIALISATION extends INITIALISATION
then
...
@act1_9 app_user ≔ {User1, User2}
@act1_10 endpoint ≔ {EPPromoteUser}
@act1_11 function ≔ {FunUsrPromote,FunUpdRole,FunUpdData}
@act1_12 api_fun_integ ≔ {EPPromoteUser↦FunUsrPromote}

@act1_13 project ≔ {Project1}
@act1_14 department ≔ {Department1}
@act1_15 managed_by ≔ {Project1↦User1}
@act1_16 dep_managing ≔ {Department1↦User1}
@act1_17 contribute_to ≔ {User1↦Project1}
@act1_18 work_at ≔ {User1↦Department1, User2↦Department1}
@act1_19 run_in ≔ {Project1↦Department1}
@act1_20 project_status ≔ {Project1↦0}

@act1_21 r_data_usr ≔ ∅
@act1_22 r_data_pro ≔ ∅
@act1_23 r_data_promUsr ≔ ∅
end
```

FIGURE 6.26: Initialisation for Scenario 2 Abstraction

As shown in Figure 6.26, the configurations for scenario 2, "promoting a user as project manager", have many similarities with scenario 1 (Figure 6.7). However there are some differences. Firstly, we use CBSR pattern in Scenario 2 so the order of the requests are modelled based on the approach detailed in Section 4.2.4. The structure of request body also is different from the request body of the scenario 1 because different information is required to fulfill the functionality in the scenario 2. In Scenario 2, data of two distinct app users and one project are required : the user who attempts to promote role of a user (*r_data_usr*), the user whose role is prompted (*r_data_promUsr*), and the project which a project manager is allocated to (*r_data_pro*).

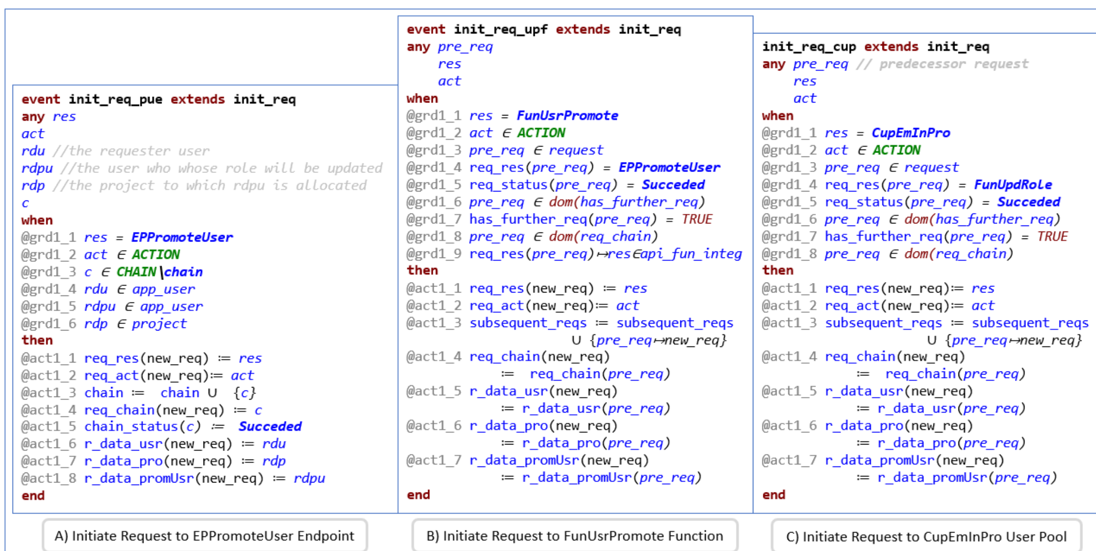


FIGURE 6.27: Request Initiation Events for Scenario 2 Abstraction

Figure 6.27 depicts initiating some requests during the fulfilment of scenario 2. The *init_req_pue* event (Figure 6.27A) introduces the initiation of a request targeted to the "EPPromoteUser" endpoint, whereas the *init_req_upf* represents initiating a request sent to the *FunUsrPromote* function (Figure 6.27B). Then, *init_req_cup* defines initiating a request sent to *CupEmInPro* cognito user pool (Figure 6.27C). The value of the *res* parameter specifies the target resource of the request (*grd1_1* in all events in the figures.

It's worth noting that, as explained in the model of Scenario 1 in Section 6.1.1.1 as well, the information in a request must be transferred to a further request during the process of functionality fulfilment. In the *init_req_pue* event (illustrated in Figure 6.27A), the parameters - *rdu*, *rdp*, and *rdpu* - show the requester user, the project that a new project manager is allocated to, and the user that will be promoted as project manager, respectively. The corresponding actions, such as *act1_6*, *act1_7*, and *act1_8* in Figure 6.27A, represent the association between these informational elements and the new request, respectively. During the process of the functionality, when a new request is initiated, this information is transferred to it. For instance, when the *EPPromoteUser* endpoint initiates a request targeting the *FunUsrPromote* function, as seen in *init_req_upf* (Figure 6.27), it transfers the information received from the app user in its request. (*act1_5*, *act1_6*, and *act1_7* in *init_req_upf*).

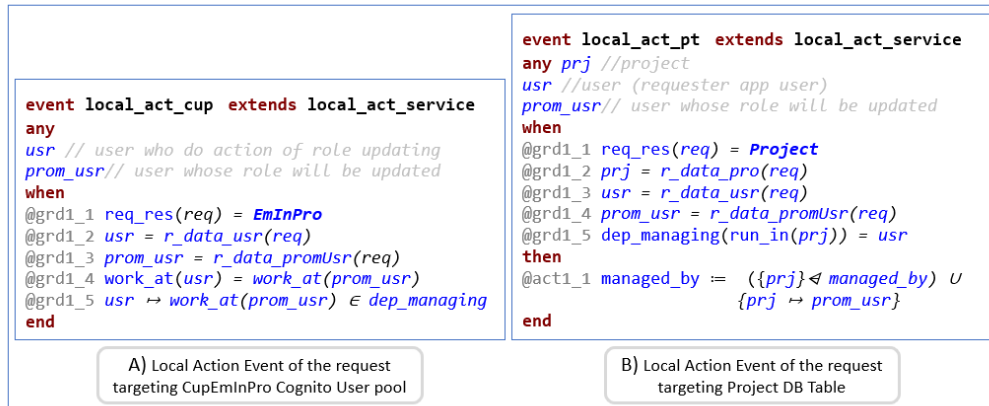


FIGURE 6.28: Local Action Events for Scenario 2 Abstraction

After req1, req2, req3, and req4 requests in Table 6.7 successfully executed, the *local_act_cup* in Figure 6.28A that represents updating the role of an app user (cognito user) will be enabled to execute. Moreover, the successful executions of req1, req2, req3, req4, req5, and req6 requests in Table 6.7 enable the *local_act_pt* in Figure 6.28B that defines the update on the project DB table. As app user roles are not introduced in this abstract level of the model, the action related to app users' role will be introduced in the refined event of *local_act_cup*.

Implementing And Refining Authorisation Mechanism in Scenario 2 :

To introduce the authorization mechanism in the model of Scenario 2, we follow the approach explained in Chapter 5. Therefore, the implementation authorization mechanism in the model of Scenario 2 ("Promoting a User as Project Manager" functionality) is the same as introducing

and refining the authorization mechanism in the model of Scenario 1 ("Update Project Status" functionality) in Section 6.1.1.2.

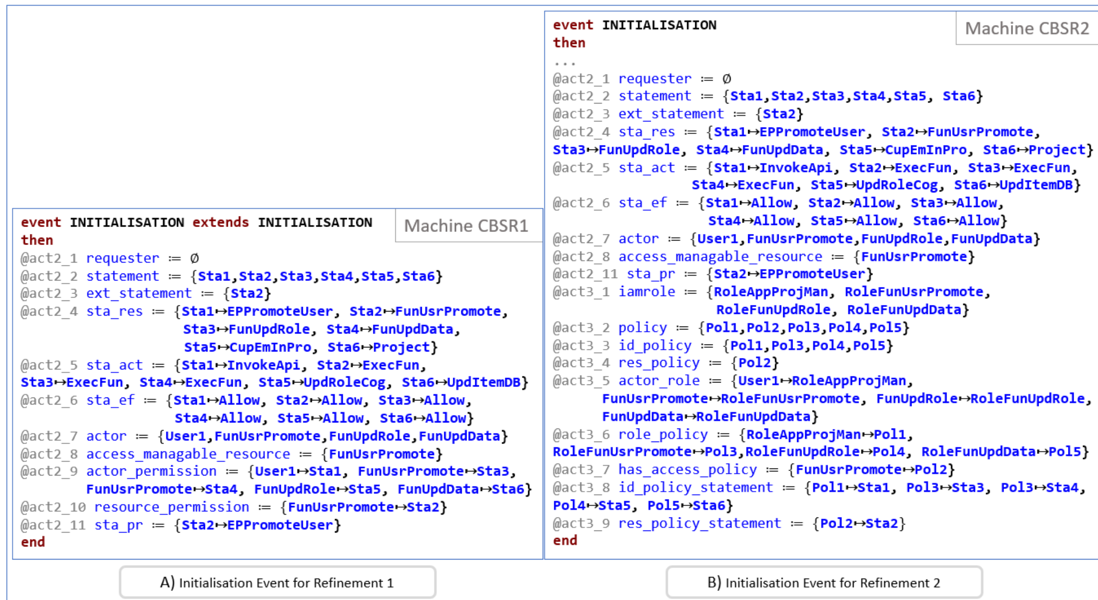


FIGURE 6.29: Initialisation Events for Scenario 2 in Refinement Steps

Furthermore, likewise, in the model of Scenario 1, the setup configuration of Scenario 2 is set in initialisation events. Initialisation events in Figure 6.29 are the required setup configurations for the fulfilment of Scenario 2 functionality. In refinement 1 step, the structure of permissions and their impacts on the authorization mechanism are introduced, while refinement 2 step introduces roles, policies (AWS-specific authorization entities), and their effects on the authorization mechanism. Therefore, Figure 6.29A shows setup configurations: the required users, permissions, and resources for fulfilment of Scenario 1, whereas Figure 6.29B adds the configurations regarding roles, policies, and corresponding relations. For instance, in the Initialisation event in Figure 6.29A, *Sta1* is a permission statement (*act2_2*) which allows (*act2_6*) to invoke (*act2_5*) EPPromoteUser endpoint (*act2_4*). The *Sta1* permission is granted to the *User1* app user (*act2_9*).

6.2 Case Study: Learning Management System

The learning management system serves as a comprehensive solution for educational institutions, facilitating various functions such as module management for lecturers and assignment submission for students, among other capabilities. The case study system, representing the learning management system, is characterised by several key entities, including user, personal information, role, school, module, content, and assignment entities. Users are categorised based on their roles in the system, with designations such as students, lecturers, heads of school, and admin, each granted distinct access and administrative privileges. For instance,

moderating a module requires a user to have a lecturer role. General requirements for the Learning Management System are shown in Table 6.8, whereas requirements for admins, heads of school, lecturers, and students are outlined in Table 6.9, Table 6.10, Table 6.11, and Table 6.12, respectively.

R1	Each module is moderated by a lecturer.
R2	Each module belongs to only one school.
R3	Each module has a specific content.
R4	A student can enrol in multiple modules.
R5	To pass the modules that are taken, a student should hand-in and pass their exams.
R6	Each module has one or more exam(s).
R7	Each user has a specific personal information to define him/her.
R8	Each user has a role that determines his/her access level in the system.
R9	Those roles are: Student, Lecturer, Head of School, and Admin.

Table 6.8: General Requirements for Learning Management System

R10	An admin can add/delete a student to/from the system.
R11	An admin can add/delete a lecturer to/from the system.
R12	An admin can view the personal information of any user.

Table 6.9: Requirements of Admins

R13	A head of school can create/delete/update a new module in his/her school.
R14	A head of school can assign a lecturer to a specific module in his/her school.
R15	A head of school can view the personal information of all lecturers/students in his/her school.
R16	A head of school can create/update the maximum limit of student enrollments for a module.

Table 6.10: Requirements of Head of School

R17	A lecturer can view the personal information of all students who have enrolled in his/her module(s).
R18	A lecturer can view/update their personal information.
R19	A lecturer can add/upload module materials.
R20	A lecturer can set an exam for their modules.
R21	A lecturer can manage/view grades of assignments that done by students (insert, update, calculate final grade).

Table 6.11: Requirements Lecturers

R22	A student can view/update his/her personal information.
R23	A student can enrol in courses that are available.
R24	A student can hand in assignments of modules that s/he takes.
R25	A student should be able to view only their own grade.

TABLE 6.12: Requirements of Students

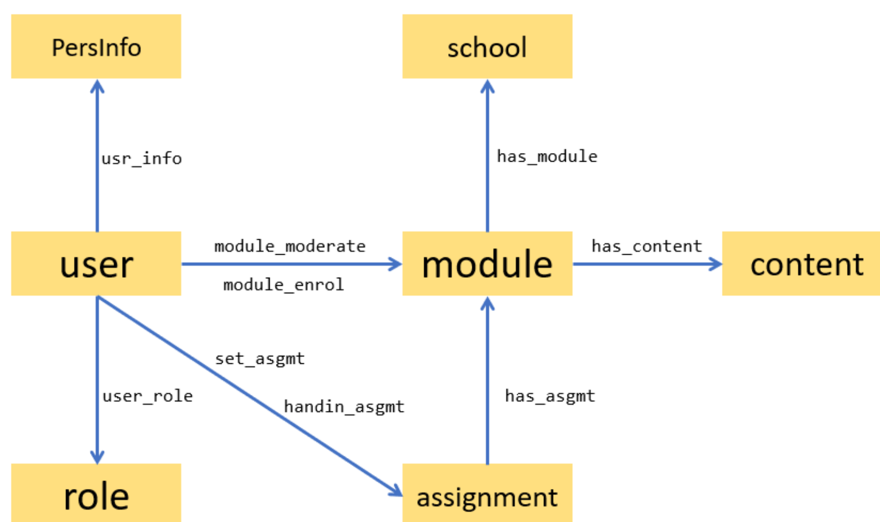


FIGURE 6.30: A Basic Structure of Learning Management System

The graph in Figure 6.30 visualises the relations between entities in the system detailed above requirements. When this system is built on the AWS cloud environment, for application users and their personal information, the Amazon Cognito service is used, whereas the DynamoDB database service is used to manage *module*, *school*, *content*, and *assignment* features. Moreover, to design different access level rights, like student and lecturer rights, the AWS IAM service is used. The structure of the *learning management system* that is built with an AWS-based serverless architecture is similar to the *project management system* (case study 1) in terms of the services used. Therefore, Figure 6.1 also represents the structure of the *learning management system* in the AWS cloud environment. As mentioned before, each system functionality maps to an API Gateway endpoint. Moreover, to satisfy a functionality, a user must be able to execute the corresponding endpoint.

6.2.1 Scenario 3: Updating Project Status

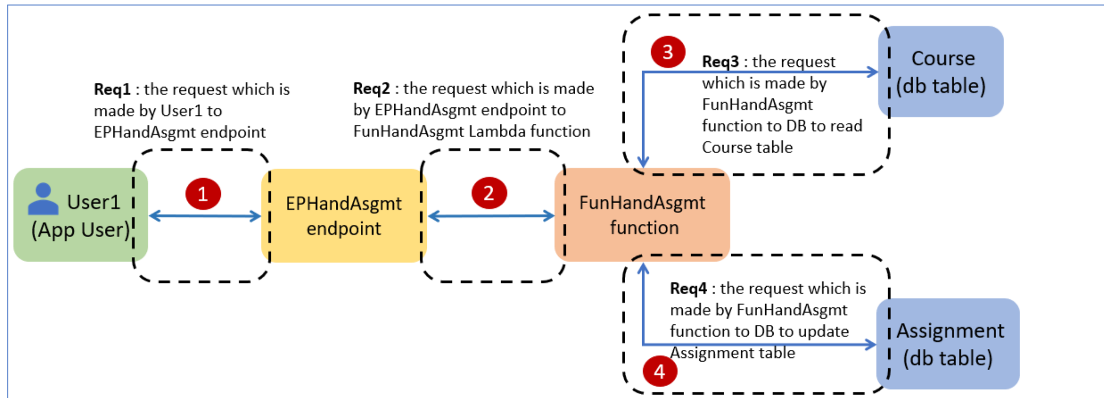


FIGURE 6.31: The Structure of "Hand in an Assignment" Functionality

The scenario: User1, with a Student role, wants to hand in Assignment1 of Module1, a module that s/he is enrolled in. The structure of the ”Hand in an Assignment” functionality in the AWS environment is illustrated in Figure 6.31. The following steps are executed to fulfil this functionality:

1. User1 initiates a request to hand in Assignment1 of Module1.
2. The client app sends User1’s request to the API Gateway service.
3. The API Gateway receives the request.
4. The authorizer of the EPHandAsgmt endpoint determines whether User1 has permission to execute the EPHandAsgmt endpoint, the endpoint associated with "Hand in an Assignment" functionality.
5. If User1 has the proper permission, the EPHandAsgmt endpoint is executed.
6. When the requested endpoint is executed successfully, the endpoint calls the FunHandAsgmt lambda function.
7. The authorizer of the FunHandAsgmt function determines whether the EPHandAsgmt endpoint is allowed to execute the function in the function’s resource-based policy.
8. The FunHandAsgmt function is executed if the EPHandAsgmt endpoint is permitted.
9. Then the executed FunHandAsgmt lambda function sends a request to DynamoDB to read the Module DB table to check whether User1 enrolled in Module1.
10. The authorizer of the Module table determines whether the FunHandAsgmt function has permission to read the requested table.

11. The FunHandAsgmt function makes a request to the Assignment DB table to put Assignment1 if User1 enrolled in Module1 that Assignment1 belongs to.

12. The authorizer of the Assignment table determines whether the FunHandAsgmt function has permission to update the requested table.

13. The Assignment1 is written if the FunHandAsgmt function is allowed.

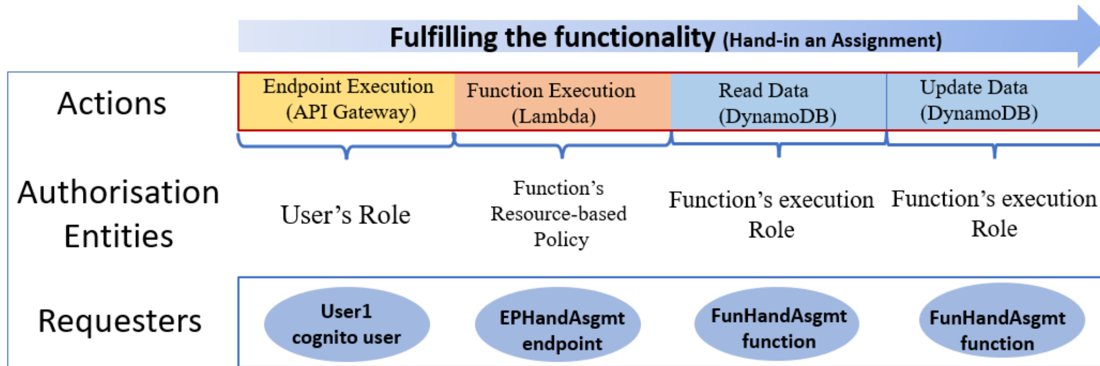


FIGURE 6.32: The Fulfilling of "Hand in an Assignment" Functionality

Moreover, Figure 6.32 illustrates the authorization entities involved, such as IAM roles or policies, and their respective responsibilities during the process of the fulfilment of the "Hand in an Assignment" functionality.

6.2.1.1 Model "Hand in an Assignment" Functionality

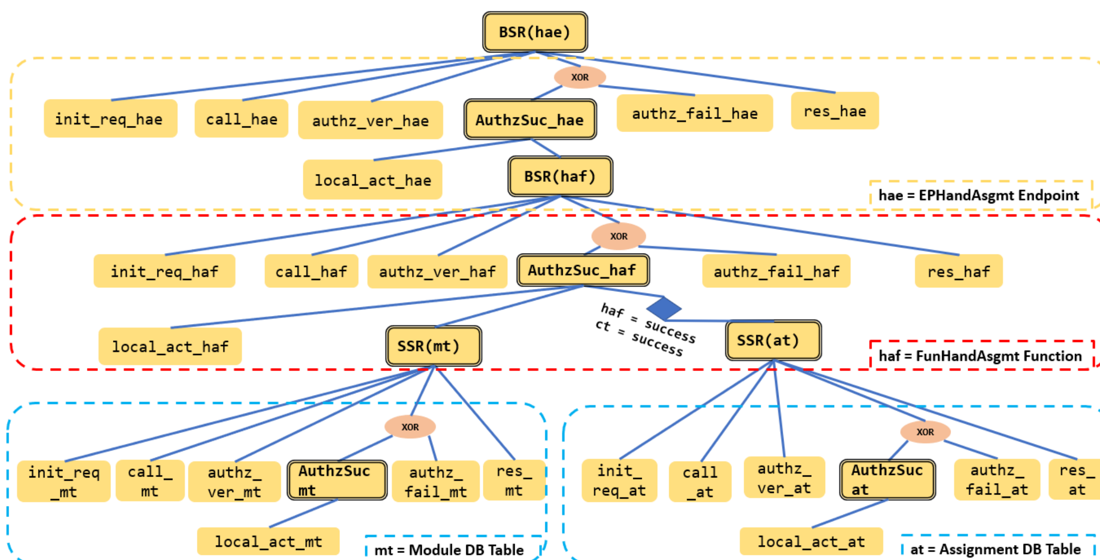


FIGURE 6.33: The tree-like representation of "Hand in an Assignment" functionality

As depicted in Figure 6.31, for the fulfilment of the "Hand in an Assignment" functionality, four distinct requests must be successfully executed. Moreover, there is a branching in the

sequential order of requests. Therefore, the BSR pattern, detailed in Section 4.2.3, could be helpful to model this functionality.

The tree-like diagram in Figure 6.33 visualises the use of BSR pattern to build a model for ”Hand in an Assignment” functionality. As illustrated in Figure 6.33, the requests required in the process of ”Hand in an Assignment” functionality fulfilment are represented as follows:.

In Pattern	In The System	Request
BSR(hae)	The request that is sent the EPHandAsgmt endpoint to execute	req1
BSR(haf)	The request that is sent the FunHandAsgmt function to execute	req2
SSR(mt)	The request that is sent the Module DB table to read	req3
SSR(at)	The request that is sent the Assignment DB table to update the requested the assignment value	req4

Table 6.13: Requests During The Process of ”Hand in an Assignment” Scenario

Therefore, as clearly detailed in Figure 6.33, a request to update the Assignment table (*SSR(at)*) can be executed after a request to execute the FunHandAsgmt lambda function (*BSR(haf)*) and then a request to read the Module table (*SSR(mt)*) are successfully executed consecutively. Moreover, a request to execute the FunHandAsgmt lambda function (*BSR(haf)*) can be executed if a request to invoke the EPHandAsgmt API Gateway endpoint (*BSR(hae)*) is successfully executed.

BSR(hae) =
RHP(ir_hae, cs_hae, av_hae, la_hae, **BSR(haf)**, af_hae, r_hae)

BSR(haf) = **SSR(mt)** $<>$ **SSR(at)**

SSR(mt) =
RHP(ir_mt, cs_mt, av_mt, la_mt, SKIP, af_mt, r_mt)

SSR(at) =
RHP(ir_at, cs_at, av_at, la_at, SKIP, af_at, r_at)

(6.3)

Furthermore, Textual Representation 6.3 shows the textual form of BSR pattern implementation to model the ”Hand in an Assignment” scenario.

6.2.1.2 Event-B Model The Scenario

In this section, we model the ”Hand in an Assignment” scenario in Event-B based on the BSR implementation in Section 6.2.1. Our Event-B model of the scenario consists of six contexts and three machines. In the first context, as shown in Figure 4.4, the features that are required to track events in the life-cycle of a request execution and flags related to authorization outcome are introduced. Then, resources, resource types, and actions are introduced in context 02, whereas in context 03, details about Data and Cognito resource types are defined. These two contexts are shown in Figure 6.5 in Section 6.1.1.2.

```
context context04 extends context03
constants
User1 // a cognito user (app user)
CupLrnMngSys // a cognito user pool for app users
EPHandAsgmt //endpoint for the case study scenario
FunHandAsgmt //function

// DB tables for case study system
Module  School  Content
Assignment  PersInfo  OtherDBTables

//entities for scenario
School1   Module1   Asgmt1   Content1  PersInfo1
OtherDataItems
axioms
  @axm3_1 User1 ∈ CogUser
  @axm3_2 CupCrsMngSys ∈ CogUserPool
  @axm3_3 UsrInPool(User1) = CupCrsMngSys
  @axm3_4 EPHandAsgmt ∈ Endpoint
  @axm3_5 FunHandAsgmt ∈ Function
  @axm3_6 partition(DBTable,{Module},{School},{Content},
                {Assignment},{PersInfo},OtherDBTables)
  @axm3_7 partition(DataItem,{Module1},{School1},{Content1},
                     {Asgmt1},{PersInfo1},OtherDataItems)
  @axm3_8  ItemInTable(Module1)= Module
  @axm3_9  ItemInTable(School1)= School
  @axm3_10 ItemInTable(Content1)= Content
  @axm3_11 ItemInTable(Asgmt1)= Assignment
  @axm3_12 ItemInTable(PersInfo1)= PersInfo
end
```

FIGURE 6.34: Features Specific to the Case Study System and the Scenario

Figure 6.34 depicts context 04, where entities specific to the case study system (Learning Management System) and the scenario ("Hand in An Assignment") are introduced. The entities of the case studies, including School, Module, Content, Assignment, Personal information, are defined as DBTable (database table) typed resources (*axm3_6*), while CupLrnMngSys represents a Cognito user pool for the case study system (Learning Management System) (*axm3_2*). User1, School1, Module1, Asgmt1, Content1, PersInfo1 constants represent the features required for the case study scenario (”Hand in an Assignment”).

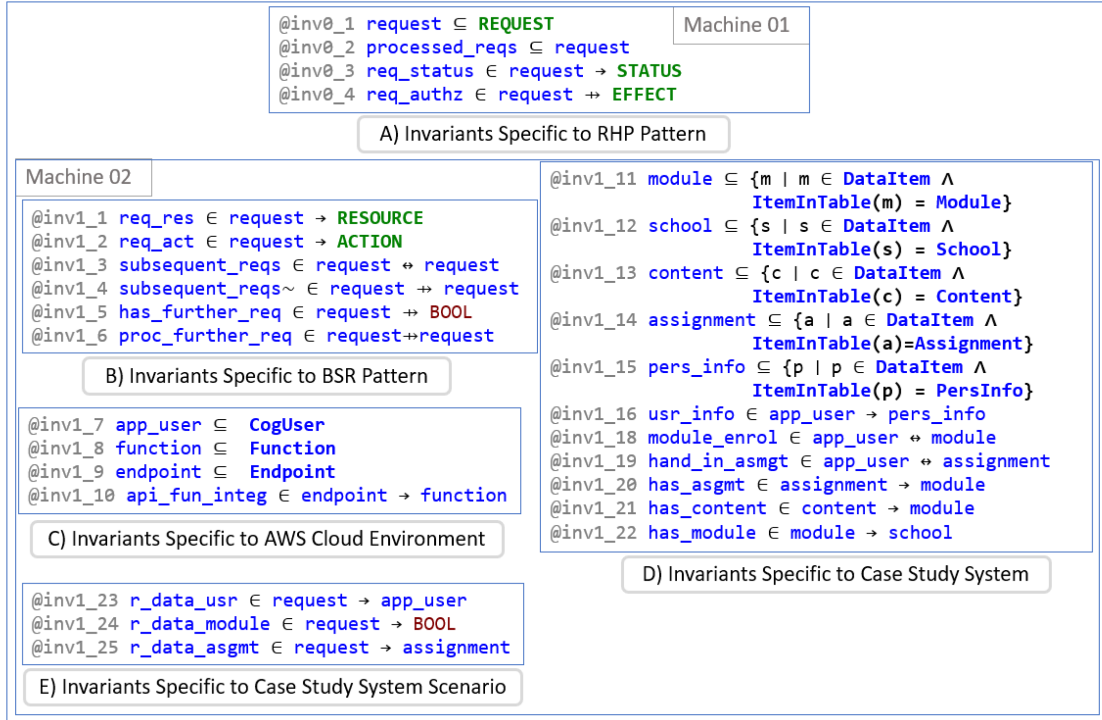


FIGURE 6.35: Invariants for "Hand in An Assignment" Scenario (Abstract Machine)

Figure 6.35 illustrates invariants in the abstract machine of the model of the case study scenario. To begin with, like previous case study scenarios implementation, Figure 6.35A shows the invariants about the RHP pattern features in the abstract machine. Then, invariants in Figure 6.35B introduce the BSR pattern features that we proposed in Section 4.2.3. Moreoever, Figure 6.35C depicts invariants specific to the AWS cloud environment features, while Figure 6.35D introduces the invariants specific to the case study system (*Learning Management System*). Invariants that are between *inv1_11* and *inv1_15* show the data entities of the case study system. For example, *inv1_11* represents the registered modules in the system. Moreover, each user has one personal information (*inv1_16*). A user can enrol in multiple modules (*inv1_18*). Types of users, such as students and lecturers, will be introduced in further refinement steps. Additionally, Each assignment and content belongs to one module (*inv1_20*, *inv1_21*, respectively), while each module belong to a school (*inv1_22*). Lastly, a user may hand in multiple assignments (*inv1_19*).

Furthermore, Figure 6.35E introduces information in the request's payload body that includes the required data to satisfy the scenario. Therefore, those invariants are specific to our case study scenario ("Hand in An Assignment"). The required data involves the requester app user (*inv1_23*), the assignment that is handed in (*inv1_25*), and information about whether the requester app user is enrolled in the module of the assignment (*inv1_24*).

```
event INITIALISATION
then
 ...
@act1_6 app_user ≔ {User1}
@act1_7 endpoint ≔ {EPHandAsgmt}
@act1_8 function ≔  {FunHandAsgmt}
@act1_9 module ≔ {Module1}
@act1_10 school ≔ {School1}
@act1_11 content ≔ {Content1}
@act1_12 assignment ≔ {Asgmt1}
@act1_13 pers_info ≔ {PersInfo1}
@act1_14 api_fun_integ ≔ {EPHandAsgmt↦FunHandAsgmt}

@act1_15 usr_info ≔ {User1↦PersInfo1}
@act1_17 module_enrol ≔  {User1↦Module1}
@act1_18 hand_in_asmgt ≔ {User1↦Asgmt1}
@act1_19 has_asgmt ≔  {Asgmt1↦Module1}
@act1_20 has_content ≔ {Content1↦Module1}
@act1_21 has_module ≔ {Module1↦School1}
@act1_22 r_data_usr ≔ ∅
@act1_23 r_data_module ≔ ∅
@act1_24 r_data_asgmt ≔ ∅
end
```

FIGURE 6.36: Initialisation Event in Abstraction Machine

The required configurations and setup are done in Initialisation event (Figure 6.36) for the execution of the case study scenario. For instance, the following are some of the required statements for the scenario.

- User1 is a registered user in the system (*act1_6*).
- Module1 is a module in the system (*act1_9*).
- Asgmt1 is an assignment in the system (*act1_12*).
- User1 is enrolled in Module1 (*act1_17*).
- Asgmt1 is an assignment of Module1 (*act1_19*).

Event-B events and their order is same with events in BSR pattern, detailed in Section 4.2.3. Section 6.2.1.1 shows how the BSR pattern is implemented to model the "Hand in An Assignment" scenario, illustrating the events and their orders in the modelling of the scenario. For instance, based on the pattern implementation, in the beginning, the only enable event is a request initiation to the endpoint (*init_req_hae*) and, following calling the initiated request (*call_hae*), the authorization check of the called request (*authz_ver_hae*). If the authorization outcome is "succeeded", a request to execute the corresponding function is initiated (*init_req_haf*) after the local action in the endpoint proceeds (*local_act_hae*), and so on so forth.

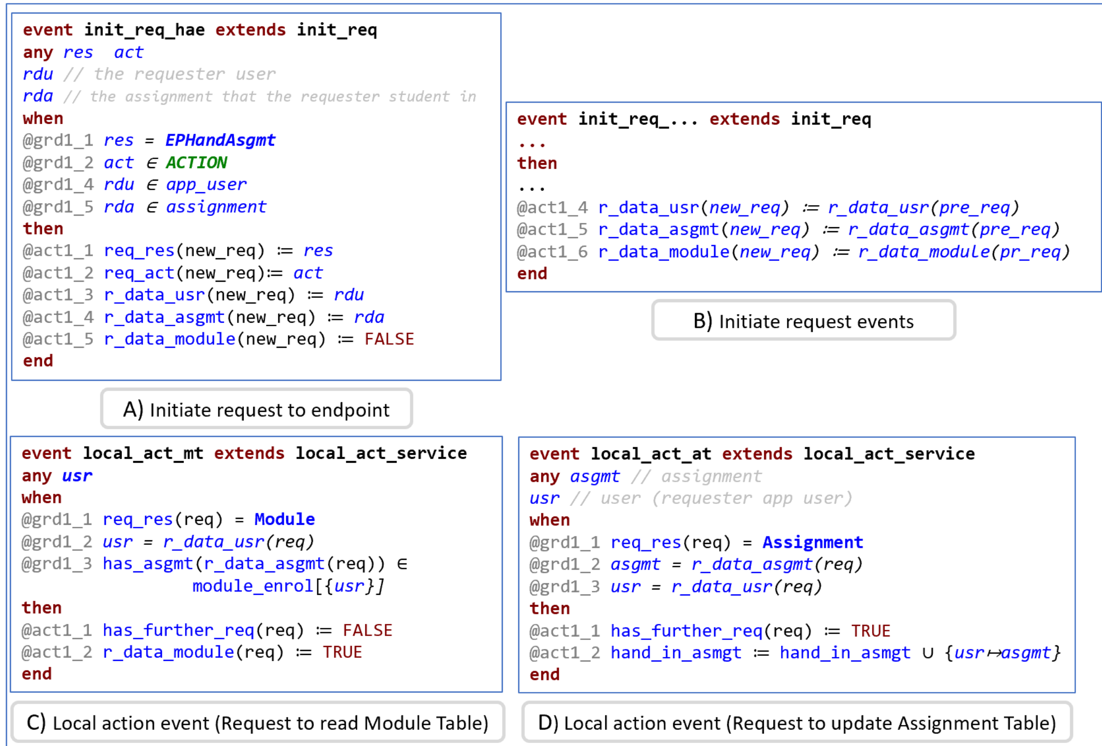


FIGURE 6.37: Data Transition during Among Requests

As mentioned in previous case study scenarios implementation in Chapter 6, the data required for a functionality fulfilment is put in the body of a request while it is initiated. This data is transmitted to the further request if there is a subsequent request.

Moreover, the structure of a request payload body is specific to its corresponding functionality. Invariants in Figure 6.35C introduce the structure of a request body for "Hand in An Assignment" functionality. Furthermore, Figure 6.37 shows how the request payload body is created/updated and transmitted to the further request, which impacts the satisfaction of the functionality. Firstly, in the initation request event of the corresponding endpoint (*init_req_hae* in Figure 6.37A), information about the requester user, the assignment that will be handed in, and enrolment of the requester in module of the assignment are put in the request payload body, *act1_3*, *act1_4*, and *act1_5*, respectively. In further request initiations (Figure 6.37B), the information in the preceding request is transmitted to the new initiated request.

As mentioned before, *r_data_module* gets a Boolen value to show whether the requester app user is enrolled in the module of the assignment that s/he attempts to hand in. In the initiation of the request to endpoint (Figure 6.35A), "FALSE" is assigned. After the request to read the "Module" DB table is successfully executed, in the *local_act_mt* event (Figure 6.35C), the value of *r_data_module* is updated as "TRUE". The *grd1_3* guard in the event ensures that the requester is enrolled in the module of the assignment.

Furthermore, to initiate a request targeting to update assignment, the requester user must enrol in the module of the assignment. To ensure this condition, the *init_req_at* event has an extra

guard (grd1_10), which is
r_data_module(pre_req) = TRUE

Finally, the *local_act_at* event will be available if the authorization result of the request gets "Succeeded" in the *authz_ver_at* event. This can be clearly seen in the textual or visual representation of the implementation of the pattern (Textual Rep.6.3 or Fig.6.33). Then, in the *local_act_at* event (Fig.6.35D), the assignment DB table is updated based on the information in the request body (*act0_3* in Fig.6.37).

To define users who have different access levels, permissions are defined. By defining permission, we can refine the authorization mechanism, which is non-deterministic, into a deterministic authorization mechanism.

Implementing Deterministic Authorization Mechanism

In the AWS cloud environment, each request must satisfy the authorization mechanism to perform the requested action. The *authz_ver_...* in the model represents the authorization mechanism for corresponding requests non-deterministically. Therefore, we follow the approach detailed in Chapter 5 to refine the non-deterministic authorization mechanism into a deterministic one by using permissions. As illustrated in Figure 5.5, app users and functions are defined as actors (*inv2_7*), and their permissions are introduced by associating them with permission statements (*inv2_10*). It is worth noting that endpoint is not an actor, so there is no permission statement directly associated with endpoint. However, a function may have resource-based permission (*inv2_12*) that allows requesters to access the function as a resource. Therefore, when an endpoint makes a request to execute a function, the function must have resource-based permission that allows the requester endpoint to execute the function. Lastly, the structure of a permission statement is shown in Figure 5.5A in Chapter 5.

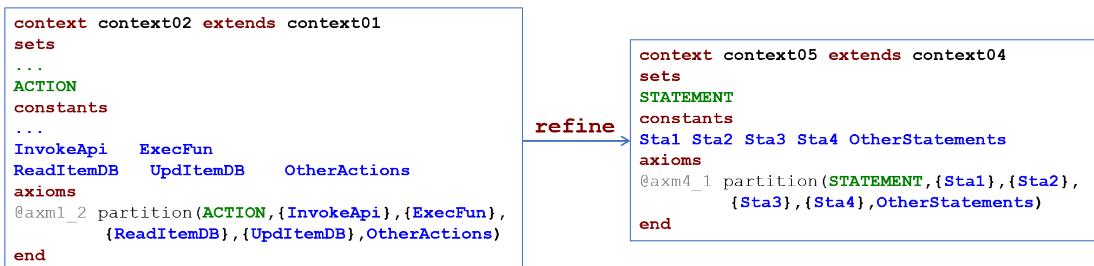


FIGURE 6.38: Permission Entities for "Hand in An Assignment" Scenario

To model a functionality fulfilment, several permissions should be introduced as a setup configuration. Figure 6.38 shows introducing the required actions and statements for "Hand in An Assignment" functionality. For instance, *InvokeApi* represents the action of an endpoint execution, *Execfun* represents the action of a function execution, and so on and so forth. As detailed in Chapter 5, actions are introduced in Context02, while statements are introduced in Context05. Moreover, Figure 6.38 shows the entities specific to the case study scenario.

```
event INITIALISATION extends INITIALISATION
then
@act2_1 requester ≔ ∅
@act2_2 statement ≔ {Sta1, Sta2, Sta3, Sta4}
@act2_3 ext_statement ≔ {Sta2}
@act2_4 sta_res ≔ {Sta1↦EPHandAsgmt,Sta2↦FunHandAsgmt,
                   Sta3↦Module, Sta4↦Assignment}
@act2_5 sta_act ≔ {Sta1↦InvokeApi, Sta2↦ExecFun,
                   Sta3↦ReadItemDB, Sta4↦UpdItemDB}
@act2_6 sta_ef ≔ {Sta1↦Allow, Sta2↦Allow, Sta3↦Allow,
                  Sta4↦Allow}
@act2_7 actor ≔ {User1,FunHandAsgmt}
@act2_8 access_managable_resource ≔ {FunHandAsgmt}
@act2_9 actor_permission ≔ {User1↦Sta1,
                FunHandAsgmt↦Sta3, FunHandAsgmt↦Sta4}
@act2_10 resource_permission ≔ {FunHandAsgmt↦Sta2}
@act2_11 sta_pr ≔ {Sta2↦EPHandAsgmt}
end
```

FIGURE 6.39: Ref 1: Initialisation Event for "Hand in An Assignment" Scenario

The Initialisation event, as detailed in Figure 6.39, sets setup configurations, including configurations for a successful execution of the case study functionality. For instance, for a successful execution, there must be a registered app user who is able to execute the *EPHandAsgmt* endpoint in the system. Therefore, as shown in Figure 6.39, **User1** has **Sta1** (*act2_9*) statement which grants permission to execute *EPHandAsgmt* endpoint (*act2_4*, *act2_5*, *act2_6*).

We follow the refining strategy in Section 5.1.3. Therefore, each *authz_ver_...* event that is for the EPHandAsgmt endpoint, Module DB table, and Assignment DB table is refined into three distinct events, mapping to case 1, case 2, and case 2 of the authorization process shown in Section 5.1.3. *authz_ver_...* event for FunHandAsgmt function is refined into three distinct events, mapping to case 4, case 5, and case 6 of the authorization process since resource-based permissions are needed when an endpoint is a requester.

Implementing Role and Policy features

In the previous refinement step (Section 6.2.1.2), the permission statements and their associations with corresponding requesters and resources are introduced. However, this is an abstraction of the authorization mechanism that works in the AWS cloud environment. To refine this abstraction model to fit the real system, we introduce role and policy entities and their associations, as detailed in Section 5.1.4. The diagram in Figure 5.9 shows how role and policy are implemented in the current abstract model, whereas invariants in Figure 6.18 illustrate introducing them in the refined machine.

```
context context06 extends context05

constants
IamUser     IamRole      IamPolicy

RoleAppStudent      RoleAppLecturer       RoleAppHeadSchool
RoleAppAdmin        RoleFunHandAsgmt      OtherRoles

Pol1     Pol2     Pol3      OtherPols
axioms
@axm5_1 partition(IAM, IamUser, IamRole, IamPolicy)
@axm5_2 partition(IamRole, {RoleAppStudent}, {RoleAppLecturer},
           {RoleAppHeadSchool},{RoleAppAdmin},{RoleFunHandAsgmt},OtherRoles)
@axm5_3 partition(IamPolicy, {Pol1}, {Pol2}, {Pol3},OtherPols)
end
```

Figure 6.40: The required Role and Policy entities in Context

The constants in context in Figure 6.40 introduce the roles and policies that are used in the Learning Management System. For instance, RoleAppLecturer represents the role that defines the rights of lecturers in the system.

```
event INITIALISATION
then
...
@act2_2 statement ≔ {Sta1,Sta2,Sta3,Sta4}
@act2_3 ext_statement ≔ {Sta2}
@act2_4 sta_res ≔ {Sta1↦EPHandAsgmt,Sta2↦FunHandAsgmt,Sta3↦Module,Sta4↦Assignment}
@act2_5 sta_act ≔ {Sta1↦InvokeApi,Sta2↦ExecFun,Sta3↦ReadItemDB,Sta4↦UpdItemDB}
@act2_6 sta_ef ≔ {Sta1↦Allow,Sta2↦Allow,Sta3↦Allow,Sta4↦Allow}
@act2_7 actor ≔ {User1,FunHandAsgmt}
@act2_8 access_managable_resource ≔ {FunHandAsgmt}
@act2_11 sta_pr ≔ {Sta2↦EPHandAsgmt}
@act3_1 iamrole ≔ {RoleAppStudent, RoleFunHandAsgmt}
@act3_2 policy ≔ {Pol1,Pol2,Pol3}
@act3_3 id_policy ≔ {Pol1,Pol3}
@act3_4 res_policy ≔ {Pol2}
@act3_5 actor_role ≔ {User1↦RoleAppStudent,FunHandAsgmt↦RoleFunHandAsgmt}
@act3_6 role_policy ≔ {RoleAppStudent↦Pol1,RoleFunHandAsgmt↦Pol3}
@act3_7 has_access_policy ≔ {FunHandAsgmt↦Pol2}
@act3_8 id_policy_statement ≔ {Pol1↦Sta1,Pol3↦Sta3,Pol3↦Sta4}
@act3_9 res_policy_statement ≔ {Pol2↦Sta2}
end
```

Figure 6.41: Ref 1: Initialisation Event for "Hand in An Assignment" Scenario

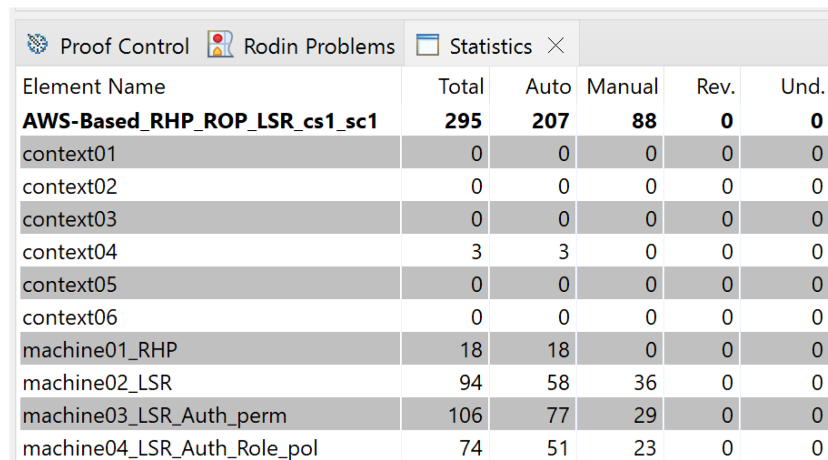
The Initialisation Event in Figure 6.41 introduces setup configurations for "Hand in An Assignment" in terms of role, policies, and their permissions. Here, just the roles and policies of a student to be able to hand in an assignment and their corresponding entities, like FunHandAsgmt Function, and the rights of those policies are introduced.

Lastly, the effect of introducing role and policy on authorization mechanism (*authz_ver_...* events) is detailed in Figure 5.10, while the proposing refinement strategy for modelling authorization mechanism in the AWS cloud environment is detailed in Chapter 5.

6.3 POs of Case Study Scenarios' Models

Proof Obligations (POs) are formal conditions to be verified to ensure the correctness and reliability of an Event-B model. In this section, statistics about the POs of the case studies' models that we developed by using our proposed formal patterns are illustrated.

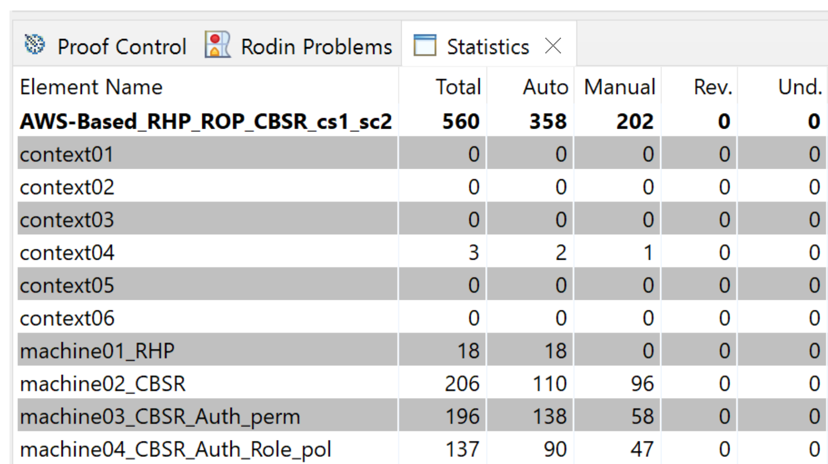
Figure 6.42 depicts the statistics about POs of the first case study scenario, "Update Project Status" in the Project Management System (Section 6.1.1). To model this functionality, the LSR pattern is used.



Element Name	Total	Auto	Manual	Rev.	Und.
AWS-Based_RHP_ROP_LSR_cs1_sc1	**295**	**207**	**88**	**0**	**0**
context01	0	0	0	0	0
context02	0	0	0	0	0
context03	0	0	0	0	0
context04	3	3	0	0	0
context05	0	0	0	0	0
context06	0	0	0	0	0
machine01_RHP	18	18	0	0	0
machine02_LSR	94	58	36	0	0
machine03_LSR_Auth_perm	106	77	29	0	0
machine04_LSR_Auth_Role_pol	74	51	23	0	0

Figure 6.42: POs of The First Case Study Scenario Model

As shown in Figure 6.42, in model of the functionality, the Rodin tool generated 295 POs, of which 71% were automatically discharged.

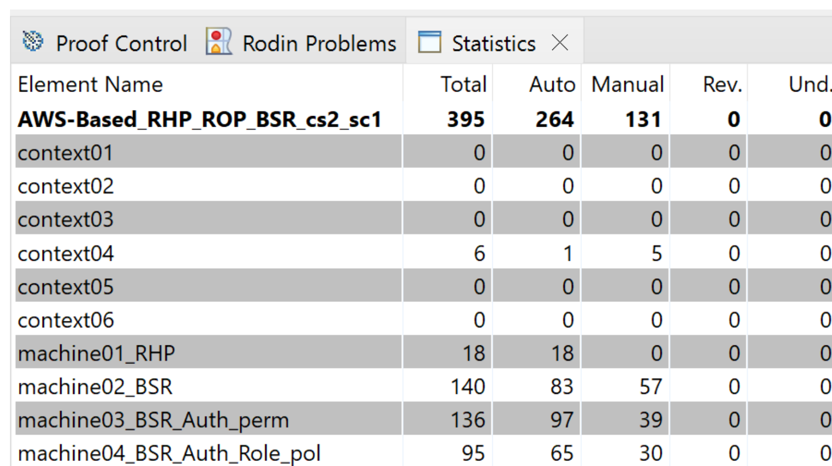


Element Name	Total	Auto	Manual	Rev.	Und.
AWS-Based_RHP_ROP_CBSR_cs1_sc2	**560**	**358**	**202**	**0**	**0**
context01	0	0	0	0	0
context02	0	0	0	0	0
context03	0	0	0	0	0
context04	3	2	1	0	0
context05	0	0	0	0	0
context06	0	0	0	0	0
machine01_RHP	18	18	0	0	0
machine02_CBSR	206	110	96	0	0
machine03_CBSR_Auth_perm	196	138	58	0	0
machine04_CBSR_Auth_Role_pol	137	90	47	0	0

Figure 6.43: POs of The Second Case Study Scenario Model

Moreover, the second case study scenario is "Promoting a User as Project Manager" in Project Management System (Section 6.1.2). CBSR pattern was used to model the second functionality. Figure 6.43 demonstrates POs statistics of the model of the second case study functionality. In

total 560 POs were generated. 358 of them automatically were proved by Rodin tool, whereas 202 of them required a manual trigger to be discharged.



Proof Control | Rodin Problems | Statistics

Element Name	Total	Auto	Manual	Rev.	Und.
AWS-Based_RHP_ROP_BSR_cs2_sc1	**395**	**264**	**131**	**0**	**0**
context01	0	0	0	0	0
context02	0	0	0	0	0
context03	0	0	0	0	0
context04	6	1	5	0	0
context05	0	0	0	0	0
context06	0	0	0	0	0
machine01_RHP	18	18	0	0	0
machine02_BSR	140	83	57	0	0
machine03_BSR_Auth_perm	136	97	39	0	0
machine04_BSR_Auth_Role_pol	95	65	30	0	0

FIGURE 6.44: POs of The Third Case Study Scenario Model

Furthermore, Figure 6.44 shows the statistics of the model of the third case study scenario that is "Updating Project Status" in Learning Management System (Section 6.2.1). The BSR pattern is implemented to model the functionality. Figure 6.44 illustrates the proof obligations that generated by Rodin tool. 264 of them were automatically discharged, while the remaining 131 POs were manually proved.

6.4 Conclusion

In order to perform a certain functionality of a cloud native system, one or more requests to the relevant cloud services must be fulfilled. Each request made in the AWS cloud environment needs to be checked whether the requester is authorised to perform the requested action on the requested resource, which makes the authorization mechanism quite complex. This makes a proper design of the authorization mechanism and the system crucial. Our proposed pattern offers an effective way to model AWS-based cloud native functionalities. Moreover, the reason that multiple scenarios from different domains are used is to show the usability and effectiveness of our proposed patterns to model various serverless system functionalities.

Chapter 7

Conclusion

Service—oriented architecture offers significant benefits in system reusability and security aspects. Firstly, each service focuses on a solution, which can be used in different problem domains. In the cloud—native architecture, services, specifically web services, which are mostly provided by a third—party cloud vendor like AWS, are used to build a/an system/application. This modular approach enhances reusability and alleviates a considerable workload from developers. Moreover, the serverless approach goes a step further by offering serverless functions by cloud providers. Serverless functions that may represent business logic are executed on the cloud without any considerations about managing servers. This enables the creation of backend systems without the need for extensive server-side operations.

Nevertheless, it brings its own set of challenges in the meanwhile, such as those stemming from the complexity of service or function composition and the configuration of access control mechanisms. Those challenges can lead to flaws during application development or configuration. As emphasised in the report referenced in the introduction chapter (Chapter 1), misconfigurations of cloud account users and cloud-native application users have been identified as a prominent factor in cloud incidents. Such flaws and improper configurations can severely compromise an application's vulnerability, leading to potential security breaches, malfunctions, or system failures. As a result, to deal with the inherent complexity of serverless systems, including the configuration of serverless functions and the corresponding access control mechanism, a rigorous approach is required when cloud-native application developers design their systems on serverless cloud platforms.

Therefore, this research focuses on the question of how to deal with the high complexity of the design and configuration of resource management and access control mechanisms in serverless cloud environments. We explore how abstraction and refinement approaches in formal methods could assist cloud-native application developers in effectively managing these complexities during the design phase of serverless systems.

To address this question, the main focus of this research is the development of some generic formal modelling patterns to help cloud-native developers design their systems in serverless

cloud environments. In doing so, we developed RHP and four ROP Event-B based formal modelling patterns to help developers to model their serverless system. RHP helps to model the behaviour of a request that sends/manages to/in a serverless cloud environment, whereas ROP patterns focus on modelling an application level scenario/functionality that may consist of a single or multiple request(s). We also developed a refinement strategy to model authorization mechanism.

7.1 Contributions

The main contribution of this thesis is to develop some generic patterns and refinement strategies for cloud-native app developers to help in modelling their serverless systems using formal methods. The objective of these patterns and refinement strategies is to assist cloud-native app developers in handling the complexity of development and ensuring the robustness of the final application. Specifically, we have developed a set of patterns to incorporate various aspects of serverless application development in a stepwise manner to handle the complexity of this kind of system. Furthermore, the formal refinement and verification incorporate robust resource management to minimise the possibility of security gaps and introduce inconsistencies in the serverless application. Finally, these patterns were applied to formally model various case study scenarios, demonstrating their usefulness and effectiveness.

These contributions can be detailed as follows:

1. Request Handling Pattern (RHP) - A generic formal modelling pattern for modelling the behaviour of a request :

As mentioned before, serverless architecture structures an application as a collection of small, independent functions, each dedicated to a specific task. This structure provides significant flexibility and a high level of fine granularity in system design. In a serverless cloud environment, the smallest business task or functionality might be encapsulated within a single request life-cycle, while more complex functionalities may require a sequence of requests executed in a specific order. Recognising the pivotal role of requests as fundamental building blocks for a functionality that is exposed on a serverless system, we developed the Request Handling Pattern (RHP), as detailed in Section 4.1. The RHP is a generic formal modelling pattern that represents the behaviour of a request life-cycle. Based on the RHP, the lifecycle of a request has been segmented into five distinct phases, each represented by an Event-B event. Those phases are:“initiation of the request”, “calling a service”, “authorization verification check", either “service local action” (if the authorization outcome is "Allow") or “authorization fail”, and “service response”. Both graphical and textual representations of this pattern are provided in Section 4.1.

2. Request Ordering Patterns (ROP) - Generic formal modelling patterns for modelling a functionality :

In a serverless cloud environment, a functionality is exposed through the execution of requests. For the fulfilment of a functionality, the required resource, data, or processes could be provided by either a single request or a set of requests in a specific order. To model a functionality that is fulfilled in a serverless cloud environment, several ROP patterns were developed in Section 4.2. Those patterns represent the order of requests and their corresponding effects during a functionality satisfaction. To develop ROP patterns, the RHP pattern is also used to introduce the behaviour of requests that are required for the fulfilment of the functionality that is modelled.

- **Single Service Request (SSR):** This pattern is ideal for modelling functionalities that require only one single request execution. For example, consider a scenario where a user directly requests a database to update data. The SSR pattern succinctly captures this straightforward, one-step process.
- **Linear Service Request (LSR):** This pattern is developed for the case that multiple requests' execution in linear order is required to perform a functionality. To put it another way, each request in the process of a functionality satisfaction can require at most one further request to be fulfilled. An illustrative example is an application where users access a serverless function to update data in a database table. In this case, a user's request to the function triggers a subsequent request from the function to the database. As a result, to satisfy the requested functionality (update data), two requests are required, and their order must be:
  $Request_{user \mapsto function}, Request_{function \mapsto DB_table}$
- **Branching Service Request (BSR):** The BSR pattern is developed for cases where a single request in the process of a functionality satisfaction might lead to multiple subsequent requests to be fulfilled. For instance, if application users are authorised to execute a serverless function to update data in Table 1 in DB, but the function needs to read data from Table 2 first, the process involves multiple steps. Therefore, to satisfy the functionality, after the app user makes a request to a function, the function sends a request to the DB to read the data in Table 2. Based on the data read, the function sends a second request to database to update the data in Table 1. Thus, the request order must be:
  $Request_{user \mapsto function}, Request_{function \mapsto DB_table_2}, Request_{function \mapsto DB_table_1}$
- **Chained Branching Service Request (CBSR):** During the process of fulfilling a functionality, the order of the required requests may be forked, like in the above case, which is well-suited for the BSR pattern. However, in each branch, multiple requests could be required. For those more complicated cases, we developed a CBSR pattern. This pattern is particularly useful for modelling complicated interactions in serverless systems where functionalities involve complex, multi-step, and branched sequences of requests.

3. A Refinement Strategy for Authorization Mechanism

In serverless cloud environments, specifically in the AWS environment, each request goes through the authorization mechanism to verify whether the requester has the proper permissions to perform the requested action on a specific resource.

In our earlier work (RHP and ROP patterns), we focused on the behaviour of requests and the order and association between them when they fulfil a functionality. In those patterns, we introduced authorization mechanisms at a high level, modelling them with *authz_ver_...* events in a nondeterministic way. In Chapter 5, we refined these non-deterministic events into multiple refined deterministic events by introducing the concept of permissions. In this refinement stage, the model of the authorization mechanism is refined from a high-level abstract view (non-deterministic) to a more detailed and deterministic one. In the subsequent refinement phase, we introduced AWS specific features, namely role and policy, tailoring our model to fit an AWS-based serverless system more accurately.

4. Sub-typing Generalisation in Formal Modelling

During the modelling of functionalities in a serverless cloud environment, we identified commonalities across various elements, such as requests and permissions, which apply to different cloud objects.

For instance, the action of making a request to a serverless function versus a database table. Originally, defining each cloud object as a separate carrier set led to redundant definitions for different types of requests. To address this duplication or redundancy, we adopted a sub-typing generalisation approach, as detailed in Section 5.1.2. In this approach, all objects in the cloud environment are collectively defined as 'RESOURCE', with specific object types like endpoints and database tables differentiated through partitioning. For example, we define different resource types in the model as:

$$@axm1 : partition(RESOURCE, Function, EndPoint \ldots)$$

As a result, we have a generic concept/notation for different types of resources. And then the *req_res* represents the target resource of a request.

$$@inv1 : req_res \in request \rightarrow RESOURCE$$

Therefore, this approach simplifies the model by providing a generic notation for different resource types, enabling the representation of requests to any resource in a cloud environment.

5. Modelling Case Studies by Using The Proposed Patterns

By implementing our developed patterns, several case study scenarios are modelled to show the efficiency and usability of the patterns. We modelled three case study scenarios, two of them from "Project Management System" and one of them from "Learning Management System". Modelling different scenarios from different case study systems illustrates the usability of the developed patterns and approaches. Modelling case studies may also help gain a better understanding of serverless cloud systems.

7.2 Future Works

While the current study has significantly contributed to our understanding of serverless systems, authorization mechanisms in serverless cloud environments, and the use of formal modelling in serverless architecture, it also unveils a spectrum of opportunities for future inquiry.

In this section, we outline some potential directions for future work as follows:

- Modelling more case studies from different domains by using the current developed patterns may help us identify further challenges in serverless environments and formal modelling applications of serverless systems.
- The authentication mechanism could also be crucial for requests in serverless cloud environments. Moreover, it could be quite complicated. For example, in the AWS cloud environment, each request is uniquely signed, and the signature algorithm is highly complex. Therefore, an effective formal modelling approach to modelling this signature algorithm (authentication mechanism) can be developed. Furthermore, this approach might be implemented in our patterns and the refinement approach to the model authorization mechanism.
- Rodin is an effective tool to model in Event-B. A RODIN plug-in can be developed to get a textual or tree-like graphical representation of a ROP pattern with some parameters and then translate them into corresponding Event-B modelling scripts. This may increase the use and efficiency of the proposed patterns.
- As briefly mentioned in Section 2.5, cloud platforms that offer serverless technology may have different structures and concepts in terms of building a serverless system and configuring and implementing authorization mechanisms. More research can be done to investigate serverless architecture on different cloud platforms. Moreover, the proposed patterns could be developed to implement and model the functionalities of different serverless systems built on different cloud environments.

References

[1] Imed Abbassi, Mohamed Graiet, Souha Boubaker, Mourad Kmimech, and Nejib Ben Hadj-Alouane. A formal approach for verifying qos variability in web services composition using event-b. In *2015 IEEE International Conference on Web Services*, pages 519–526. IEEE, 2015.

[2] Kumar Abhishek and Srinivasa Mahendrakar. *Serverless Integration Design Patterns with Azure*. Packt Publishing, 2019. ISBN 978-1-78839-923-4.

[3] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[4] Jean Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. A roadmap for the rodin toolset. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5238 LNCS, page 347, 2008. ISBN 3540876022. .

[5] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International Journal on Software Tools for Technology Transfer*, 12:447–466, 2010.

[6] Gojko Adzic and Robert Chatley. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pages 884–889, New York, New York, USA, 2017. ACM Press. ISBN 9781450351058. . URL http://dl.acm.org/citation.cfm?doid=3106237.3117767.

[7] Microsoft Azure, 2019. URL https://azure.microsoft.com.

[8] Microsoft Azure. Azure api management, 2019. URL https://azure.microsoft.com/en-gb/services/api-management/.

[9] Microsoft Azure. Azure functions, 2019. URL https://azure.microsoft.com/en-gb/services/functions/.

[10] Rahul Kumar B, Thomas Ball, Jakob Lichtenberg, Nate Deisinger, Apoorv Upreti, and Chetan Bansal. CloudSDV enabling Static Driver Verifier Using Microsoft Azure. In

Integrated Formal Methods, volume 1, pages 523–536. Springer International Publishing, 2016. ISBN 978-3-319-33692-3. .

[11] Guillaume Babin, Yamine Aït-Ameur, and Marc Pantel. Web service compensation at runtime: formal modeling and verification using the event-b refinement and proof based formal method. *IEEE Transactions on Services Computing*, 10(1):107–120, 2016.

[12] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design, FMCAD 2018*. IEEE, jan 2018. ISBN 9780983567882. .

[13] Kelly W Bennett and James Robertson. Security in the cloud: understanding your responsibility. In *Cyber Sensing 2019*, volume 11011, page 1101106. International Society for Optics and Photonics, 2019.

[14] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1:81–84, 2014.

[15] Elisa Bertino. Policies, access control, and formal methods. In *Handbook on Securing Cyber-Physical Critical Infrastructure*, pages 573–594. Elsevier Inc., 2012. ISBN 9780124158153. . URL http://dx.doi.org/10.1016/B978-0-12-415815-3.00023-6.

[16] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud Computing : a Study of Infrastructure As a Service (Iaas). *International Journal of Engineering*, 2(1):60–63, 2010. ISSN 0976-0253.

[17] Dines Bjørner. Pinnacles of software engineering: 25 years of formal methods. *Annals of Software Engineering*, 10(1-4):11–66, 2000.

[18] B. W. Boehm, R. K. Mcclean, and D. E. Urfrig. Some experience with automated aids to the design of large-scale reliable software. *IEEE Transactions on Software Engineering*, SE-1(1):125–133, March 1975. .

[19] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, 1995. ISSN 07407459. .

[20] Michael Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009*. Springer, 2009.

[21] Michael Butler. Mastering system analysis and design through abstraction and refinement. *Engineering Dependable Software Systems*, 34:49–78, 2013. .

[22] Michael Butler. Reasoned modelling with Event-B. In *Engineering Trustworthy Software Systems*, volume 10215 LNCS, pages 51–109. Springer, Cham, 2016. ISBN 9783319568409. .

[23] Michael Butler and Stefan Hallerstede. The Rodin Formal Modelling Tool. In *BCS-FACS Christmas 2007 Meeting Formal Methods In Industry London*, pages 1–5, 2007. URL http://eprints.ecs.soton.ac.uk/14949/.

[24] Michael Butler, Asieh Salehi Fathabadi, and Renato Silva. Event-B and Rodin. In Jean-Louis Boulanger, editor, *Industrial Use of Formal Methods: Formal Verification*. Wiley-ISTE, 2012. ISBN 978-1-848-21363-0. URL https://eprints.soton.ac.uk/340229/.

[25] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 5–13. Ieee, 2008.

[26] Miguel A. Calles. Authentication and Authorization. In *Serverless Security: Understand, Assess, and Implement Secure and Reliable Applications in AWS, Microsoft Azure, and Google Cloud*, pages 229–256. Apress, 2020. ISBN 9781484261002.

[27] Hana Chockler, Georg Weissenbacher, David Hutchison, Takeo Kanade, Josef Kittler, Jon M Kleinberg, Friedemann Mattern, John C Mitchell, Moni Naor, C Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, and Gerhard Weikum. Model Checking Boot Code from AWS Data Centers. In *Computer Aided Verification*, pages 467–486. Springer International Publishing, 2018. ISBN 978-3-319-96141-5. . URL http://dx.doi.org/10.1007/978-3-319-96142-2{_}12.

[28] Byron Cook. Formal Reasoning About the Security of Amazon We Services. *International Conference on Computer Aided Verification*, Computer A:38–47, 2018. .

[29] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, 2017. ISBN 978-3-319-67425-4. . URL https://doi.org/10.1007/978-3-319-67425-4_12.

[30] Dropbox, 2019. URL https://www.dropbox.com.

[31] Eclipse, 2019. URL https://www.eclipse.org/.

[32] Adam Eivy. Be Wary of the Economics of 'Serverless' Cloud Computing. *IEEE Cloud Computing*, 4(2):6–12, 2017. ISSN 23256095. .

[33] Albert Endres and H. Dieter Rombach. Requirements definition , prototyping , and modeling. In *A handbook of software and systems engineering: empirical observations, laws and theories*. Pearson Education, 2003.

[34] Peter Feiler, John Goodenough, Arie Gurfinkel, Charles Weinstock, and Lutz Wrage. Four pillars for improving the quality of safety-critical software-reliant systems. Technical report, Carnegie-Mellon University Pittsburgh PA Software Engineering

Inst., 2013. URL https://apps.dtic.mil/dtic/tr/fulltext/u2/a585679.pdf.

[35] Firebase, 2019. URL https://firebase.google.com/.

[36] Flexera. 2019 state of the cloud from rightscale. Technical report, Right Scale, 2019. URL https://media.flexera.com/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf.

[37] Susan J Fowler. *Production-ready microservices: Building standardized systems across an engineering organization.* O'Reilly Media, Inc., 2016.

[38] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[39] Eric Tierling Frank Simorjay. Shared responsibilities for cloud computing. whitepaper, Microsoft Azure, 2019. URL https://azure.microsoft.com/en-gb/resources/shared-responsibility-for-cloud-computing/.

[40] Robert L Glass. *Software runaways-Lessons learned from massive software project failures.* Prentice Hall, 1998.

[41] Joseph A Goguen and Joseph J Tardo. An introduction to obj: A language for writing and testing formal algebraic program specifications. In *Proceedings of the Conference on Specifications of Reliable Software*, pages 170–189. IEEE CS Press, 1979.

[42] Velkoski Goran, Simjanoska Monika, Ristov Sasko, and Gusev Marjan. Business case: From iaas to saas. *SMEs Development and Innovation: Building Competitive Future of South-Eastern Europe*, 801, 2014.

[43] Karl Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to web services architecture. *IBM systems Journal*, 41(2):170–177, 2002.

[44] Antonios Gouglidis, Vincent C. Hu, Jeremy S. Busby, and David Hutchison. Verification of Resilience Policies that Assist Attribute Based Access Control. In *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control*, pages 43–52, 2017. ISBN 9781450349109. .

[45] Synergy Research Group. The leading cloud providers continue to run away with the market. Technical report, Tech. rep, 2017.

[46] Heroku, 2019. URL https://www.heroku.com.

[47] Pooyan Jamshidi, Claus Pahl, Nabor das Chagas Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35: 24–35, 2018.

[48] Eric Jendrock. *Building RESTful Web Services with JAX-RS*, pages 207–218. Pearson Education India, 2011.

[49] Geewax JJ. What is "Cloud". In *Google Cloud Platform in Action*, pages 3–23. Manning Publications, 2018. ISBN 9781617293528.

[50] James E Johnson, David E Langworthy, Leslie Lamport, and Friedrich H Vogt. Formal specification of a web services protocol. *The Journal of Logic and Algebraic Programming*, 70(1):34–52, 2007.

[51] Cliff B Jones. *Systematic software development using VDM*, volume 2. Prentice Hall Englewood Cliffs, 1990.

[52] Ali Kanso and Alaa Youssef. Serverless: Beyond the Cloud. *Proceedings of the 2Nd International Workshop on Serverless Computing*, pages 6–10, 2017. . URL `http://0-doi.acm.org.mylibrary.qu.edu.qa/10.1145/3154847.3154854`.

[53] Etienne J. Khayat and Ali E. Abdallah. A formal model for flat role-based access control. *IFIP Advances in Information and Communication Technology*, 173:233–246, 2005. ISSN 18684238. .

[54] Robert Kowalski and Marek Sergot. A logic-based calculus of events. In *Foundations of knowledge base management*, pages 23–55. Springer, 1989.

[55] Nane Kratzke. A brief history of cloud application architectures. *Applied Sciences (Switzerland)*, 8(8):1–25, 2018. ISSN 20763417. .

[56] Aida Lahouij, Lazhar Hamel, Mohamed Graiet, and Mohammed El Malki. A formal approach for cloud composite services verification. In *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 161–168. IEEE, 2018.

[57] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[58] Leslie Lamport. Who builds a house without drawing blueprints? *Communications of the ACM*, 58(4):38–41, mar 2015. ISSN 00010782. .

[59] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with TLA+. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 45–48, 2002. .

[60] Axel Van Lamsweerde. Formal specification: a Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159. Association for Computing Machinery, 2000. ISBN 1581132530. .

[61] Michael Leuschel and Michael Butler. Prob: A model checker for b. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, pages 855–874, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[62] David S. Linthicum. Practical Use of Microservices in Moving Workloads to the Cloud. *IEEE Cloud Computing*, 3(5):6–9, 2016. ISSN 23256095. .

[63] Venkata Swamy Martha and Maurin Lenglart. *Webservices Engineering*, pages 173–196. Springer Singapore, Singapore, 2019. ISBN 978-981-13-3224-1. . URL https://doi.org/10.1007/978-981-13-3224-1_7.

[64] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017.

[65] Anna Monus. SOAP vs REST vs JSON comparison, 2019. URL https://raygun.com/blog/soap-vs-rest-vs-json/.

[66] Laurence Moroney. Cloud Functions for Firebase. In *The Definitive Guide to Firebase: Build Android Apps on Google's Mobile Platform*, pages 139–162. apress, 2017. ISBN 978-1-4842-2943-9. .

[67] Netsuite, 2019. URL https://www.netsuite.com.

[68] Palo Alto Networks. Unit 42 cloud threat report. Technical report, Palo Alto Networks, 2020. URL https://www.paloaltonetworks.com/prisma/unit42-cloud-threat-research.

[69] Chris Newcombe. Why Amazon chose TLA+. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8477 LNCS:25–39, 2014. ISSN 16113349. .

[70] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the Acm*, 58(4), 2015. .

[71] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: Making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2. . URL http://doi.acm.org/10.1145/1367497.1367606.

[72] Google Cloud Platform. Google app engine, 2019. URL https://cloud.google.com/appengine/.

[73] Google Cloud Platform. Google compute engine, 2019. URL https://cloud.google.com/compute/.

[74] Google Cloud Platform. Google cloud functions, 2019. URL https://cloud.google.com/functions.

[75] Google Cloud Platform, 2019. URL https://cloud.google.com.

[76] Google Cloud Platform. Cloud endpoints, 2019. URL https://cloud.google.com/endpoints/.

[77] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In Jorge Cardoso and Amit Sheth, editors, *Semantic Web Services and Web Process Composition*, pages 43–54, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30581-1.

[78] Irum Rauf, Inna Vistbakka, and Elena Troubitsyna. Formal verification of stateful services with rest apis using event-b. In *2018 IEEE International Conference on Web Services (ICWS)*, pages 131–138. IEEE, 2018.

[79] Abdolbaghi Rezazadeh. *Formal Patterns for Web-based Systems Design*. Phd thesis, University of Southampton, 2006. URL https://eprints.soton.ac.uk/267101/.

[80] Mike Roberts. Serverless architectures what is serverless?, 2018. URL https://martinfowler.com/articles/serverless.html.

[81] Ken Robinson. System modelling & design using event-b. *The University of New South Wales. Recuperado en agosto de*, 2012.

[82] Iman Saleh, Gregory Kulczycki, and M Brian Blake. Formal specification and verification of data-centric service composition. In *2010 IEEE International Conference on Web Services*, pages 131–138. IEEE, 2010.

[83] Asieh Salehi Fathabadi, Michael Butler, and Abdolbaghi Rezazadeh. Language and tool support for event refinement structures in event-b. *Formal Aspects of Computing*, page 499–523, 2015. .

[84] SalesForce, 2019. URL https://www.salesforce.com.

[85] Peter Sbarski. Going Serverless. In *Serversless Architecture on AWS*, pages 3–15. Manning Publications, 2017. ISBN 9781617293825.

[86] Sarah Scalia. A Break in the Clouds. *Weatherwise*, 59(1):42–47, 2006. ISSN 0043-1672. .

[87] Amazon Web Services. Identity-based policies and resource-based policies, 2019. URL https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_identity-vs-resource.html.

[88] Amazon Web Services, 2019. URL https://aws.amazon.com/.

[89] Amazon Web Services. Aws api gateway, 2019. URL https://aws.amazon.com/api-gateway.

[90] Amazon Web Services. Aws cognito, 2019. URL https://aws.amazon.com/cognito.

[91] Amazon Web Services. Aws elastic beanstalk, 2019. URL https://aws.amazon.com/elasticbeanstalk/.

[92] Amazon Web Services. Aws elastic compute cloud, 2019. URL https://aws.amazon.com/ec2.

[93] Amazon Web Services. Aws identity and access management, 2019. URL https://aws.amazon.com/iam/. Accessed: 2024-07-12.

[94] Amazon Web Services. Aws lambda, 2019. URL https://aws.amazon.com/lambda.

[95] Amazon Web Services. Aws lambda limits, 2019. URL https://docs.aws.amazon.com/lambda/latest/dg/limits.html.

[96] Amazon Web Services. Aws simple queue service, 2019. URL https://aws.amazon.com/sqs/.

[97] Amazon Web Services. Aws shared responsibility model, 2020. URL https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/shared-responsibility.html.

[98] Amazon Web Services. Aws services that work with iam, 2023. URL https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_aws-services-that-work-with-iam.html.

[99] Murray Shanahan. The event calculus explained. In *Artificial intelligence today*, pages 409–430. Springer, 1999.

[100] Quan Z. Sheng, Xiaoqiang Qiao, Athanasios V. Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218–238, 2014. ISSN 00200255. .

[101] Praveen Kumar Sreeram. Accelerate Your Cloud Application Development Using Azure Function Triggers and Bindings. In *Azure Serverless Computing Cookbook*, pages 7–34. Packt Publishing, 2017. ISBN 978-1-78839-082-8.

[102] Maddie Stigler. Getting Started. In *Beginning Serverless Computing*, pages 15–40. apress, 2018. ISBN 978-1-4842-3083-1. .

[103] Slobodan Stojanovic and Aleksandar Simovic. Introduction to serverless with Claudia. In *Serverless Applications with Node.js*, pages 3–18. Manning Publications, 2019. ISBN 9781617294723.

[104] Ivan Tarkhanov. Policy algebra for access control in enterprise document management systems. *9th International Conference on Application of Information and Communication Technologies, AICT 2015 - Proceedings*, pages 225–228, 2015. .

[105] Dob Todorov and Yinal Ozkan. Aws security best practices. Technical report, Amazon Web Services, 2013. URL https://d1.awsstatic.com/whitepapers/Security/AWS_Security_Best_Practices.pdf.

[106] Will Venters and Edgar A. Whitley. A critical review of cloud computing: Researching desires and realities. *Journal of Information Technology*, 27(3):179–197, 2012. ISSN 02683962. .

[107] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications*, 11(2):233–247, 2017. ISSN 18632394. .

[108] Inna Vistbakka and Elena Troubitsyna. Towards integrated modelling of dynamic access control with UML and Event-B. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, 271:105–116, 2018. ISSN 20752180. .

[109] William Voorsluys, James Broberg, and Rajkumar Buyya. Introduction to Cloud Computing architecture. In *Cloud Computing: Principles and Paradigms*. John Wiley & Sons, Inc., 2011. ISBN 9780470940105. .

[110] Lizhe Wang, Gregor Von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu. Cloud computing: A perspective study. *New Generation Computing*, 28(2): 137–146, 2010. ISSN 02883635. .

[111] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall International, 1996.

[112] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

[113] Ehtesham Zahoor, Zubaria Asma, and Olivier Perrin. A Formal Approach for the Verification of AWS IAM Access Control Policies. *European Conference on Service-Oriented and Cloud Computing*, Service-Or:59–74, 2017. .

[114] Ehtesham Zahoor, Asim Ikram, Sabina Akhtar, and Olivier Perrin. Authorization policies specification and consistency management with in Multi-Cloud environment. *Nordic Conference on Secure IT Systems*, Secure IT:272–288, 2018. .

[115] Diego Zanon. Understanding the Serverless Model. In *Building Serverless Web Applications*, pages 7–25. Packt Publishing, 2017. ISBN 978-1-78712-647-3.