An Event-B Formal Model for Access Control and Resource Management of Serverless Apps*

Mehmet Said Nur Yagmahan[1[0000−0003−4981−4286]], Abdolbaghi Rezazadeh[1[0000−0002−0029−469X]], and Michael Butler[1[0000−0003−4642−5373]]

School of Electronics and Computer Science (ECS), University of Southampton, Southampton SO17 1BJ, U.K.
{msny1y17, m.j.butler}@soton.ac.uk , ra3@ecs.soton.ac.uk

Abstract. Cloud computing technologies helps developers build scalable distributed apps. Serverless architecture, or Function as a Service (FaaS), which separates app business into multiple functions, is one of the cloud computing concepts that has gained popularity. Those functions can be developed and deployed independently without provisioning infrastructure.

Despite the considerable advantages and increasing popularity of cloud-native apps, developers face many challenges when building their cloud-native applications. To ensure the robustness and security of cloud-native apps and protect crucial resources, the design and implementation of functions and associated access control systems play a pivotal role.

In this paper, we have employed formal methods and tools to develop a set of patterns to help cloud-native application developers design robust serverless systems. We have used Event-B and its associated toolset, Rodin, to construct these formal patterns.

Keywords: Serverless · FaaS · AWS · Access Control · Authorisation · Formal Modelling · Event-B · Formal Methods

1 Introduction

Cloud computing not only facilitates data and resource sharing but also serves as a platform for creating new types of distributed apps, cloud-native apps. Cloud-native apps adopt innovative architectural models such as serverless architecture offering to divide a app business logic into functions that act as a glue between cloud platform resources and services, to meet system requirements.

Access control mechanisms manage resource and function utilization, with variations across cloud platforms. Our focus is on AWS, a leading provider in cloud technology [12].

Defects in system design, especially in access control, can lead to catastrophic issues or significant losses. A report by Synergy Research Group [10] highlights

* The first author is supported by the Turkish National Education Ministry in the U.K. Butler is supported by the HD-Sec project, part of the Digital Security by Design (DSbD) Programme delivered by UKRI to support the DSbD ecosystem.

that 65% of cloud incidents result from customer misconfigurations. This paper introduces a formal approach to model serverless apps and their access control on AWS. By employing the robustness and verification power of formal methods, we aim to provide modeling patterns to guide developers in accurately modeling serverless systems and access control, enhancing system security and reliability.

2 Background

In this section, we first provide an overview of Serverless systems, particularly in the AWS cloud environment. And then, we briefly introduce Event-B.

2.1 Serverless Architecture

Serverless architecture is “an event-driven cloud execution model” [7] that breaks down the business logic of an app into a set of small independent functions [8].

In this architecture, authentication via Cognito and IAM roles enables varied access levels to AWS resources for app user. Authenticated users interact with API Gateway endpoints, each mapping to a serverless app functionality. Endpoint execution depends on user role permissions. After execution, endpoints call related lambda functions, executed in containers to fulfill requests. Moreover the fulfilment may require input from other AWS services, requiring appropriate permissions, often granted via function execution roles (IAM roles).

IAM manages AWS authorization, using IAM identities for authentication and IAM policies for resource authorization. IAM policies define access levels and are categorised into identity-based (granting IAM identities access to resources) and resource-based (attached directly to a resource to manage its own access). For example, to trigger a lambda function via an API Gateway endpoint, the function’s resource-based policy must authorize the endpoint request.

An AWS policy consists of a list of statements, each granting/restricting permission for actions on resources. Through IAM roles whose permission is defined by IAM policies, app users or functions can access resources in AWS. Moreover, “Principal” in resource-based policy’s statement shows who can access the corresponding resource [3].

When a request for a resource is made, AWS evaluates authorization by pulling related policy statements. By default, requests are denied. If at least one allow statement exists, with no deny statement, the request is allowed. Presence of a deny statement overrides allow statements, leading to request rejection.

2.2 Event-B

Event-B [1] is a formal modeling language based on 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 [5]. Although refinement is a common feature in formal methods, the stepwise refinement mechanism in Event-B [1] is a particular feature. To verify the correctness of the model and the consistency between refinement steps, mathematical proofs are used [1].

An Event-B model consists of two components: *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 [6].

Rodin, an open-source toolset, which is developed on top of the Eclipse platform, supports Event-B modelling, refinement, and mathematical proof [2]. There is a considerable set of extensions on the Rodin platform to assist in modelling in Event-B.

3 Modelling Request Handling and Authorization

In this section, we developed a generic pattern for request handling (RHP) and then introducing authorization mechanisms through subsequent refinements.

3.1 Request Handling Pattern

Functionalities of cloud-native apps rely on individual or sequenced requests. To model these applications, modeling request execution is a key aspect.

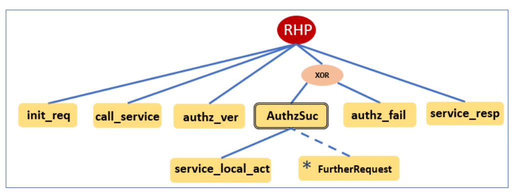

Fig. 1: A Graphical Representation for Request Handling Pattern

The tree-like diagram in Fig. 1 represents the Request Handling Pattern (RHP), detailing request execution stages. Inspired by Fathabadi's work [11], our representation focuses on request sequence and interrelations rather than refinement strategy. Yellow leaves in the RHP diagram represent request lifecycle processes, map to Event-B events. Processes flows left to right, initiating with *init_req* and proceeding to *call_service*. "XOR" connectors indicate decision point based on *authz_ver* outcome, leading to either *service_local_act* or *service_fail*, leading to *service_resp* from the cloud system. For complex scenarios with multiple requests, 'FurtherRequest' (star-marked leaf) indicates RHP for subsequent requests. Dashed lines denote optional entities.

In the model, we define *REQUEST*, *STATUS*, and *EFFECT* sets. *STATUS* includes potential request execution stages (*Initiated*, *Called*, *Succeeded*, *Failed*). *EFFECT* includes potential authorization outcomes (*Allow*, *Deny*).

The variable *req_status* tracks the status of current existing requests (*request*) throughout their execution (*inv0_3*), whereas the *req_authz* variable represents the authorization outcome of requests.

@inv0_3 : req_status ∈ request → STATUS
@inv0_4 : req_authz ∈ request ⇸ EFFECT

Figure 2 shows the Event-B representation of the RHP pattern, in which events map to the yellow leaves in the tree diagram in Fig. 1. Before a request is made, only *init_req* is enabled. The *init_req* event represents the initiation of a new request (*new_req*) by adding it to the *request* variable (*@act0_1* in *init_req* event) and setting its status as *Initiated* (*@act0_2*), which enables the *call_service* event.

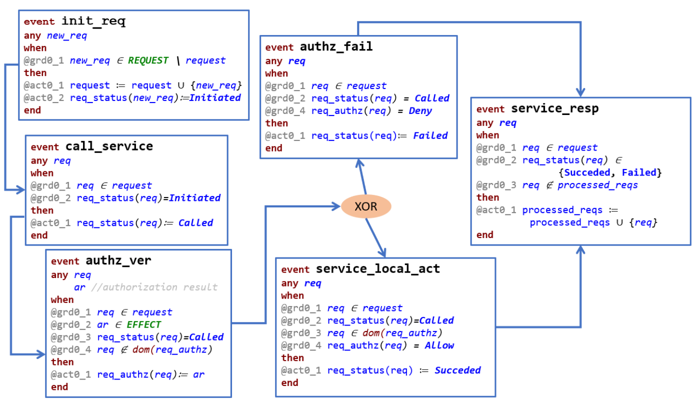

Fig. 2: Request Handling Pattern in Event-B

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. We introduce an event, called *authz_ver* (Fig. 2), to model the authorization mechanism at a high level. In this event, the authorization outcome for a request is assigned non-deterministically by using the parameter *ar*. The type of *ar* is defined by *grd0_2* as *EFFECT*, determining which event is enabled in the next stage.here

3.2 A Unifying Generic Approach for RHP

@axm1_1 : partition(RESOURCE, Cognito, Function, Data, IAM, EndPoint...)

On a cloud platform, each service is a resource, with each access requiring a request. We introduce *RESOURCE* to systematically represent resource hierarchy. Following this, we categorises *RESOURCE*, the abstract concept, into distinct sub-types(*axm1_1*), like functions or data tables. Moreover, *inv1_1* and *inv1_2* define requested resource and action in request context.

@inv1_1 : req_res ∈ request → RESOURCE
@inv1_2 : req_act ∈ request → ACTION

Sub-typing allows us to refine the abstract RHP pattern to represent requests targeting various resources. For example, *req_res* can specify the target resource for any request. However, if cloud services are uniquely defined as different types not subtypes of the same abstract concept, different variables must be defined based on the targeted resource. This complexity poses in using the same pattern for different kinds of requests.

3.3 Refinements Authorization Mechanism

To successfully fulfill a request, the requester needs permission for the requested access. To handle the complexity of AWS authorization mechanisms, we develop an abstract Event-B specification of access control, gradually refining it for detail.

The core idea of this abstraction is to link actors/requesters directly to their corresponding permission statements, bypassing IAM roles and policies for simplicity. This relation shows in Fig. 3B. We used the *actor* variable to represent all AWS entities that can perform action based on their associated permission statements (*inv2_10*), while *access_managable_resouce* is a subset of resource that has permission statements define to access itself.

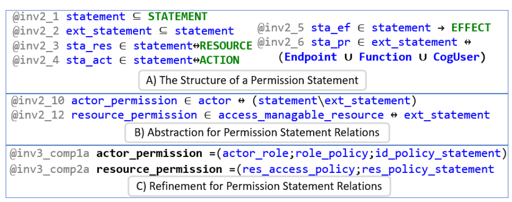

Fig. 3: Event-B Invariants authorization Mechanism

Fig. 3A illustrates the structure of statements. Each statement allows/denies (*inv2_5*) to perform a set of actions (*inv2_4*) on a set of resources (*inv2_3*). In resource-based permission (*inv2_2*), a set of *actor* entities who authorized to access the resource is specified(*inv2_6*).

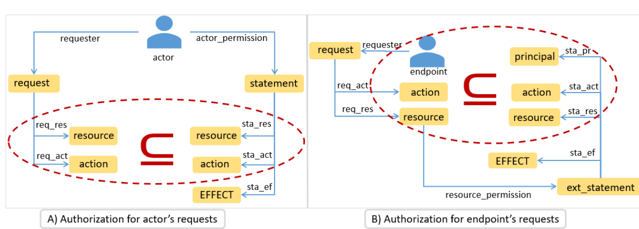

Fig. 4: Authorization of a request

Introducing concrete variables of the access control paves the way for refining authorization rules. Fig. 4 illustrates the authorisation mechanism. A request context consists of the requester (*requester*), the requested action(*req_act*), and the requested resource(*req_res*). Moreover, a *request* will only be allowed if the pairs of *action-resource* in the *request* context are covered by those in the corresponding permission statements. Permission can be actor-based or resource-based, represented by *actor_permission* and *resource_permission*, respectively (Fig.4A and B). Conditions illustrated in Fig.4A and4B are encoded in *inv2_14* and *inv2_15* in Fig. 5.

```
@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)
```

Fig. 5: Event-B Encoding of the Request Authorization

After introducing necessary variables and invariants, we refine the authorization mechanism (*authz_ver*) to make it deterministic. The process distinguishes between actor and endpoint requesters and checks for the presence of corresponding permission.

For example, Fig. 6A shows guards that capture the case that the requester is an actor having an *allow* value in the statement. Guard *grd2_1* ensures that the requester is an actor, while *grd2_3* states that *pm* includes all the requester's statements that are related to the requested action on the requested resource. Moreover, *grd2_4* ensures that there is at least one statement in the *pm* that grants permission, whereas *grd2_5* makes sure there is no deny statement for the requested action in *pm*.

```
@grd2_1 requester(req) ∈ actor
@grd2_3 pm = {st | st ∈ statement ∧
    st ∈ actor_permission[{requester(req)}]∧
    req_res(req) ∈ sta_res[{st}] ∧
    req_act(req) ∈ sta_act[{st}]}
@grd2_4 ∃st·st∈pm ∧ sta_ef(st)=Allow
@grd2_5 ∀st·st∈pm ⇒ sta_ef(st) ≠ Deny
```

A) Permit case for Actor (Ref 2)

```
@grd3_4 pm = {st | st ∈ statement ∧
        st ∈ id_policy_statement[
            role_policy[{
            actor_role(
            requester(req))}]] ∧
        req_res(req) ∈ sta_res[{st}] ∧
        req_act(req) ∈ sta_act[{st}]}
```

B) Permit case for Actor (Ref 3)

Fig. 6: Guards for Deterministic Authorizer Events

In further refinement, IAM roles and policies are introduced. The core idea is to replace the *actor_permission* and *resource_permission* relations with more concrete versions that encompass IAM roles and IAM policies, as defined in *inv3_comp1a* and *inv3_comp2a* in Fig. 3A. In more detail, *actor_permission* is replaced with an actor role that is linked with policies that have a set of statements (*inv3_comp1a*), while *resource_permission* is replaced with a resource-based policy (*res_access_policy*) that has a set of statements (*inv3_comp2a*).

Fig. 6B shows refinement of abstract guards in authorization mechanism events. For example, *grd3_4* replaces abstract *grd2_3*, specifying that *pm* includes all statements of policies of the requester actor's role related to the requested action on the requested resource. [1]

4 Modelling a Specific Serverless App's Scenario

In this section, we focus on modelling of a specific scenario in an AWS-based serverless application —a project management system's "update project status" functionality— to demonstrate our modeling approach.

4.1 Modelling "Update Project Status" Scenario

Typically, the implementation of a serverless app's business logic involves the orchestration of multiple AWS services interacting via Restful APIs. In this scenario, a client application's request initiates an API call to the 'EPUpdProSt' endpoint, initiating the 'FunUpdProData' function's execution to process the request. This function, in turn, updates the 'Project' database table. Fig. 7 demonstrates how this scenario can be modelled by applying the RHP pattern.

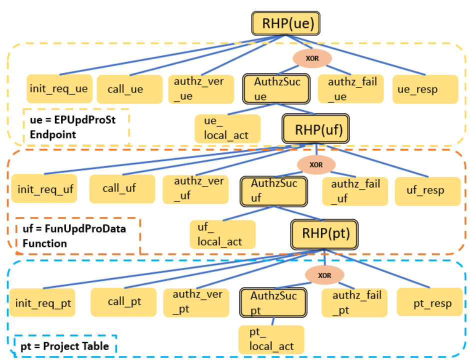

Fig. 7: Use of the RHP pattern to Model a Scenario

In Figure 7, *RHP(ue)* indicates the request targeting to the 'EPUpdProSt' endpoint by an app user, while *RHP(uf)* represents the request sent to the 'FunUpdProData' function by the 'EPUpdProSt' endpoint. Moreover, *RHP(pt)* shows the request by sent the function to update the 'Project' table. This diagram demonstrates that the RHP pattern, as detailed in Fig 1, is capable of being recursively applied to encompass an entire serverless app functionality.

As previously discussed, the authorization mechanism in serverless applications is encapsulated at a high level in the *authz_ver_...* events of the RHP patterns. These events undergo further refinement, detailed in sections 3.3. Due to space limitations, the complete Event-B representation of the entire scenario is not included here but it can be accessible from [13].

[1] Full model for authorization mechanism can be found in [13]

Element Name	Total	Auto	Interactive
machine01_RHP	18	18	0
machine02_LSR	94	58	36
machine03_LSR_Auth_perm	106	77	29
machine04_LSR_Auth_Role_pol	74	51	23
"Update Project Status" Model	295	207	88

Table 1: Proof Control Statistics

Finally, Table 1 shows the proof obligations (POs) statistics in the Event-B model of the "update project status" scenario. POs are formal conditions ensuring model correctness and consistency, categorized across different modeling stages, reflecting the model's evolution and refinement.

5 Related Work

Since 2011, Amazon has implemented formal methods and this assists Amazon engineers in resolving challenging design problems [9]. Amazon security team developed ZELKOVA [4], which is a tool based on formal techniques for customers to check whether their policies has any public access.

There are also some researches focusing on using formal reasoning to find conflicts in authorization policies in cloud systems. [14] models IAM policies to find conflicts between policy statements, while [15] extends the work to cover conflicts in policy statements from multi-cloud environments.

Both [14] and [15] focus on conflicts in a policy statements, whereas ZELKOVA [4] checks only the openness of a policy in AWS environment. However, our work focuses on modelling functionalities of serverless apps built on an AWS environment and corresponding access control configurations. The aim is to help developers design their serverless systems rigorously.

6 Conclusion and future work

Serverless systems, characterized by their highly distributed and complex nature, necessitate rigorous formal design to ensure robustness. Formal modeling clarifies system and environment understanding, aiding in complexity management through the strategic use of abstraction.

We identify some key aspects for modelling a serverless system built in the AWS cloud environment including a generalization and sub-typing approach, an abstraction strategy for the authorization mechanism, and the use of RHP pattern to model a scenario. Moreover, the automation of pattern instantiation represents a future direction in our research.

Furthermore, we plan to extend our study to include threat modelling scenarios in our models. This will allow us to identify potential security vulnerabilities systematically and address them by integrating appropriate controls into the model.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010). https://doi.org/10.1017/CBO9781139195881
2. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: A Roadmap for The Rodin Toolset. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (2008). https://doi.org/10.1007/978-3-540-87603-8$_3$5
3. Amazon Web Services Inc: Aws services that work with iam (2022), https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_aws-services-that-work-with-iam.html
4. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: 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 (2018). https://doi.org/10.23919/FMCAD.2018.8602994
5. Butler, M.: Mastering system analysis and design through abstraction and refinement. Engineering Dependable Software Systems (2013). https://doi.org/10.3233/978-1-61499-207-3-49
6. Butler, M., Fathabadi, A.S., Silva, R.: Event-B and Rodin. In: Industrial Use of Formal Methods: Formal Verification. Wiley-ISTE (2012). https://doi.org/10.1002/9781118561829.ch7
7. Kanso, A., Youssef, A.: Serverless: Beyond the Cloud. Proceedings of the 2Nd International Workshop on Serverless Computing (2017). https://doi.org/10.1145/3154847.3154854
8. McGrath, G., Brenner, P.R.: Serverless computing: Design, implementation, and performance. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE (2017). https://doi.org/10.1109/ICDCSW.2017.36
9. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services Uses Formal Methods. Communications of The ACM (2015). https://doi.org/10.1145/2699417
10. Palo Alto Networks: Unit 42 cloud threat report (2020)
11. Salehi Fathabadi, A., Butler, M., Rezazadeh, A.: Language and tool support for event refinement structures in event-b. Formal Aspects of Computing . https://doi.org/10.1007/s00165-014-0311-1
12. Synergy Research Group: Huge cloud market still growing at 34% per year; amazon, microsoft & google now account for 65% of the total (apr 2022), https://tinyurl.com/yhy7hdwy
13. Yagmahan, M.S.N., Rezazadeh, A., Butler, M.: Rhp (request handling pattern) and authorization mechanism in event-b (June 2024). https://doi.org/https://doi.org/10.5258/SOTON/D3041
14. Zahoor, E., Asma, Z., Perrin, O.: A Formal Approach for the Verification of AWS IAM Access Control Policies. European Conference on Service-Oriented and Cloud Computing (2017). https://doi.org/10.1007/978-3-319-67262-5$_5$
15. Zahoor, E., Ikram, A., Akhtar, S., Perrin, O.: Authorization Policies Specification and Consistency Management with in Multi-Cloud Environment. Nordic Conference on Secure IT Systems (2018). https://doi.org/10.1007/978-3-030-03638-6$_1$7