From Formal SYMBOLEO Specifications to Secure and Interactive Smart Contract Code

by

Sofana Alfuhaid

A thesis submitted to the
University of Ottawa
In partial fulfillment of the requirements
for the Ph.D. degree in
Digital Transformation and Innovation

School of Engineering Design and Teaching Innovation
Faculty of Engineering
University of Ottawa

Abstract

Context: Legal contracts have served as the bedrock of business transactions for millennia. They are core to modern supply chains, and their execution can now be automated through the use of i) smart contracts, supported by blockchain technology that safeguards data integrity, and ii) Internet-of-Things technologies to support their monitoring functions. SYMBOLEO is a specification language used to formalize legal contracts, enable property analysis, and generate smart contracts for a permissioned blockchain platform (Hyperledger Fabric). However, automation around resulting smart contracts poses security challenges, particularly regarding who should have access to operate on contract elements. Additionally, how such smart contract should interact with their Cyber-Physical System (CPS) environment, including IoT devices, remains challenging.

Purpose: The thesis proposes an architecture to integrate smart contracts, Complex Event Processing (CEP), message brokers, and a blockchain platform (namely Hyperledger Fabric) to support end-to-end Cyber-Physical Smart Contracts (CPSCs). This architecture makes it possible to connect IoT devices with smart contracts (generated using SYMBOLEO) through a CEP engine and a message broker. Additionally, this thesis proposes an access control model, treating all contract elements as resources and ensuring regulated access by designated parties. This model extends the SYMBOLEO ontology and language for legal contracts with new modeling concepts inspired by Role-Based Access Control (RBAC), tailored for the legal contract domain, resulting in SYMBOLEOAC (SYMBOLEO Access Control). SYMBOLEOAC also extends the SYMBOLEO language to handle dynamic contract execution scenario.

Methodology: This research follows a Design Science Research methodology, which guides the development and evaluation of the research artifacts. This research is conducted in several iterative steps that are divided into two main phases, one that focuses on theoretical aspects and the other on the design, demonstration, and evaluation of the research artifacts.

Contributions: The contributions of this thesis are:

- An architectural framework for CPSCs that leverages complementary aspects of CPS and smart contracts;
- SYMBOLEOAC, an access control ontology for SYMBOLEO;
- An extension of the current SYMBOLEO specification language (syntax and semantics) that supports smart contract requirements, including automation and control actions, access control, and CPS components;
- An implementation of the SYMBOLEOAC ontology and semantics into a reusable JavaScript library (SYMBOLEOACJS), together with a tool, SYMBOLEOAC2SC, that generates JavaScript smart contract code with security aspects for a designated platform (Hyperledger Fabric); and

- A secure and event-driven SymboleoAC Application Programming Interface (API) that orchestrates the runtime ecosystem connecting IoT sensors, the message broker, the CEP engine, and the blockchain platform.

Through extensive and the evaluation of multiple variations of two contract case studies, SymboleoAC (architecture, ontology, and language), along with its associated tools, is shown to be an effective environment for CPSCs, simplifying the design of secure smart contracts and their connections to message brokers, CEP engines, and IoT devices.

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Prof. Daniel Amyot, Prof. John Mylopoulos, Dr. Amal Ahmed Anda. Their invaluable guidance, encouragement, and expertise have been instrumental in shaping this work and my development as a researcher. I am incredibly grateful for their support and patience throughout this journey. I am also thankful to Prof. Luigi Logrippo (Université du Québec en Outaouais) and Prof. Marco Roveri (University of Trento) for useful discussions and collaborations throughout this journey, to Dr. Alireza Parvizimosaed and Aidin Rasti for their help with tool issues, and to many other students who crossed my path and helped me during my doctoral studies.

In addition, I would like to thank for their support and constructive feedback the members of my thesis advisory committee, namely Prof. Tet Yeap (who also supervised my master's thesis) and Prof. Morad Benyoucef, as well as my examiners, namely Prof. Mohammad Hamdaqa (École Polytechnique de Montréal) and Prof. Timothy C. Lethbridge.

I am also deeply thankful to my home country, the Kingdom of Saudi Arabia, for its unwavering support and the scholarship that has enabled me to pursue my academic goals. This work would not have been possible without this investment in my future.

My heartfelt thanks go to my family – my parents, whose love and sacrifices have always driven me to strive for excellence; my grandmother, whose wisdom and encouragement are a source of constant inspiration; and my brothers, whose support mean the world to me.

To all of you, thank you for your endless support throughout this journey.

This thesis was partially funded by a scholarship from King AbdulAziz University, by Prof. Mylopoulos' NSERC Strategic Partnership Grant titled *Middleware Framework and Programming Infrastructure for IoT Services* and NSERC Discovery Grant titled *From Legal Contracts to Smart Contract*, and by the ORF-RE Grant *CyPreSS: Software Techniques for the Engineering of Cyber-Physical Systems*.

Dedication

To my beloved parents, grandmothers, and brothers, whose unwavering love and support during my studies have inspired me every day.

To those who stood beside me in silence and in strength, whose kindness, encouragement, and presence carried me through the most difficult moments.

Table of Contents

List of Tables

List of Figures

List of Listings

List of Acronyms

AC	Access Control
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CA	Certificate Authority
CEP	Complex Event Processing
CoAP	Constrained Application Protocol
CPS	Cyber-Physical System
CPSC	Cyber-Physical Smart Contract
DLT	Distributed Ledger Technology
DSL	Domain-Specific Language
DSR	Design Science Research
EMF	Eclipse Modeling Framework
EPL	Event Pattern Language
FDA	U.S. Food and Drug Administration
IDE	Integrated Development Environment
IoT	Internet-of-Things
IPFS	Interplanetary File System
LLM	Large Language Model
LOC	Lines of Code
MCDC	Medical CBRN (Chemical, Biological, Radiological and Nuclear) Defense Consortium Transport

MQTT	Message Queuing Telemetry Transport
mTLS	mutual TLS
NFR	Non-Functional Requirement
OCL	Object Constraint Language
PEM	Privacy Enhanced Mail
PRISMA	Preferred Reporting Items for Systematic reviews and Meta-Analyses
RBAC	Role-Based Access Control
REST	Representational State Transfer
SC	Smart Contract
SymboleoAC	Symboleo Access Control
SymboleoAC2SC	SymboleoAC to Smart Contract tool
SymboleoACJS	SymboleoAC JavaScript library
SymboleoJS	Symboleo JavaScript library
SymboleoPC	Symboleo Property Checker tool
Symboleo2SC	Symboleo to Smart Contract tool
TLS	Transport Layer Security
UFO-L	Unified Foundational Ontology-Legal
UML	Unified Modeling Language

Chapter 1

Introduction

This thesis aims to advance the generation of smart contracts from SYMBOLEO specifications by integrating action-based constructs into the language, enabling the semi-automated execution of legal contracts, and supporting security requirements through access control. This thesis further aims to develop an architecture for Cyber-Physical Systems (CPSs) where such smart contracts will be deployed, leading to *Cyber-Physical Smart Contracts (CPSCs)* that support effective and access-controlled interactions between physical and digital components.

This chapter begins by presenting the context and motivation behind this work, followed by the research questions and goals. Additionally, it provides a brief overview of the selected methodology, which is based on Design Science Research (DSR), and of the contributions and main limitations of the thesis.

1.1 Problem Context

A *legal contract* is a document containing various contractual clauses that formulate relationships between parties in the real world in a way that supports shared understanding of and commit to business transactions. *Smart contracts* (SCs) are programs intended to partially encode, automate, and control some aspects of the execution of legal contracts, monitor them for compliance with relevant terms and conditions, and intervene when contractual agreements are breached [117]. A critical issue for SCs is the integrity of the data they handle. To address this issue, many SCs make use of blockchain technology for storing both programs and data, providing a high level of resilience against data integrity attacks [6]. Embedding smart contracts in larger Cyber-Physical Systems that contain Internet-of-Things (IoT) sensors and actuators provides also an opportunity to monitor and enforce the correct execution of contractual agreements with applications in various industries, such as energy, supply chain, and transportation, to name a few [98].

However, the above industries are increasingly dynamic and driven by continuous IoT data generated across distributed cyber-physical environments. The challenge is not only

about collecting such data, but also about ensuring trusted interactions, contractual compliance, authenticated communication, and timely decision-making across multiple organizations and heterogeneous systems. This creates a growing need for architectures and tools that can support secure and reliable event-driven smart contract execution while integrating real-time monitoring and automated enforcement mechanisms.

As a supply chain example, consider a meat sale transaction between a supplier (the seller) and a supermarket (the buyer). The contract includes obligations for both parties: the supplier must deliver meat of specified quality, while the buyer must pay before shipment. IoT sensors monitor factors like temperature (e.g., keeping meat below 5°C) and location during transport. Sensor data enables the smart contract to track compliance, triggering a breach notification if required conditions are not met. This integration of IoT and smart contracts demonstrates how CPS technology enforces real-time contract compliance, building trust and accountability.

For such contexts, monitoring the execution of legal contracts that involve communication between physical objects and software in CPSs is complex due to dependencies between CPS components. A major challenge here is ensuring that the contractual terms and conditions have happened as agreed upon.

SYMBOLEO is a formal language, based on an ontology, state machines, and event calculus, for specifying the terms and conditions of many types of business-oriented legal contracts. In particular, SYMBOLEO uses the concepts of *obligation* (something that must happen between the parties involved) and *power* (the right of a party to create, suspend, or cancel obligations and other powers under some condition) for contract monitoring and compliance [92, 109]. The language is supported by:

- SYMBOLEOWEB [81], an Integrated Development Environment (IDE) for editing specifications;
- The SYMBOLEO Property Checker tool (SYMBOLEOPC) [90], an analysis tool for checking temporal logic properties, at design time;
- The SYMBOLEO to Smart Contract tool (SYMBOLEO2SC) [100], a compiler that generates smart contract code that monitors contract executions for compliance against the specification, at run-time.

1.2 Motivation

In the age of digitalization, there have been numerous efforts aiming to enable the automation of legal contracts, ranging from e-contracts to the more recent development of smart contracts [40]. Smart contracts as Cyber-Physical Systems have been widely used with distributed ledger technologies [6]. Such smart contracts operate by responding to triggers that are provided by users or by the surrounding environment (e.g., IoT sensors), and their primary purpose is to monitor the execution of legal contracts or business transactions [40].

However, the typical design architecture of a CPSC, as shown in Section 3.4, reveals unclear connections between physical and cyber components, making it difficult to validate smart contracts against legal contracts. Additionally, the design of one component often affects the others, and vice versa; such interdependency often results in smart contracts that are prone to errors. Their quality needs to be assessed due to the complex interactions between physical and cyber components, as well as the lack of proper coding abstractions and tools. Meanwhile, developers typically code smart contracts manually, increasing the risk of errors. These issues are supported by the findings presented in the literature review (Chapter 3).

Although the SYMBOLEO language and its supporting tools provide a useful environment in that context, they fall short of addressing important issues such as:

- Integrating smart contracts generated from specifications capturing legal contracts into a larger CPS ecosystem, enabling effective interactions with the external world (including IoT devices).
- Generating smart contracts that not only monitor the execution of legal contracts, but also *execute* these legal contracts (or part thereof).
- Supporting important quality requirements, especially around security and privacy.

Solving these issues would make SYMBOLEO-based solutions more attractive and effective for practitioners seeking to automate and enforce legal contracts as CPSCs. Additionally, improved quality and faster coding capabilities would streamline development, enabling practitioners to create secure and reliable smart contracts more efficiently.

1.3 Research Goals

The long-term vision for this research is to develop conversion tools for legal contract specifications that allow practitioners to generate smart contracts that can monitor (for compliance) and execute legal contracts while satisfying important and customizable quality criteria.

The goal of this thesis is to develop and validate a conversion tool for transforming SYMBOLEO specifications into smart contract code that can monitor and partially execute legal contracts on one specific platform (namely Hyperledger Fabric [13], using JavaScript), in a CPS for a predefined set of quality criteria (namely security and privacy).

As a strategy, we intend to satisfy this goal by:

- Introducing an architecture for embedding SYMBOLEO-generated SCs in larger CPSs;
- Supporting executable SYMBOLEO specifications by integrating action-based constructs into the language;

- Designing rules that convert SYMBOLEO specifications supplemented with security/privacy quality criteria into a target SC language (namely JavaScript); and
- Assessing the efficiency of the tool-supported conversion process in terms of coding, deployment, and other concerns.

As of today, SYMBOLEO is likely the most advanced language for specifying legal contracts, with the best analysis and code generation tools [90, 101].

1.4 Research Questions

The following research questions (**RQ1-RQ4**) are derived from the goal of this thesis.

RQ1: What is the CPS architecture needed to deploy legal smart contracts?

The question focuses on the design and configuration of the CPS architecture necessary to support the deployment and operation of legal smart contracts, especially around the monitoring and control of the CPS environment. This question involves studying and selecting the various components, protocols, and interactions required to establish a robust and efficient CPSC architecture.

RQ2: How can SYMBOLEO specifications be extended to generate smart contract code that monitors for compliance the execution of a contract, semi-automates selected steps of the execution, and controls the execution?

This question involves extending the existing SYMBOLEO language to SYMBOLEO Access Control (SYMBOLEOAC), a new version of the language (including its ontology) aiming to facilitate the automation and control of contract executions. While the current version of SYMBOLEO supports producing code that monitors and verifies legal contract execution, it falls short in addressing other critical requirements of smart contract systems (e.g., sending notifications). By introducing such extensions, SYMBOLEOAC would provide a means to express and enforce obligations and powers that pertain to the behavior and functionality expected from the smart contract system as a whole.

RQ3: How can SYMBOLEO be extended to specify security and privacy quality criteria when assessing compliance?

This question aims to determine how SYMBOLEO's language and code generator can take into account these quality criteria to ensure that the resulting system not only monitors compliance effectively but also meets the desired levels of security and privacy. This involves exploring techniques and approaches to assess and enhance the converted SYMBOLEO specification, including around access control.

RQ4: How can the newly extended SYMBOLEO specifications be converted into cyber-physical smart contracts using automated code generation?

This question focuses on automating the conversion of the newly extended SYMBOLEO specifications (i.e., in SYMBOLEOAC) into legal smart contracts deployable on the selected CPSC architecture. This indirectly involves the conversion of the underlying SYMBOLEOAC ontology into a reusable library, as done previously by Rasti [101] for plain SYMBOLEO.

1.5 Methodology Overview

This thesis follows a Design Science Research methodology [50], which guides the development and evaluation of the research artifacts. This research is conducted in several iterative steps that are divided into two main phases, one that focuses on theoretical aspects and the other on the design, demonstration, and evaluation of the research artifacts. Detailed information on the research methodology, including the development of artifacts and their evaluation, is presented in Chapter 4.

1.6 Contributions

The primary contributions of this thesis include (see Figure 1.1):

1. A CPSC architectural framework that provides an integrated solution leveraging CPS and SC aspects, including message broker and Complex Event Processing (CEP) components supporting interactions between SCs and IoT devices (**RQ1**).
2. An access control extension of SYMBOLEO (**RQ3**), that includes:
   - A generic access control ontology with Role-Based Access Control (RBAC) primitives (**RQ3-1**).
   - An integrated SYMBOLEOAC ontology that combines the RBAC ontology with SYMBOLEO's (**RQ3-2**).
3. SYMBOLEOAC, an extension of the SYMBOLEO specification language (syntax and semantics) (**RQ2**), that supports:
   - Variable assignment in events used by contractual obligations and powers (**RQ2-1**).
   - Notifications sharing contract state knowledge (including violations) as events during contract execution (**RQ2-2**).
   - The SYMBOLEOAC ontology (❶) with RBAC constructs addressing privacy and security concerns (**RQ3-3**).

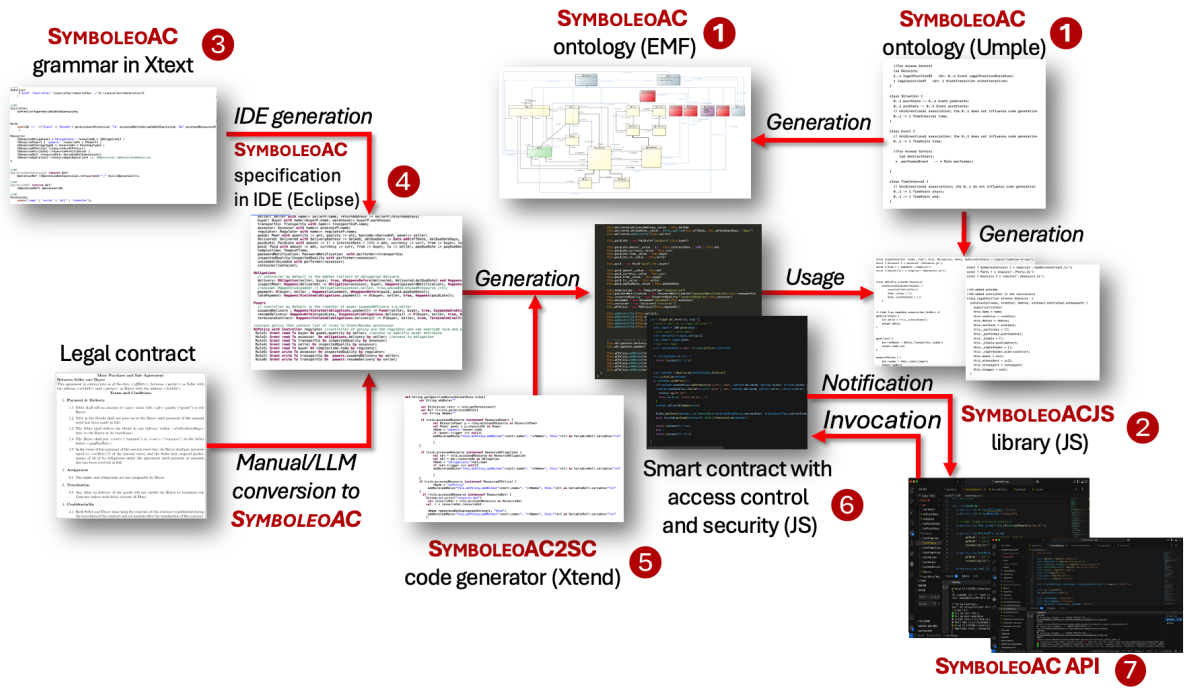


Figure 1.1: SymboleoAC overview.

- Controller rules for each resource derived from the semantics of legal concepts, as well as pre-authorization rules that give access to resources for different roles of a legal contract (**RQ3-4**), and supported by the SymboleoAC grammar (3) and editor (4).

4. A code generator that extends Symboleo2SC with support for SymboleoAC (**RQ4**), which includes:

   - The SymboleoAC JavaScript library (SymboleoACJS) 2, a JavaScript library featuring utility classes and methods that embody the ontology and semantics of SymboleoAC (**RQ4-1**).
   - The SymboleoAC to Smart Contract tool (SymboleoAC2SC) 5, a compiler that generates smart contract code (6) (in JavaScript for the Hyperledger Fabric platform) from SymboleoAC specifications, including the access control model and rules. The code also interoperates with the components of the CPSC architectural framework mentioned earlier (message broker and CEP) (**RQ4-2**).

5. An access-controlled and event-driven SymboleoAC Application Programming Interface (API) that orchestrates the runtime ecosystem connecting IoT sensors, the message broker, the CEP engine, and the smart contract deployed on a blockchain platform 7. This API, implemented in JavaScript and Java, coordinates real-time event publication, subscription, filtering, and smart contract transaction invocation through an integrated gateway and wallet mechanism, ensuring controlled, authenticated, and authorized execution of CPSCs across on-chain and off-chain components of the SymboleoAC architecture (**RQ1**).

Secondary contributions include:

1. A mapping review of architectures, platforms, and challenges related to Cyber-Physical Smart Contracts, already published [6].
2. An illustrative contract example (Vaccine Procurement) that exploits the new SymboleoAC features.

Overall, this thesis proposes a framework for engineering cyber-physical smart contracts, consisting of a specification language (SymboleoAC), supporting tools for automated code generation and execution (SymboleoAC2SC), and an integrated architecture connecting on-chain and off-chain components (SymboleoAC CPSC). The feasibility and effectiveness of the proposed framework are demonstrated through two case studies, which illustrate how legal contracts can be specified, transformed into executable smart contracts, and deployed in a cyber-physical environment.

Additionally, This thesis has already led to eleven Symboleo-related publications (including five as first and main author), co-authored with my co-supervisors and other members of our research laboratory.

Journal Publications

- **Sofana Alfuhaid**, Daniel Amyot, Amal Ahmed Anda and John Mylopoulos, "A Mapping Review on Cyber-Physical Smart Contracts: Architectures, Platforms, and Challenges". *IEEE Access*, vol. 11, pp. 65872–65890, 2023, http://doi.org/10.1109/ACCESS.2023.3290899 [6]
  - This paper represents the core of the literature review in Chapter 3.
- Alireza Parvizimosaed, Marco Roveri, Aidin Rasti, Amal Ahmed Anda, **Sofana Alfuhaid**, Daniel Amyot, Luigi Logrippo, and John Mylopoulos, "SymboleoPC: checking properties of legal contracts". *Software and Systems Modeling*, vol. 24, pp. 1093–1126, 2025, https://doi.org/10.1007/s10270-024-01180-2 [90]
  - In this paper, I contributed an extension of Symboleo for variable assignment, discussed in Section 8.2, with code generation for the nuXmv language (not discussed in this thesis).
- Aidin Rasti, Amal Ahmed Anda, **Sofana Alfuhaid**, Alireza Parvizimosaed, Daniel Amyot, Marco Roveri, Luigi Logrippo, and John Mylopoulos, "Automated generation of smart contract code from legal contract specifications with Symboleo2SC", *Software and Systems Modeling*, vol. 24, pp. 1127–1156, 2025, https://doi.org/10.1007/s10270-024-01187-9 [101]
  - I contributed an extension of Symboleo for variable assignment (Section 8.2), with code generation for the JavaScript language (Section 9.4.8), and the Symboleo specification of a vaccine procurement contract (Section 10.3.3).

- **Sofana Alfuhaid**, Amal Ahmed Anda, Daniel Amyot, Marco Roveri, John Mylopoulos, "SymboleoAC: An Access Control Model for Smart Legal Contracts". *Software and Systems Modeling*, 2025, https://doi.org/10.1007/s10270-025-01327-9 [8]
  - This is the main paper describing SymboleoAC (Chapters 7 and 8) and the corresponding code generator for smart contracts Chapter 9).

Conference Publications

- **Sofana Alfuhaid**, Amal Ahmed Anda, Daniel Amyot, Marco Roveri, and John Mylopoulos, "SymboleoAC: An Access Control Model for Legal Contracts". In: *17th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling (PoEM)*, Stockholm, Sweden, 2024. LNBIP 538, Springer, pp. 227–243. https://doi.org/10.1007/10.1007/978-3-031-77908-4_14 [7]
  - This paper introduced a first version of SymboleoAC; it was invited for a journal extension [8], already discussed above.
- **Sofana Alfuhaid**, "Towards Secure and Interactive Smart Contract Code from Formal Symboleo Specifications". In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Ottawa, Canada, 2025. IEEE CS, pp. 58–62. http://doi.org/10.1109/ICSE-Companion66252.2025.00024 [4]
  - This doctoral symposium paper provides an early overview of this thesis, including its research methodology (Chapter 4).
- Regan Meloche, Durga Sivakumar, Amal Ahmed Anda, **Sofana Alfuhaid**, Daniel Amyot, Luigi Logrippo, and John Mylopoulos, "A Web-Based Environment for the Specification and Generation of Smart Legal Contracts". In: *Compliance for Artificial Intelligence Systems: Strategies, Principles and Methods*, LNAI 14377, 2026. Springer, pp. 1–13, https://doi.org/10.1007/978-3-032-12795-2_2 [81]
  - This paper presents a Web-based editor for Symboleo that I helped develop, deploy, and maintain (not discussed in this thesis).

Workshop Publications

- Daniel Amyot, Luigi Logrippo, John Mylopoulos, Marco Roveri, Amal Ahmed Anda, Alireza Parvizimosaed, **Sofana Alfuhaid**, Sepehr Sharifi, Regan Meloche, and Daniel Sousa-Dias, "Engineering Smart Contracts with Symboleo: A Progress Report". In: *CyPress: 3rd Workshop on Software Techniques for Engineering Cyber-Physical Systems*, CASCON'23, Las Vegas, USA. IBM Corp., pp. 235–237, 2023, https://dl.acm.org/doi/10.5555/3615924.3623631 [12]
  - This is a summary of yearly improvements to the Symboleo ecosystem, where I had minor contributions (covered implicitly in this thesis).

- John Mylopoulos, **Sofana Alfuhaid**, Daniel Amyot, Amal Ahmed Anda, Luigi Logrippo, Regan Meloche, Ashkan Rahimi-Kian, Sahil Rajpal, Marco Roveri, Durga Sivakumar, Daniel Sousa-Dias, "Engineering Smart Contracts with Symboleo: Progress Report 2024". In: *CyPress: 4th Workshop on Software Techniques for Engineering Cyber-Physical Systems*, CASCON'24, Toronto, Canada, 2024. IEEE CS, pp. 1–5, http://doi.org/10.1109/CASCON62161.2024.10838220 [85]

  - This is a summary of yearly improvements to the SYMBOLEO ecosystem, where I had minor contributions (covered implicitly in this thesis).

- **Sofana Alfuhaid**, Amal Ahmed Anda, Daniel Amyot, Marco Roveri, John Mylopoulos, "Towards an Architecture and Code Generator for End-to-End Access Control in Cyber-Physical Smart Contracts". In: *CyPress: 5th Workshop on Software Techniques for Engineering Cyber-Physical Systems*, CASCON'25, Toronto, Canada, 2025. IEEE CS, pp. 687–691, https://doi.org/10.1109/CASCON66301.2025.11422826 [9]

  - This paper introduces the SYMBOLEOAC architectural framework (Chapter 5) and API (Chapter 6).

- Gurdarshan Singh, Sahil Rajpal, Amal Ahmed Anda, **Sofana Alfuhaid**, Daniel Amyot, Marco Roveri, John Mylopoulos. "Towards an LLM-Based Auto-Corrector Agent for SYMBOLEO Specifications". In: *CyPress: 5th Workshop on Software Techniques for Engineering Cyber-Physical Systems*, CASCON'25, Toronto, Canada, 2025. IEEE CS, pp. 672–676, https://doi.org/10.1109/CASCON66301.2025.00124 [113]

  - This paper explores the use of Large Language Models for converting English legal contracts to SYMBOLEO specifications, with automated error correction; I provided minor contributions (not discussed in this thesis).

1.7 Limitations and Delimitations

This thesis demonstrates the feasibility and effectiveness of the proposed SYMBOLEOAC environment for specifying, generating, and executing access-controlled cyber-physical smart contracts. Through the design of the SYMBOLEOAC ontology, the design of a dedicated SYMBOLEOAC architecture, the corresponding extension of the SYMBOLEO language with access control and IoT related constructs, and the implementation of supporting tools and runtime components, this work provides an end-to-end solution that integrates formal contract specifications with blockchain-based execution and off-chain cyber-physical systems. The conducted case studies illustrate that contract, access control, and event-driven monitoring can be specified at a high level of abstraction and automatically translated into deployable smart contracts, thereby reducing development effort and improving correctness and traceability.

However, as with any research, this thesis is subject to certain limitations and delimitations that define its scope. These boundaries reflect the design choices made to ensure a focused and feasible evaluation. More detailed discussions of limitations and future extensions are respectively provided in Chapters 11 and 12.

1.7.1 General Limitations and Assumptions

In this thesis, the proposed SymboleoAC framework is evaluated using the Hyperledger Fabric platform, which aligns with the requirements of the considered application domains, including multiple stakeholders operating within a controlled environment. While the underlying ideas of SymboleoAC are not tied to a specific blockchain platform, public blockchain environments such as Ethereum (whose core blockchain is permissionless [25]) are outside the scope of the current evaluation.

The thesis primarily targets business-oriented contracts, such as for supply chains and energy trading, where contractual obligations, access control, and IoT-driven monitoring play a central role. These contracts can be of different natures, from business-to-business (e.g., large supply chains), to business-to-consumer (e.g., direct sales), or peer-to-peer (e.g., transactive energy). However, other types of legal agreements, including financial trading, personal contracts, or property contracts, are not considered in this work.

Finally, due to time and scope constraints, the evaluation emphasizes a selected set of non-functional requirements, with a particular focus on security and privacy. Other concerns, such as usability, robustness, and availability, are acknowledged as important directions for future work.

1.7.2 Security and Privacy: Scope and Assumptions

Security and privacy in CPSC involves multiple dimensions, including identity management, authentication, authorization, communication security, device trustworthiness, physical security, and human or social factors. In this thesis, the primary focus is on improving privacy and the security of interactions between on-chain and off-chain components of the SymboleoAC architecture (Figure 5.1) through end-to-end access control, authenticated communication, blockchain-based tempering protection, and permissioned execution mechanisms rather than providing a complete security solution for all possible types of attacks. From the cybersecurity functions identified in the NIST Cybersecurity Framework 2.0 [93], this thesis primarily focuses on the *Govern*, *Identify*, and *Protect* functions. More specifically, and using the NIST category identifiers (Table 1 in [93]), the proposed SymboleoAC framework focuses on:

- Role-based uthorization and policy enforcement through the SymboleoAC access control model (*Govern*: GV.RR, GV.PO);
- Identity management through certificate-based authentication (*Identify*: ID.AM);

- Authentication of users, IoT devices, CEP, and message broker (*Protect*: PR.AA);
- Permissioned blockchain execution using Hyperledger Fabric, assuming a sufficient number of participating blockchain nodes to reduce the risk of ledger tampering (*Protect*: PR.DS);
- Protection of communication channels through mutual Transport Layer Security (TLS) and encrypted messaging, assuming IoT devices and external components support the required authentication and encryption mechanisms (*Protect*: PR.PS);
- Accountability and traceability through blockchain records, event logging, and role-based interactions (*Govern*: GV.OS);
- Predefined contractual roles and permissions that are specified at design time within the SymboleoAC specification and access control policies (*Govern*: GV.RR).

The architecture also introduces mechanisms intended to reduce risks associated with unauthorized or compromised IoT devices, including certificate-based authentication, credential-based authentication, and controlled publish/subscribe permissions at the message broker level. In addition, SymboleoAC introduces controller rules for each resource derived from the semantics of legal concepts, as well as pre-authorization rules that grant different legal contract roles access to contractual resources and legal positions. These mechanisms provide an additional policy enforcement layer over contractual operations and event-driven interactions.

However, this thesis does not claim to provide complete security guarantees against all possible types attacks, and several aspects remain outside the scope of this work, including physical attacks on IoT devices, hardware tampering, social engineering, insider threats, denial-of-service attacks, and broader organizational security concerns. Instead, this thesis aims to improve the security and trustworthiness of cyber-physical smart contract interactions while providing a foundation for future extensions and stronger security mechanisms.

1.8 Thesis Outline

The rest of this thesis is organized as follows.

Chapter 2 provides an overview of the fundamental technologies and tools that are relevant to the thesis, including the Symboleo language and contributions made by others to its ecosystem, along with other technologies.

Chapter 3 provides a review of the theoretical part this thesis has built on. In the first part, we survey existing literature to explore platforms used for event-driven systems that integrate smart contracts and cyber-physical systems, examine event production and consumption methods, investigate the challenges developers face when creating cyber-physical smart contract systems (focusing on robustness, security, privacy, and availability concerns)

and discuss approaches to mitigate them. In the second part, we delve into Role-Based Access Control (RBAC) by exploring its foundational concepts and mechanisms. Additionally, we review significant works on RBAC to understand existing implementations within this model. We also compare our access control model with studies in both similar and different domains.

Chapter 4 presents the DSR-based methodology followed for conducting the research, detailing the approach, techniques, and tools used to address the research questions.

Chapter 5 presents an architecture for deploying SymboleO-based smart contracts as cyber-physical systems. It provides a detailed description of the architectural layers, including the one-chain layer, off-chain layer (with components such as IoT devices, message brokers, and a Complex Event Processing engine) and smart contracts generated from SymboleoAC specifications.

Chapter 6 reports on the development and implementation of the SymboleoAC API, including its integration with the message broker, CEP engine, IoT devices, smart contract listeners, and related components.

Chapter 7 presents the SymboleoAC ontology as an access control extension of SymboleO's with new modeling concepts inspired from Role-Based Access Control, tailored for the legal contract domain.

Chapter 8 provides the Xtext-based syntax and semantics of the SymboleoAC language, together with new well-formedness rules and a new version of the Eclipse-based IDE supporting the language.

Chapter 9 reports on the development and implementation of the SymboleoAC2SC tool, including its SymboleoACJS library.

Chapter 10 demonstrates and evaluates the application of the implemented tools on variations of two realistic legal contracts, including complex cases with multiple instances of multiple contracts with shared parties.

Chapter 11 provides a critical analysis of the results presented throughout the thesis. It discusses the contributions related to SymboleoAC and outlines related limitations and threats to validity.

Chapter 12 concludes the work presented in this thesis, provides answers to the research questions, and outlines directions for future research.

Note: a snapshot of the code-related artifacts, which reflects their status at the time the final version of this thesis was completed (May 2026), is available online, on Zenodo [5].

Chapter 2

Background

This chapter provides an overview of the technological foundations relevant to this thesis, including Cyber-Physical Systems, Smart Contracts and Blockchains, Complex Event Processing, and message brokers. It also serves as an introduction to the key concepts and tools of SYMBOLEO, which we will build upon in this thesis, and which are crucial for understanding the context of our work.

2.1 Cyber-Physical Systems

The foundational idea behind the concept of CPS is to keep an eye on and exert control over the physical world by incorporating cyber world capabilities (e.g., computation and communication) into aggregations of hardware [98], often composed of IoT devices. Thus, CPSs can be considered as an integration of physical elements, computing systems, and networks within a larger system where they can be controlled and monitored intelligently. Figure 2.1 shows a typical CPS architecture in its most abstract terms.

CPS applications include energy systems, smart systems, automotive systems, aerospace systems, robotic systems, industrial systems, IoT applications, and many more. Each of these applications is expected to adapt to changes that come from the outside world and react differently based on the requirements in a safe, secure, efficient, and (ideally) real-time manner [98].

Typically in CPSs, the cyber and physical worlds are exposed to each other through the use of APIs where the physical devices contain sensors that report actions and states of the environment being monitored [46, 111].

2.2 Smart Contracts and Blockchains

The term *smart contract* was first proposed by Szabo in 1997 [117]. It represents the contractual terms of a legal agreement in the real world, but in a completely digital way.

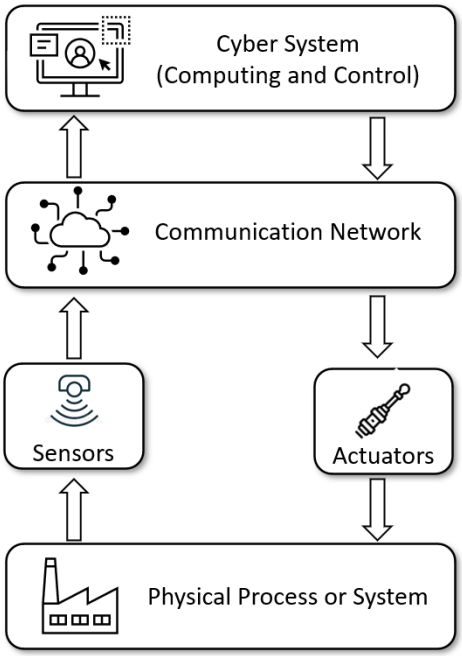


Figure 2.1: Bird's eye view of a typical cyber-physical system architecture.

These terms are translated and embedded in smart contracts in the form of code that i) dictates what we can and cannot do, and ii) is executed automatically based on the terms of the contract. The general goal here is that smart contracts will self-enforce these terms and minimize the need for (trusted) third parties between transnational or trans-organizational parties, while obtaining better monitoring and verification where terms must be satisfied [117].

Smart contracts often represent terms or conditions of a legal agreement using functions and events [49], possibly with rule-based patterns to recognize those events [47]. For example, if a contract's terms specify that *if the shipment exceeds the expected arrival date, then fees must be triggered against the shipping company*, then the arrival of the shipment at a given date must be a recognizable event.

As observed by Niya et al. [87], blockchain is the Distributed Ledger Technology (DLT) most commonly used in recent years. Blockchain is a peer-to-peer technology that enables storing and monitoring data in a distributed and decentralized manner [28]. The venue of blockchain platforms has revived the concept of smart contract (Figure 2.2) while providing decentralized execution and additional benefits such as immutability and transparency.

Smart contract implementations have capabilities for storing, sending, and receiving data [76]. Implementations can rely on a trusted centralized model, on a decentralized model, or on some hybrid approach [106, 115]. Buterin [25] created a leading DLT platform, called Ethereum[1], that features smart contract capabilities allowing the creation of distributed applications in many areas. Bitcoin[2], which is the first and likely the most

[1] https://ethereum.org/
[2] https://bitcoin.org/

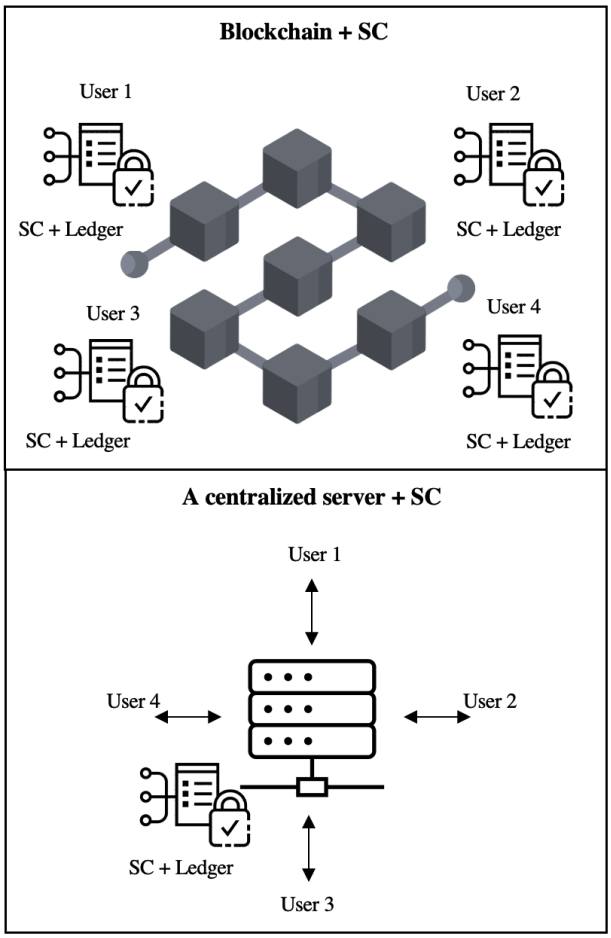


Figure 2.2: Centralized smart contract vs. decentralized smart contract (blockchain).

well-known blockchain platform, supports smart contracts that can process simple transactions. In contrast, Ethereum and other blockchain platforms such as Hyperledger Fabric[3] can process complex transactions and store records of any data.

There are different types of smart contracts that exist on such decentralized platforms, and they are often developed using different languages:

1. Bitcoin-style smart contracts: Use simple instructions as the Bitcoin platform features limited support for conditions, basic arithmetic, logical operations, and cryptography operations (e.g., for verifying digital signatures) on the blockchain [17].

2. General-style smart contracts: Use advanced scripts, written in common high-level languages, which are hosted on virtual machines (e.g., deployed using Docker[4]) in order to support the execution of smart contracts on the blockchain. For example, a smart contract for the Hyperledger platform can be written in Java, JavaScript (Node.js), or Go [13].

[3] https://www.hyperledger.org/use/fabric
[4] https://www.docker.com/

3. Domain-specific smart contracts: Use programming languages that exploit domain-specific knowledge to support contract-related concepts. For example, Ethereum supports Solidity, which features a Turing-complete scripting language for a variety of smart contract applications [123], as well as many other domain-specific programming languages (e.g., LLL, Serpent, and Vyper). Furthermore, many approaches [39, 45, 109, 127] have introduced new specification languages for modeling smart contracts.

There are also three types of blockchain-based ledgers: private, public, and consortium. They differ on the ability of parties to read and write from a ledger, the ability of nodes to join the blockchain network, the ability of nodes to validate and publish a block, and the type of consensus mechanism. For example, only assigned nodes can join the network and validate transactions in a private ledger. However, in a public ledger, anyone can enter the network and publish a new block. Consortium ledgers are in between private and public ledgers [28, 45, 123]. Both Bitcoin and Ethereum are examples of public blockchains, and their respective ledger is available to anyone. Hyperledger Fabric is an example of a private blockchain where the ledger is kept concealed and access is restricted [123].

2.3 Complex Event Processing and Message Brokers

A *Complex Event Processing (CEP)* system is a state-of-the-art technology designed to manage and analyze streams of real-time data to identify patterns, trends, and relationships based on a defined set of rules or criteria. Its primary objective is to respond to specific conditions or sequences of events as they occur [78].

For instance, if we want to monitor temperature readings from IoT sensors and send event data based on three consecutive readings, the CEP system would continuously analyze the incoming data. The event sources would be the temperature sensors, and the event processing engine would check each reading against the defined criteria. If three consecutive readings exceed or fall below a specified threshold, the event output mechanism would then trigger the sending of a resulting, more abstract event, which could be utilized by smart contracts to perform actions.

The CEP engine is responsible to automatically generate a complex event once the specified conditions/criteria within an event pattern are met. A CEP engine is a software tool that enables programmers to create and implement event patterns using an Event Pattern Language (EPL) [23]. EPLs can adopt various styles, each tailored to specific event processing needs. These styles range from stream-oriented EPLs to rule-oriented EPLs and imperative EPLs [23]. Given that many contractual events would be sourced from IoT devices, the most suitable type of Event Processing Language to use in this thesis would be a stream-oriented EPL. Common stream-oriented EPLs include Apache Flink SQL[5] and Esper EPL[6]. In this thesis, for the implementation of our CPSC architecture, we use the popular open-source CEP platform Esper (Event Stream Processing Engine).

[5] https://nightlies.apache.org/flink/flink-docs-master/docs/libs/cep/
[6] https://www.espertech.com/

Typical CEP products are often used in combination with *message broker* applications, such as RabbitMQ[7], Apache Kafka[8], and Pulsar[9], which provide complementary services including message queuing, transformation, and redirection (e.g., through a publish/subscribe feature). This thesis uses RabbitMQ, a popular open-source message broker that can cooperate with Esper.

In a CPS, a CEP platform will typically interact with a message broker as follows. IoT sensors send data to queues of the message broker. A CEP application connects as a consumer to these queues, and analyzes the event stream using predefined EPL rules (e.g., "*generate an alert if three temperature readings exceed $-4°C$ within 10 minutes*"). The CEP sends alerts or publishes new events back to another message broker queue for other services to consume. In this thesis, such services will include smart contracts, with an emphasis on security. In particular, this thesis will generate code for fine-grained and dynamic access control for the above components (IoT devices, CEP, message broker, and smart contract) from specifications of legal contracts.

2.4 SYMBOLEO

SYMBOLEO is a formal specification language for smart contracts developed at the Contract Specification and Monitoring (CSM) lab of the University of Ottawa, Canada. As described by Parvizimosaed et al. [92] and Sharifi et al. [109], SYMBOLEO aims to be an expressive contract-oriented language that enables property verification and the generation of monitorable and executable smart contracts. SYMBOLEO's syntax and grammar are based on an underlying ontology of contractual concepts.

2.4.1 SYMBOLEO's Contract Ontology

Figure 2.3 shows a representation of SYMBOLEO's ontology, which consists of concepts and their relationship for reasoning about contracts. The SYMBOLEO ontology includes the concepts of *obligations* and *powers*, influenced by an established ontology of legal concepts called the Unified Foundational Ontology-Legal (UFO-L) [41].

The main concepts of the SYMBOLEO ontology are as follows:

- **Contract**: the class Contract consists of obligations and powers that are concerned with assets (e.g., meat) between two or more roles (e.g., buyer and seller).
- **Asset**: represents any item that has value (tangible or intangible) that can be exchanged between parties during the execution of a contract.

[7] https://www.rabbitmq.com/
[8] https://kafka.apache.org/
[9] https://pulsar.apache.org/

- **Legal Position**: this abstract class represents the legal relationship between two roles during contract execution. The ontology shows two types of legal positions: obligation and power.
- **Obligation**: represents a legal duty from one party (the debtor) to another (the creditor), forming a legal situation of antecedent and consequent. For example, a seller (the debtor) is obligated to fulfill an obligation (e.g., meat delivery) to a buyer (the creditor) who has the right to the corresponding benefits. There are several types of obligations: conditional obligation (instantiated through specific trigger situations), unconditional obligation (instantiated at contract instantiation time), and surviving obligation (conditional or unconditional, but that can outlast the end of the contact itself, as in non-compete clauses).
- **Power**: a legal entitlement giving a party the right to create, change, suspend, or cancel legal positions. A power is instantiated through specific trigger situations and has antecedents that must be satisfied for it to take effect.
- **Legal Situation**: a type of situation linked with a particular instance of obligation or power. A situation occurs during a time interval T and holds during any sub-interval of T.
- **Event**: a fixed event that happens at a time point and cannot be altered. Events are characterized by pre-state and post-state conditions.
- **Party**: a legal entity, whether a person or an institution, possessing assets and designated roles within contractual agreements.
- **Role**: represents a debtor or a creditor in legal positions. During the execution of a contract, roles are attached to parties (e.g., party *Sofana* plays the role of a *seller*).

Both obligations and powers need three players in order to be enacted: a *performer* who acts to fulfill the obligation or exercises the power; a responsible party who is legally *liable* for the consequences of the obligation or power; and a *rightholder* who benefits from the fulfillment of the obligation. When an obligation is initiated, the debtor is its performer and is also liable for it, while the creditor is its rightholder. For powers, the creditor is performer and liable, while its debtor is rightholder. These players may change while an obligation or power is being enacted or delegated (e.g., to a transportation company) [91].

2.4.2 SYMBOLEO Specification Language

SYMBOLEO is defined with an Xtext [20] grammar, and its semantics are defined with axioms (based on event calculus) and state machines that allow monitoring the performance of real legal contracts through events [92].

A SYMBOLEO specification is structured into three main parts:

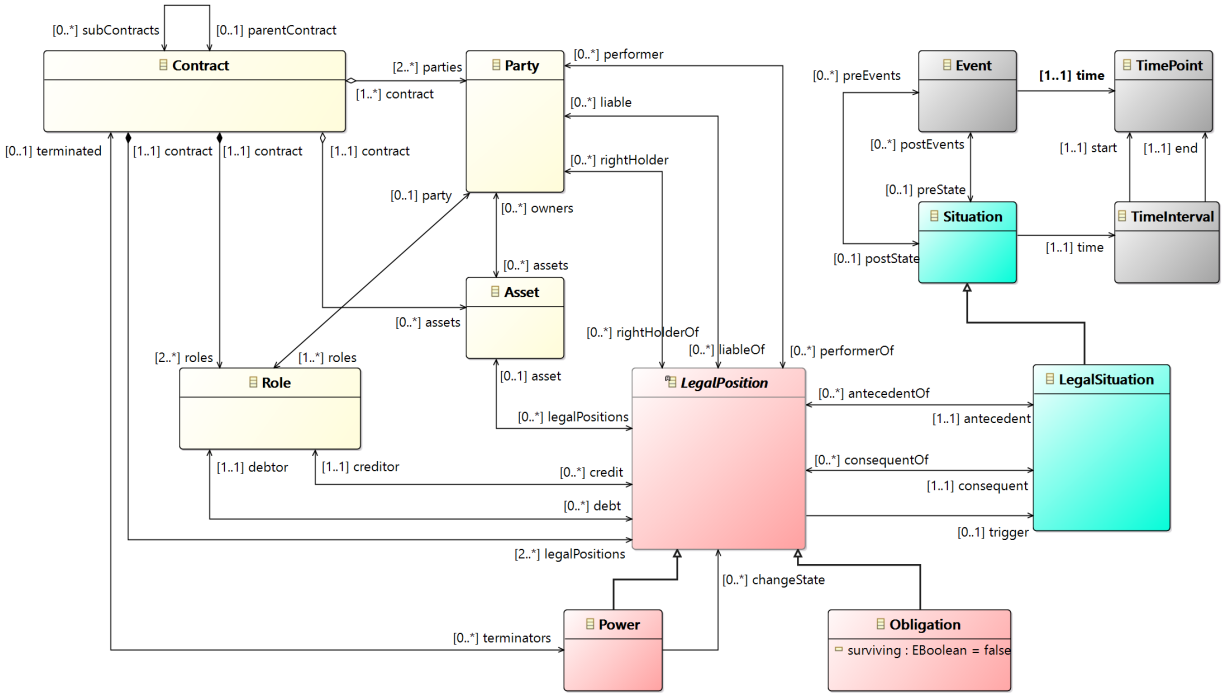


Figure 2.3: SYMBOLEO ontology [92].

1. **Domain**: represents SYMBOLEO Domain-specific classes that extend (isA) concepts from the basic SYMBOLEO ontology (e.g., assets, roles, and events) or other domain-specific classes.

2. **Contract Signature:** comes after the domain section. It begins with a Contract keyword and is followed by the contract name and its typed parameters. The signature enables one to instantiate a contract template with different values for its parameters.

3. **Body**: contains Declarations of variables (with initial values), as well as contract terms and conditions in the forms of (1) Preconditions and Postconditions; (2) Obligations ; (3) Surviving Obligations; (4) Powers; and (5) Constraints.

Listing 2.1 shows an example of a meat sale contract written in SYMBOLEO's specification language. The natural language text of this contract is provided in Table 2.1

```
Domain meatSaleDomain // the Domain section of the contract
  // participant models are defined using the Role type
  Seller isA Role with returnAddress: String, name: String;
  Buyer isA Role with warehouse: String;
  Currency isAn Enumeration(CAD, USD, EUR);
  MeatQuality isAn Enumeration(PRIME, AAA, AA, A);
  // the good to be delivered is defined as an Asset
  PerishableGood isAn Asset with quantity: Number, quality: MeatQuality;
  Meat isA PerishableGood;
  // the delivered event should be triggered when the goods are delivered
  Delivered isAn Event with item: Meat, deliveryAddress: String, delDueDate: Date;
  // the paid event should be triggered when the amounts due are paid
  Paid isAn Event with amount: Number, currency: Currency, from: Buyer, to: Seller, payDueDate: Date;
  // the paidLate event should be triggered when the penalty is paid
  PaidLate isAn Event with amount: Number, currency: Currency, from: Buyer, to: Seller;
endDomain
Contract MeatSale (buyer: Buyer, // the contract body starts here
  seller: Seller,
```

```
  qnt: Number,
  qlt: MeatQuality,
  amt: Number,
  curr: Currency,
  payDueDate: Date,
  delAdd: String,
  effDate: Date,
  delDueDateDays: Number,
  interestRate: Number) // parameters of the contract are passed here
Declarations
  // variables of the contract are initiated in this section
  goods: Meat with quantity := qnt, quality := qlt;
  delivered: Delivered with item := goods,
    deliveryAddress := delAdd,
    delDueDate := Date.add(effDate, delDueDateDays, days);
  paidLate: PaidLate with amount := (1 + interestRate / 100) * amt,
    currency := curr,
    from := buyer,
    to := seller;
  paid: Paid with amount := amt,
    currency := curr,
    from := buyer,
    to := seller,
    payDueDate := payDueDate;
Preconditions // safety conditions of the contract are specified below
  IsOwner(goods, seller);
Postconditions // safety conditions of the contract
  IsOwner(goods, buyer) and not(IsOwner(goods, seller));
Obligations
  delivery: // this obligation requires seller to deliver before the due date
    Obligation(seller, buyer, true, WhappensBefore(delivered, delivered.delDueDate));
  // the payment obligation requires buyer to pay before the specified due date
  payment: Obligation(buyer, seller, true, WhappensBefore(paid, paid.payDueDate));
  latePayment: Happens(Violated(obligations.payment)) ->
    Obligation(buyer, seller, true, Happens(paidLate));
Powers
  // if payment is violated then seller can suspend the delivery obligation
  suspendDelivery: Happens(Violated(obligations.payment)) ->
    Power(seller, buyer, true, Suspended(obligations.delivery));
  // if penalty is paid then buyer can resume the delivery obligation
  resumeDelivery: HappensWithin(paidLate, Suspension(obligations.delivery)) ->
    Power(buyer, seller, true, Resumed(obligations.delivery));
  // if delivery is violated then buyer can terminate the contract
  terminateContract: Happens(Violated(obligations.delivery)) ->
    Power(buyer, seller, true, Terminated(self));
Constraints
  not(IsEqual(buyer, seller)); // the buyer and seller must be different
endContract
```

Listing 2.1: Meat Sale contract specified in SYMBOLEO.

Table 2.1: Natural language text of the Meat Sale contract [109, 110].

This agreement is entered into effect as of <effDate>, between <party1> as Seller with address <retAdd>, and <party2> as Buyer with address <delAdd>.

1. **Payment and Delivery**

1.1 Seller shall sell an amount of <qnt> meat with <qlt> quality ("goods") to the Buyer.

1.2 Title in the Goods shall not pass on to the Buyer until payment of the amount owed has been made in full.

1.3 The Seller shall deliver the Order in one delivery within <delDueDateDays> days to the Buyer at its warehouse.

1.4 The Buyer shall pay <amt> ("amount") in <curr> ("currency") to the Seller before <payDueDate>.

1.5 In the event of late payment of the amount owed, the Buyer shall pay a late fee equal to <interestRate> owed, and the Seller may suspend performance of all of its obligations under the agreement until payment of amounts owed has been received in full.

2. **Assignment**

2.1 The rights and obligations are not assignable by the Buyer.

3. **Termination**

3.1 Any delay in delivery of the goods will not entitle the Buyer to terminate the Contract unless such delay exceeds 10 Days.

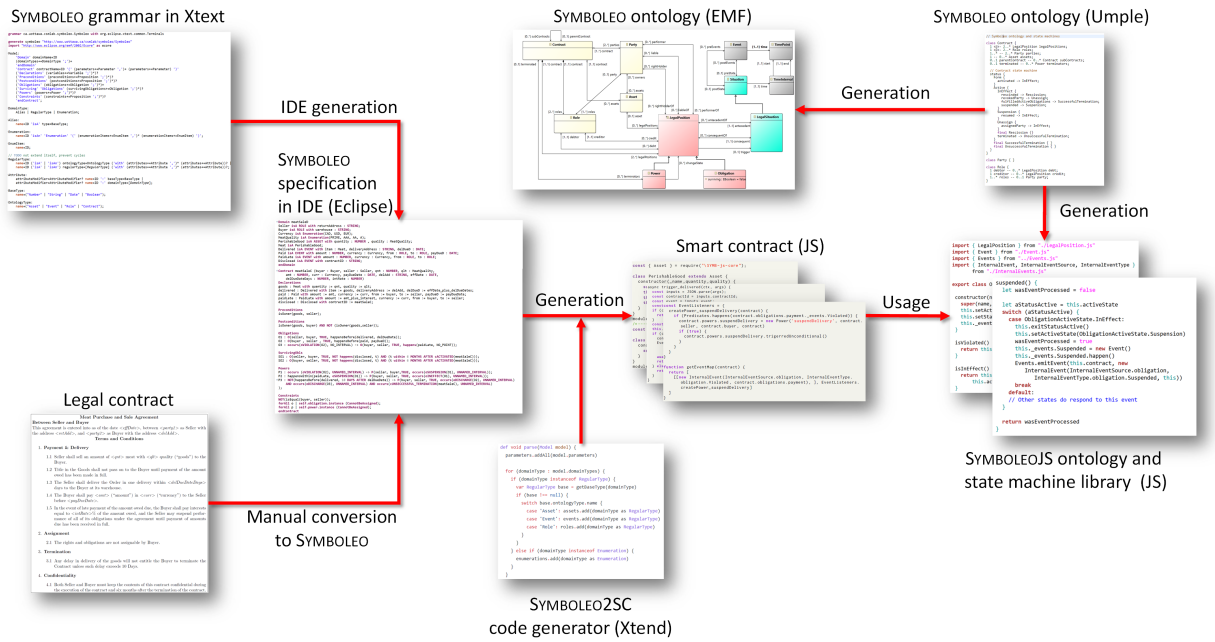


Figure 2.4: Overview of SYMBOLEO2SC and SYMBOLEOJS (from Rasti et al. [81]).

2.4.3 SYMBOLEOJS

The SYMBOLEO JavaScript library (SYMBOLEOJS) represents the implementation of SYMBOLEO's ontology and its semantics as a reusable JavaScript library. It was originally developed by Rasti et al. [101] to be utilized by the code generated through SYMBOLEO2SC (explained in Section 2.4.4).

The development of SYMBOLEOJS followed several key steps, highlighted in Figure 2.4. First, SYMBOLEO's ontology, along with its relations and state machines, was specified using the *Umple* language [38,74] (available online[10]). Since Umple supports code generation in multiple languages, Umple was used to produce Java code, which included classes and methods representing SYMBOLEO's concepts and relationships, as well as many utility methods (for instantiation, modification, and navigation) that maintain consistency in instantiated models. These Java classes, part of the ontology, are accessible online[11]. Finally, the generated Java classes were converted manually into their JavaScript equivalents (available online[12]), so that smart contracts (e.g., for the Hyperledger Fabric platform) can use them.

[10] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/blob/main/ontology/ontology.ump

[11] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/tree/main/ontology/java

[12] https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-JS-Core/tree/main/core

2.4.4 SymboleoSC

The Symboleo to Smart Contract tool (SymboleoSC), implemented by Rasti et al. [101], is used to generate executable smart contracts from Symboleo contract specifications. The code it generates invokes the methods in SymboleoJS, the reusable JavaScript library mentioned in Section 2.4.3. Since Hyperledger Fabric [13] supports JavaScript as one of its input languages, SymboleoSC simplifies the development process by generating ready-to-deploy smart contracts, saving developers time and effort. Recent studies have shown that the JavaScript code is usually 14 to 15 time larger than the corresponding Symboleo code, not even counting the 3,000 validated lines of code included in the SymboleoJS library [81, 101].

This chapter has introduced important background concepts and technologies relevant to this thesis. The next chapter will review the literature related to Cyber-Physical Smart Contracts and to Role-Based Access Control.

Chapter 3

Literature Review

This chapter presents the literature review conducted for this thesis. It is divided into two parts.

The first part, a systematic mapping review published in IEEE Access [6], examines the state-of-the-art of how Cyber-Physical Smart Contracts (CPSCs) are typically realized for compliance monitoring, together with associated challenges. It helps provide background and identify gaps mainly for the thesis's **RQ1** on the architecture framework (Section 1.4). This part is structured on the basis of four mapping review questions (detailed in Section 3.2) that focus on existing architectures, event-based interactions, infrastructure failures, and challenges. Section 3.2 covers the methodology used in this mapping review. Then, Section 3.3 presents the architectures that have been proposed for CPSCs and shows different patterns and components involved in interacting with their environment to receive and process events. As CPSC architectures may experience infrastructure failures and other challenges, Section 3.4 highlights such failures and mitigations thereof. Section 3.5 further describes technical challenges while Section 3.6 provides explicit answers to the mapping review questions that scoped this thesis and discusses threats to validity. Finally, Section 3.7 concludes the first part of the review.

In the second part, we review the Role-Based Access Control (RBAC) model and its fundamental concepts in Section 3.8. Then, we review significant works on RBAC to identify existing implementations within this model, and we also compare our proposed access control model with studies from both similar and different domains in Section 3.9. Finally, Section 3.10 concludes the second part of the review, which provides useful background especially for the thesis's **RQ3** on SYMBOLEO extensions related to privacy and security (Section 1.4).

3.1 Introduction

There is tremendous interest in developing smart contract applications in diverse markets, including banking, finance, insurance, government, agriculture, and supply chains[1]. Such

[1] https://tinyurl.com/b8sk3jnk

systems require special attention to their security and complexity as they often monitor legal transactions and compose components dynamically [111]. Some smart contracts, such as those monitoring Bitcoin transactions, operate fully in a cyber environment. Others, such as smart contracts that monitor meat sale transactions to ensure that delivery complies with perishable food transportation standards, operate in a cyber-physical environment. Our study focuses on this latter class of smart contracts, referred to herein as *Cyber-Physical Smart Contracts*.

CPSCs typically deploy Internet-of-Things technologies to monitor and control the execution of a process, often a business process or a legal contract execution. In addition, CPSCs often adopt Distributed Ledger Technologies (DLTs), including blockchain, to ensure integrity and immutability of their data in an environment that handles high-risk transactions without requiring trusted third parties. However, CPSCs may also involve trusted parties and centralized databases instead of, or in addition to, DLTs.

As CPSC applications multiply, it is important to understand how they are being built and used for supporting compliance monitoring and control. Through a mapping review, this chapter identifies and analyses relevant academic literature focused on event-based monitoring, including its architectures, platforms, interactions, infrastructure failures, and technical challenges.

3.2 Literature Review Methodology

A *mapping review* is designed to provide an overview of the literature relevant to research questions by exploiting academic research databases and complementary search approaches, selecting and analyzing the relevant articles, and synthesizing answers to the research questions. A mapping review also helps identify research gaps and is more oriented towards answering questions than a scoping review, the latter being usually more topic-based and used to scope and characterize the existence of the literature.

To provide explicit and reproducible systematic literature reviews (including mapping reviews), Okoli [88] defined a four-phase methodology that this mapping review follows:

1. *Planning*: Planning the review by identifying the review purpose and the mapping review questions.
2. *Selection*: Searching and screening the literature for relevant studies.
3. *Extraction*: Extracting from the papers the data that is relevant to the mapping review questions, and appraising their quality.
4. *Execution*: Answering the mapping review questions by synthesizing answers from the extracted data and reporting the results.

Figure 3.1 illustrates the phases involved in the performed mapping review. Also, as done in most literature reviews, a Preferred Reporting Items for Systematic reviews and Meta-Analyses (PRISMA) diagram [80, 82] is used to summarize the results of various inclusion and exclusion steps of the selection and extraction phases.

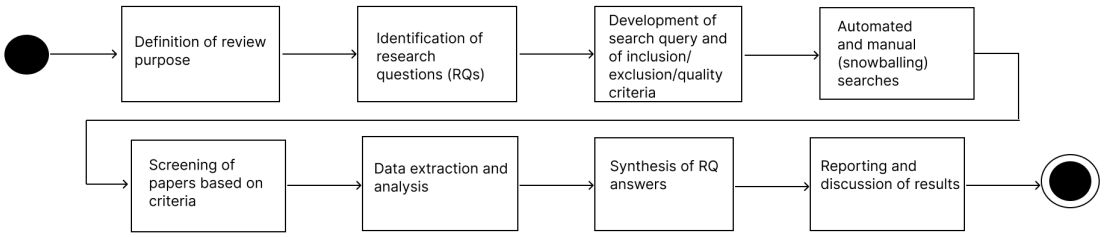


Figure 3.1: Overview of the mapping review methodology.

3.2.1 Planning Phase

This review aims to describe the state-of-the-art for CPSCs. Towards this purpose, the following four *mapping review questions* have been framed:

- **MRQ1**: What are the current platforms that support the implementation of CPSCs for event-based monitoring?
- **MRQ2**: How do CPSCs produce and consume events from/to the outside world for monitoring?
- **MRQ3**: What are current techniques for mitigating CPSC execution failures in event-based monitoring?
- **MRQ4**: What are the main technical challenges faced in the development of CPSCs for event-based monitoring?

3.2.2 Selection Phase

To answer the four review questions, an abstract search query that composes the four essential concepts related to this review (smart contract, CPS, monitoring, and event) and their synonyms/variants was designed to find relevant academic papers. The * truncation operator enables matching different variants of a keyword (e.g., for plural forms).

```
  "smart contract*"
AND
  (
    architectur* OR cyberphysical
    OR "cyber-physical" OR "cyber physical"
    OR CPS OR platform*
  )
AND
  monitor*
AND
  event*
```

This abstract query was tailored for several databases presented in Table 3.1. Scopus and Web of Science were included as they are broad-scope, curated databases that cover over 100 million records, and IEEE Xplore (which contains many CPS and smart contract papers) and Google Scholar (again, very broad in scope) were included for their full-text search capabilities.

The concept of "event" in the query was too restrictive when limited to title/abstract/keyword information for Scopus and Web of Science. Therefore, "events" was removed from their query, and a manual full-text search for "events" was conducted. However, as IEEE Xplore supports full-text search, "events" was kept in the full-text search field. For Google Scholar, given the severe limitations of its search engine, a simpler query containing the main keywords was used, and its results were only considered up to a predetermined depth based on Scholar's ranking of the papers' relevance.

Table 3.1: List of databases used in the mapping review.

Searched Database	**Fields**
Scopus	title/abstract/keywords
Web of Science	title/abstract/keywords
IEEE Xplore	title/abstract/keywords + full text: event
Google Scholar	full text, but simpler query

The review followed an automatic search from the four databases mentioned in Table 3.1. Additionally, several relevant papers have been found by exploring the referenced works through a complementary backward snowballing strategy (as suggested by Mourão et al. [84]).

The literature search was performed in April 2023. The retrieved papers were first screened using Covidence [29] based on the inclusion and exclusion criteria shown in Table 3.2. For that part of the screening, title, keywords, abstract, introduction, and conclusion of each papers were reviewed. Articles that met any of the exclusion criteria were excluded and the reasons were noted.

Although there are related patents that exist in that space, e.g., for general contract monitoring [44], smart contract compliance monitoring [31], or the monitoring of smart contracts themselves [103], patents were not included as they are not peer-reviewed according to scientific criteria.

3.2.3 Extraction Phase

After completing the first screening iteration mentioned earlier, a second iteration that combined screening and data extraction was conducted, this time using a full-text review. Some articles were excluded based on the quality assessment conducted by the first author. Only articles coming from a non-predatory source of information and meeting at least one of the following assessment criteria were included:

Table 3.2: Exclusion and inclusion criteria.

Exclusion Criteria	Inclusion Criteria
1. Studies that are not in English. 2. Studies in the form of a thesis, book, or survey. 3. Studies that are not peer-reviewed. 4. Studies that are duplicates of another study. 5. Studies whose focus does not answer any of the mapping review questions. 6. Patents	1. Peer-reviewed scholarly journals and conference papers about event monitoring systems that use smart contracts. 2. Studies that address at least one of the mapping review questions. 3. Relevant studies obtained through snowballing on previously selected studies.

1. Clarified the architecture(s) where the smart contract can be used for event-based monitoring.
2. Clarified how smart contracts interact, consume, and produce events.
3. Discussed infrastructure failures.
4. Discussed challenges faced by developers when embedding smart contracts in CPSs.
5. Evaluated the architecture(s).

Many studies were excluded because there was neither a clear description of a proposed architecture nor a discussion related to failures or challenges. Some of these papers are however cited in the previous sections and in the discussion as they still provided some useful information to support the content of the review. A PRISMA diagram [80] summarizing the selection results is presented in Figure 3.2.

The analysis was done using Microsoft Excel and is available on Zenodo[2]. A table was created with different columns used to extract relevant data, including the article information (title, authors, year); the location of the smart contract, its data, and its events in the architecture (on-chain, off-chain, hybrid); the platform used; the deployment technology; the smart contract languages used; the overall approach to event production and consumption; the types of infrastructure observed; and the types of technical challenges faced.

[2] https://zenodo.org/record/8000387

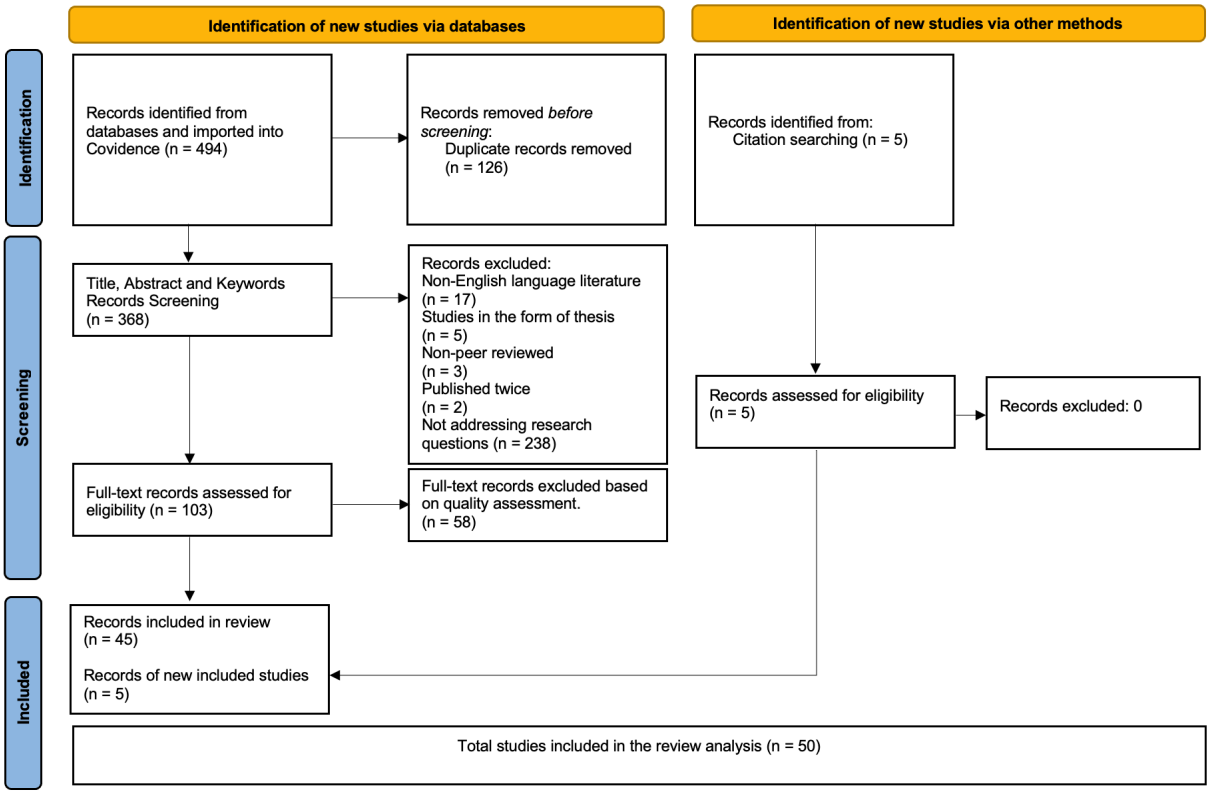


Figure 3.2: Summary of selection results, shown as a PRISMA diagram.

3.2.4 Execution Phase

Answers to the mapping review questions were synthesized by manually clustering the extracted data into different categories related to architecture, failures, and challenges, which are presented in the next sections.

The results overview, presented in Figure 3.3, shows that the research on CPSCs started in 2018 with its first publications, and that the highest number of selected publications was in 2022. Note that the queries were run in April 2023, so the results from 2023 are partial.

3.3 Architecture

In this section, the first and second mapping review questions are answered. The answer to MRQ1 on platforms is spread over Sections 3.3.1 and 3.3.3 while the answer to MRQ2 on events is provided in Section 3.3.2. The first question demands a detailed description of existing architectures for CPSCs, while the second question demands details on how data and events from the environment are produced and consumed while monitoring for compliance.

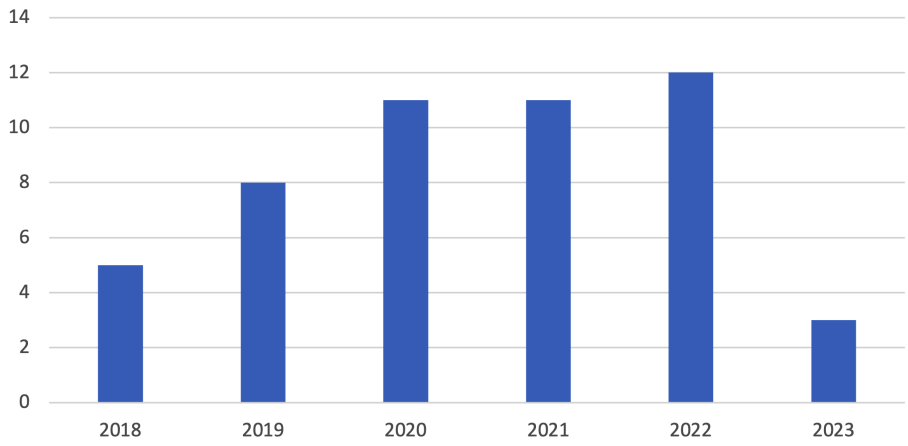


Figure 3.3: CPSC research distribution per year.

3.3.1 Overview

Typically, CPSCs consist of several physical (IoT) components that are responsible for collecting data from the outside world and controlling the environment. IoT devices are usually deployed as IoT components that include multiple devices and come in some sort of architecture. These components are networked and controlled by cyber components (e.g., by smart contracts) [111]. Figure 3.4 illustrates such physical components, cyber components, and the network. The physical components are usually sensors and actuators [46]. The sensors collect and transmit sensor data through a wired/wireless network to the cyber components. Such an architecture enables compliance monitoring and helps check the enforcement of predefined terms of the contract impacting the physical world [49].

Implementations of CPSCs that have been published in the literature are shown in Table 3.3. They are presented along different architectural characteristics, including smart contract location and data location (on-chain, off-chain, hybrid), platforms, deployment technologies, implementation languages for the smart contract, and the methods used for producing and consuming events. These implementations utilize blockchains as their underlying back-end infrastructure, with the system's operations and business logic encoded in smart contracts. When events or sensor data from the physical world are reported, the smart contract is invoked, and its predetermined terms and rules are executed automatically [46]. In recent years, relatively few studies have focused on the development of smart contracts for centralized systems, compared to decentralized ones [111], as using decentralized blockchain-based architectures offers desirable immutability and transparency features [46]. There are however undesirable side effects to immutable smart contracts when comes the time to update them, although some solutions have been recently surveyed for the Ethereum platform [97].

Blockchain is also employed as a back-end storage solution for storing sensor data and events or for keeping pointers to data stored in another layer of the architecture (off-chain). Storing information on a blockchain is however costly in terms of time, space, energy, and money. Accordingly, some studies have utilized off-chain storage options such as the

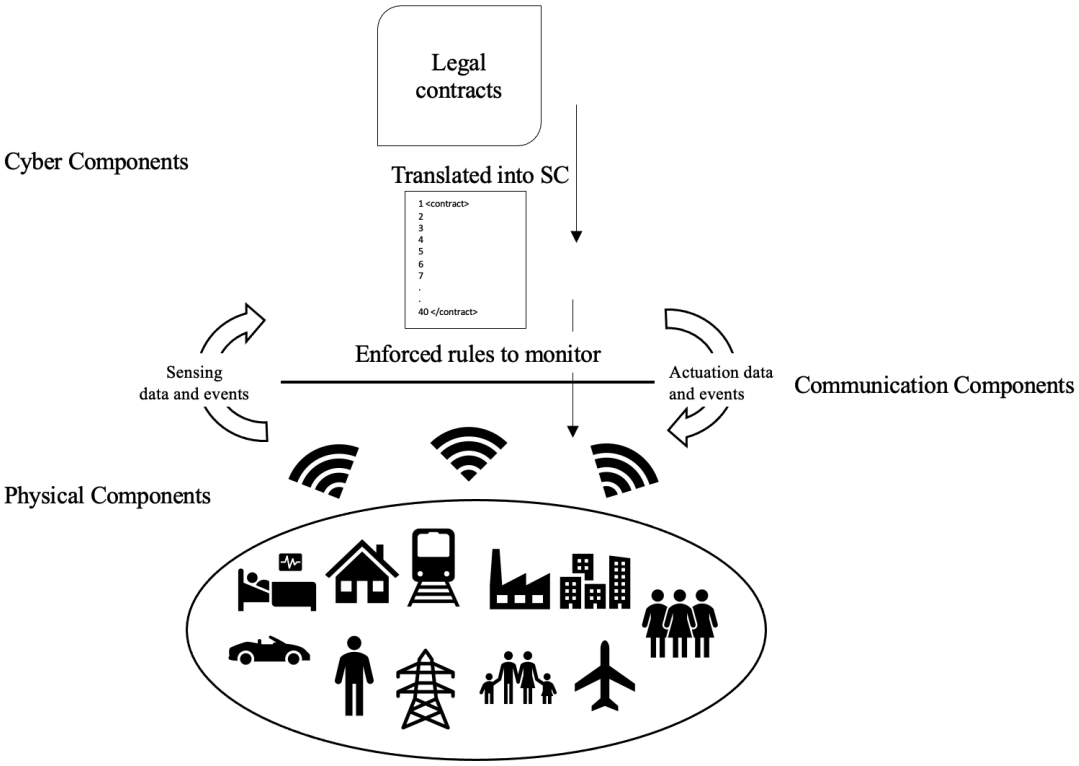


Figure 3.4: Overview of a typical CPSC architecture.

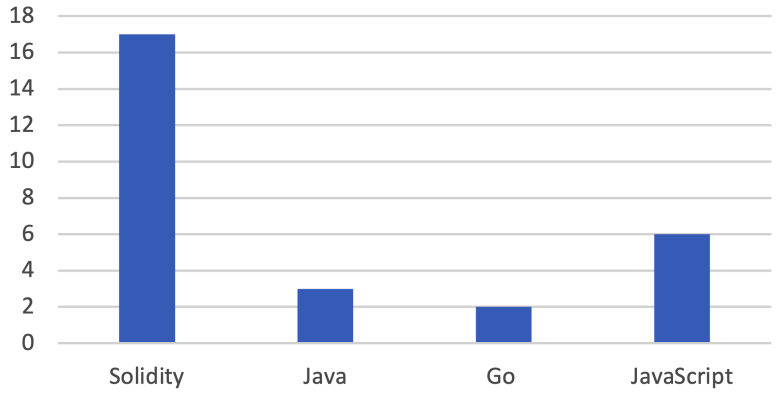


Figure 3.5: Languages used to implement smart contracts in the selected studies.

Interplanetary File System (IPFS) [15, 67, 77, 120], Swarm [77], or Couch DB [10, 47, 57]. These off-chain storage systems allow for more cost-effective and faster alternatives.

Smart contracts are created using either specialized languages like Solidity or general-purpose programming languages like Java, Go, or JavaScript. In the conducted review, Solidity was found to be the most frequently-cited language among the selected papers, as shown in Figure 3.5.

Additionally, smart contracts can be deployed on top of centralized (off-chain) or decentralized (on-chain) platforms [111]. However, some recent studies have demonstrated that

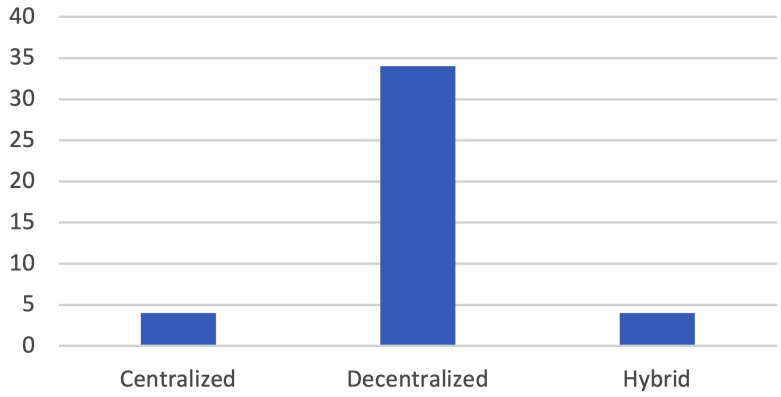


Figure 3.6: Platforms used to implement smart contracts in the selected studies.

smart contracts can also be implemented on both centralized trusted third-party and decentralized platforms (hybrid), where a smart contract running on a decentralized platform can trigger another smart contract on a centralized platform, depending on the execution process [106, 115]. Figure 3.6 illustrates the current trend of smart contract implementation platforms. Also, among the selected papers, Ethereum and Hyperledger Fabric are the leading blockchain platforms used for the implementation of smart contracts.

Furthermore, generated events play a crucial role in the implementation of smart contracts for monitoring compliance in the physical world. They are used by smart contracts to record or report violations. Event hubs or logs are available in the cyber components of the CPSC, which run on centralized or decentralized platforms and provide access to the events generated by smart contracts to the physical components of the system.

Several studies employ different protocols – such as Constrained Application Protocol (CoAP) and Message Queuing Telemetry Transport (MQTT) – to facilitate communication between physical and cyber worlds in regard to sensor data and events [48]. MQTT is the most frequently-cited protocol in the selected studies, which is used for storing, publishing, and aggregating sensor data and events and transmitting them to smart contracts. Section 3.3.2 provides a more detailed description of how smart contracts produce and consume sensor data and events from and to the physical world.

By examining the published literature on the implementation of CPSCs, we have arrived at a partial answer to the first mapping review question regarding existing architectures for CPSCs, as summarized in Table 3.3.

Table 3.3: Relevant literature related to CPSCs that includes a proof of concept.

Article	Year	Data Location	SC Location	Platform	Deployment	SC Languages	Events	Event Mngt. Approach
Niya et al. [87]	2018	hybrid	on-chain	Ethereum	-	Solidity	on-chain	-
Wright et al. [125]	2018	hybrid	on-chain	Ethereum	Truffle	Solidity	on-chain	publish/subscribe
Neidhardt et al. [86]	2018	off-chain	on-chain	Ethereum	-	Solidity	on-chain	-
Zhou et al. [130]	2018	off-chain	on-chain	Ethereum	-	Solidity	on-chain	-
Smirnov et al. [114]	2018	off-chain	on-chain	Hyperledger Fabric	Docker	GO/Java	on-chain	publish/subscribe
Hasan et al. [49]	2019	off-chain	on-chain	Ethereum	Remix IDE	Solidity	on-chain	message queues (MQTT)
Bagozi et al. [15]	2019	off-chain	on-chain	Ethereum	-	Solidity	on-chain	-
Liu et al. [75]	2019	off-chain	on-chain	Ethereum	-	Solidity	on-chain	message queues (MQTT)
Rúbio et al. [106]	2019	-	hybrid	Ethereum	-	-	-	-
Uriarte et al. [121]	2019	off-chain	hybrid	Ethereum	-	-	on-chain	-
Lopez-Pintado et al. [77]	2019	off-chain	on-chain	Ethereum	-	Solidity	on-chain	-
Hang & Kim [46]	2019	hybrid	on-chain	Hyperledger Fabric	Docker	JavaScript	on-chain	Message Broker
Breiki et al. [3]	2019	hybrid	on-chain	Ethereum	-	Solidity	-	message queues (MQTT)
Lockl et al. [76]	2020	hybrid	on-chain	Ethereum	Truffle	Solidity	on-chain	read/write
Tahmasebi et al. [120]	2020	off-chain	on-chain	Ethereum	Remix IDE	Solidity	off-chain	publish/subscribe
Solaiman et al. [115]	2020	off-chain	hybrid	Ethereum	Remix IDE	Solidity	on-chain	message queues (MQTT)

Ladleif et al. [71]	2020	off-chain	on-chain	-	-	-	on-chain	publish/subscribe
Dienbauer et al. [34]	2020	off-chain	on-chain	Hyperledger Fabric	Docker	JavaScript	on-chain	-
Taghavi et al. [119]	2020	off-chain	on-chain	Ethereum	Remix IDE	Solidity	on-chain	-
Lehnert et al. [72]	2020	off-chain	on-chain	Ethereum	Remix IDE	Solidity	-	-
Kochovski et al. [63]	2020	off-chain	on-chain	Ethereum	-	-	on-chain	-
Jamil et al. [55]	2020	off-chain	on-chain	Hyperledger Fabric	Docker		on-chain	-
Hang et al. [48]	2020	off-chain	on-chain	Hyperledger Fabric	Docker	JavaScript	on-chain	-
Hang & Kim [47]	2020	off-chain	on-chain	Hyperledger Fabric	Docker	JavaScript	on-chain	request/response
Baralla et al. [16]	2021	hybrid	on-chain	Ethereum	-	Solidity	on-chain	read/write
Cacciagrano et al. [27]	2021	hybrid	hybrid	Ethereum	-	-	on-chain	-
Alzubaidi et al. [10]	2021	on-chain	on-chain	Hyperledger Fabric	Docker	Java	on-chain	read/write
Shukla et al. [112]	2021	on-chain	on-chain	Ethereum	-	Solidity	on-chain	read/write
Jamil et al. [57]	2021	hybrid	on-chain	Hyperledger Fabric	Docker	JavaScript	on-chain	request/response
Jamil et al. [56]	2022	hybrid	on-chain	Hyperledger Fabric	Docker	JavaScript	on-chain	-
Pustišek et al. [96]	2022	hybrid	on-chain	Ethereum	-	Solidity	on-chain	publish/subscribe
Hewa et al. [51]	2022	hybrid	on-chain	Hyperledger Fabric	Docker	Java	on-chain	message queues (MQTT)
Zhang et al. [128]	2022	hybrid	on-chain	Hyperledger Fabric	Docker	Go	on-chain	-
Kumar et al. [67]	2023	off-chain	on-chain	Ethereum	-	-	on-chain	-

3.3.2 Patterns to Interact with the Physical World

Having gained insight into the programming languages and platforms involved in constructing CPSCs for compliance monitoring, it is time to address the second mapping review question by providing details on how smart contracts produce and consume events generated from the physical world. This section provides a general overview of the commonly used patterns and components in the literature for maintaining the connection between smart contracts and the physical world.

The ability of smart contracts to enforce contract terms depends on receiving events and data from the physical world. However, on-chain smart contracts cannot interact directly with the physical world; extra components (e.g., data carrier) must be used to maintain the connection between smart contracts and the physical world [75]. This is due to the fact that some DLTs are designed to run smart contracts in isolation to be disconnected from the outside world, offering secure and reliable sharing of contractual agreements of event-driven monitoring [71, 75]. For instance, Ethereum does not allow smart contracts to query data directly from the outside world, but Hyperledger Fabric does [34]. However, a hybrid blockchain has been suggested by Falazi et al. [37] as a way for smart contracts to directly access off-chain data. Regardless, data carriers may still be necessary for any blockchain-based smart contracts to ensure a deterministic behavior of smart contracts in monitoring CPSs [34].

Single-board computers like Raspberry Pi are also used to facilitate communication between the outside world and blockchain-based smart contracts. The Raspberry Pi boards gather sensor data from external sources (e.g., cloud, IoT devices) and execute the relevant function within the smart contract [48,76]. This causes the smart contract to initiate events to record the new data or contract violations.

Additionally, Representational State Transfer (REST) servers offer several REST APIs to connect the outside world to blockchain-based smart contracts [46]. These servers allow web applications or physical components to interact with smart contracts to access monitored data and events. Similarly, Ethereum provides several APIs (known as Web3 APIs) for calling back events monitored by smart contracts from the physical world [76].

Furthermore, an oracle acts as a data carrier or a mediator to establish a secure connection between blockchain-based smart contracts and external components such as APIs, IoT devices, cloud providers, and more [3,34,52,63,71,75,86,119,121,125]. Figure 3.7 illustrates the basic architecture of an oracle in conjunction with smart contracts and Table 3.4 provides a description of the function of each type besides other patterns that enable physical components and smart contracts to communicate with each other in interconnected ways.

Based on the reviewed literature, all of the above-mentioned technologies can be leveraged in centralized, decentralized, or hybrid models for CPSCs, except for the Web3 API, a built-in functionality provided by the Ethereum blockchain that can be utilized on either decentralized or hybrid models.

Table 3.4: Patterns used by smart contracts to interact with the physical world for consuming and producing sensor data and events.

Articles	Pattern	Purpose
[16,47,49,51,72,76, 87,125]	Raspberry Pi board	Catches off-chain sensor data or events and calling suitable function within the Smart Contract (SC).
[14, 34, 46, 46–48, 55,57,63,75,77,106, 111,112,115]	REST APIs	Enables web applications or physical devices to invoke smart contracts to access monitored data and events.
[67, 76, 115, 120, 125,130]	Web3 APIs	Calls back events monitored by SCs from the physical world.
[34, 63, 75, 86, 119, 121]	Off-chain oracle	Fetches off-chain sensor data and events from multiple data sources in the physical world, verify them and deliver them to the SC.
[75]	On-chain oracle	Listens to off-chain data requested by the SC.
[71]	History oracle	Provides history values besides the current or latest value
	Publish-subscribe oracle	Retrieves new values immediately from multiple off-chain data sources when changes happen as long as transactions have been subscribed.
[52]	Inbound oracle	Sends off-chain data to on-chain components.
	Outbound oracle	Allows smart contract to request data from on-chain components.

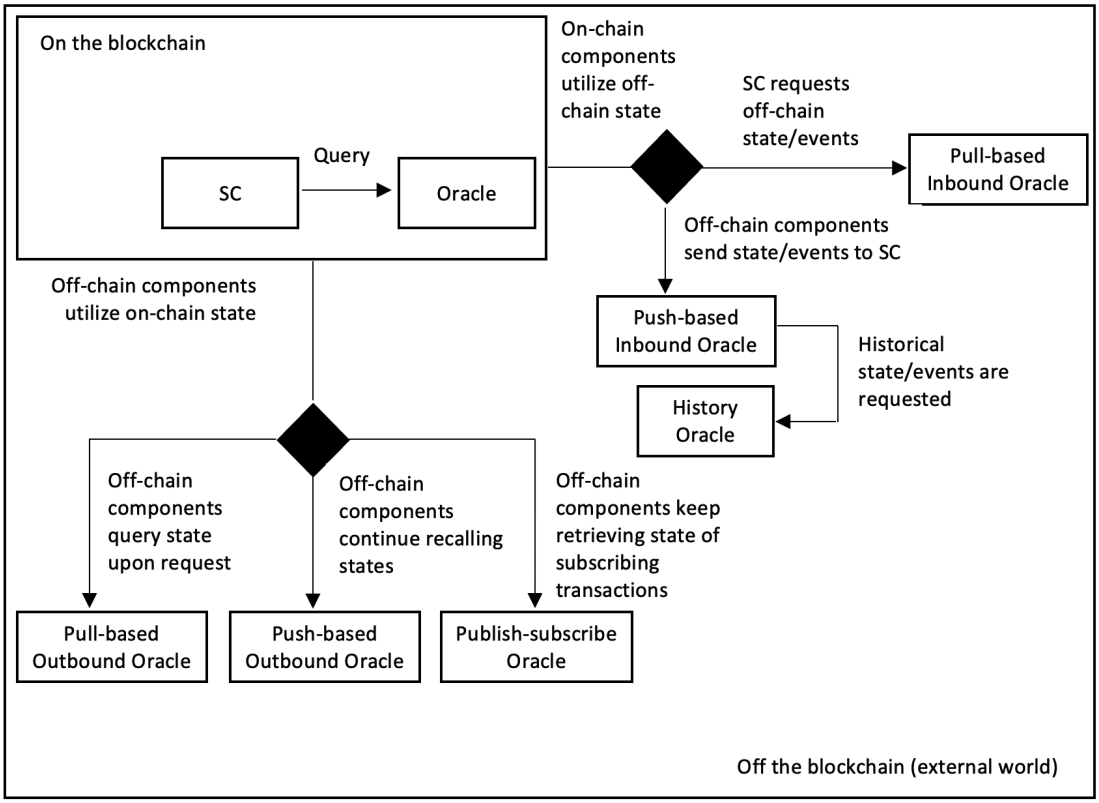


Figure 3.7: An illustration of basic oracle patterns to access data from/to blockchain and smart contracts in CPSCs.

3.3.3 Layered Architecture of CPSC

From the literature listed in Table 3.3 and according to Governatori et al. [40], there are three main approaches for applying smart contracts for compliance monitoring in CPSCs:

- Centralized (off-blockchain implementation)(off-chain): The smart contract is managed by a third-party server, the contract's terms and conditions are monitored/carried out off-chain, and contract-related data and events are stored off-chain.
- Decentralized (blockchain implementation)(on-chain): The smart contract is deployed on the blockchain, the contract's terms and conditions are monitored/carried out on-chain, and contract-related data and events are stored on-chain.
- Hybrid (off/on-chain implementation): The smart contract can be divided where a part of the smart contract is placed on-chain, while the other part remains off-chain. Some of the contract's terms and conditions are monitored/carried out off-chain, while the rest is monitored on-chain. Some contract-related data and events are stored off-chain, and the rest is stored on-chain.

Based on the reviewed literature, different architecture implementations were chosen by developers based on criteria such as privacy and scalability [115]. However, blockchain-

based architectures were the most cited ones among selected papers. Few studies investigate off-chain and hybrid architectures.

Several design concepts have been proposed for using smart contracts to monitor compliance by utilizing sensor data and events generated by IoT devices. Part of the literature suggests a monitoring architecture that includes a front-end monitoring application to observe sensor data and events produced by the physical world [76,87,120]. Typically, external applications initiate smart contracts to monitor and enforce contract terms, while smart contracts store sensor data and record related events. In their architecture, smart contracts are deployed, installed, and instantiated on every node on the blockchain network, which serves as a back-end storage and processing infrastructure.

Hasan et al. [49] proposed a blockchain-based smart contract for monitoring product shipments, defined by business rules and risk thresholds. The physical world's IoT-enabled containers and sensors are used to track the movement of the product and perform self-checks of various measurements, comparing them to predefined conditions. An MQTT server is employed to store, aggregate, and regularly publish sensor readings. If there is a violation, the container will initiate a call to the smart contract, triggering and registering an event in the events log. A similar approach was proposed by Lockl et al. [76]; however, the main difference is that a monitoring dashboard application is available to end-users, allowing them to monitor sensor data and related events and register components. Additionally, the blockchain is used as a light node, storing only the hash of blocks rather than the entire blocks. In another similar approach proposed by Hang and Kim [46], smart contracts are utilized to store sensor data and keep track of the configuration of physical components. The contracts trigger events when predefined monitoring conditions are met or violations occur. A client application can send a request via REST APIs to the smart contract to either register a new component for monitoring or access stored sensor data and events. CoAP is employed by the server to transmit sensor data from devices to smart contracts in real-time.

Furthermore, a four-layered architecture for CPSCs was proposed by Tahmasebi et al. [120], consisting of application, service management, gateway, and physical layers. The gateway layer collects and saves the sensor data from the real world in off-chain storage. Smart contracts are triggered by the aggregated data and executed accordingly. An implementation of two smart contracts on the blockchain monitors the execution and registration of IoT devices. The sensor data produced by these devices is stored in off-chain storage using IPFS. Bagozi et al. [15] also presented a CPSC in a four-layered architecture that includes the acquisition, gateway, blockchain, and application layers. However, they use smart contracts to provide anomaly detection services based on the sensor data and related events from monitored devices.

Zhou et al. [130], Lopez-Pintado et al. [77], and Kochovski et al. [63] propose decentralized architectures for CPSCs similar to the architecture of the above-mentioned literature, including physical, gateway, distributed (blockchain), and application layers. It is worth mentioning that some studies have combined some layers or used different terminology. For instance, Lopez-Pintado et al. [77] proposed a three-layered architecture of CPSCs for compliance monitoring consisting of storage, access, and process-aware layers. Zhou et

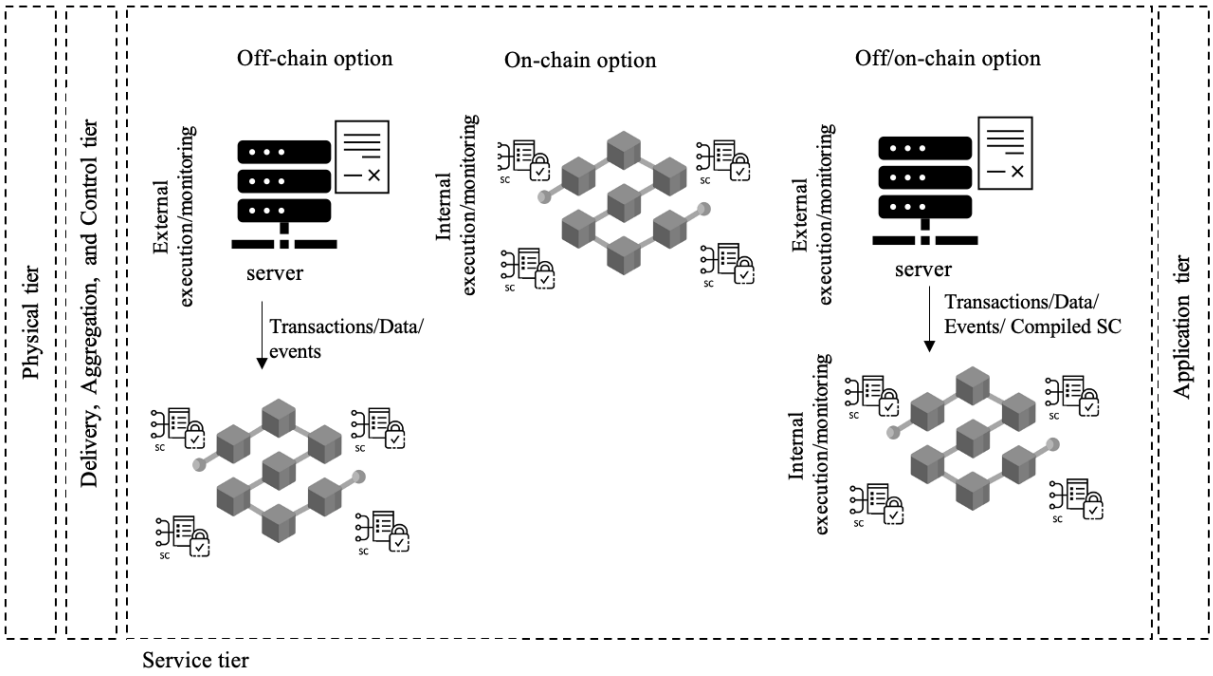


Figure 3.8: Conceptual tier architecture for CPSCs.

al. [130], Lopez-Pintado et al. [77], and Kochovski et al. [63] differ from the above literature by adding an extra layer for off-chain events monitoring. Zhou et al. [130] introduced a witness model to report violations and monitor off-chain events stored in the cloud. In contrast, Lopez-Pintado et al. [77] added an off-chain event monitoring model to retrieve and monitor on-chain events stored on an event log. On their side, Kochovski et al. [63] added a decision-making layer with a decision-making mechanism, a monitoring system, and an orchestration system for off-chain monitoring.

The literature highlights the importance of various components such as devices, event management, off-chain storage, communication interface, off-chain monitoring systems, and client applications that are connected using decentralized, centralized, or hybrid architectures. Figure 3.8 presents a typical tier architecture for compliance monitoring of CPSCs, obtained from the reviewed literature. CPSCs can have multiple independent layers that developers can modify. However, the minimum requirements for building CPSCs are displayed in the figure and described as follows:

1. **Physical tier**: This layer contains physical devices (e.g., sensors and actuators), data storage, communication protocols, and so on. Physical devices collect sensor data and events from the physical world and pass them to the next tier.

2. **Delivery, Aggregation, and Control tier**: This tier is composed of data carriers that validate and verify the data generated from the physical world before sending it to the service tier. It may also contain monitoring agents that process the data received from sensors for compliance monitoring and trigger the appropriate smart contract function in the service tier. Each of them has communication capabilities to communicate with the service tier.

3. **Service tier**: This is the tier where the smart contracts are placed to pull information from the physical world to apply the agreed policy from the agreed legal contract. Also, in this tier, events are generated based on the compliance monitoring rules specified by smart contracts. All events, sensor data, and device information are stored in this tier as well. Figure 3.8 shows three main options, where SC execution and monitoring can occur on-chain, off-chain, or both on-chain and off-chain.

4. **Application tier**: This tier provides many services, such as monitoring the execution of the physical world (e.g., a monitoring dashboard), allowing users to interact with the data, or performing analysis on them.

Additionally, based on the reviewed literature, there are several options for storing collected sensor data and events generated by the physical world, where smart contracts can access data/events through the control layer for compliance monitoring purposes. Table 3.5 lists various data storage options where sensor data and events are being consumed from and produced for the physical world.

Table 3.5: Data storage for producing and consuming events.

Article	Data Storage
[3, 46, 49]	CoAP servers
[3, 46, 71, 75, 114, 115, 125]	Message queues (MQTT) servers
[51, 70, 120]	Fog node
[3]	Cloud storage

3.4 Infrastructure Failures

After reviewing existing architectures for CPSCs, and the way events are generated and consumed in the environment that is being monitored, we focus here on MRQ3. Specifically, we study possible infrastructure failures and summarizes existing approaches to alleviating these failures. Figure 3.9 shows the risks that are identified from the reviewed literature and the corresponding mitigation approaches.

The first noticeable risk is that sensor data and related events that are generated by the physical world are not sent directly to the smart contracts; rather, they are stored and transmitted through a third party (i.e., data carrier) that resides on a centralized architecture. This approach is particularly vulnerable because it consists of a single point of failure [119]. Taghavi et al. [119] suggest utilizing multiple data carriers to feed the data to smart contracts from the outside world as a mitigation approach.

Another risk is that CPSCs rely on centralized services (e.g., cloud, fog node, MQTT, and CoAP servers) for storing physical device information and performing operations on them. However, a disadvantage here is, again, a single point of failure for such services [46]. The suggested mitigation approach is to combine centralized storage with distributed storage, such as blockchains [46]. This concern is exemplified by the Atlanta Ransomware

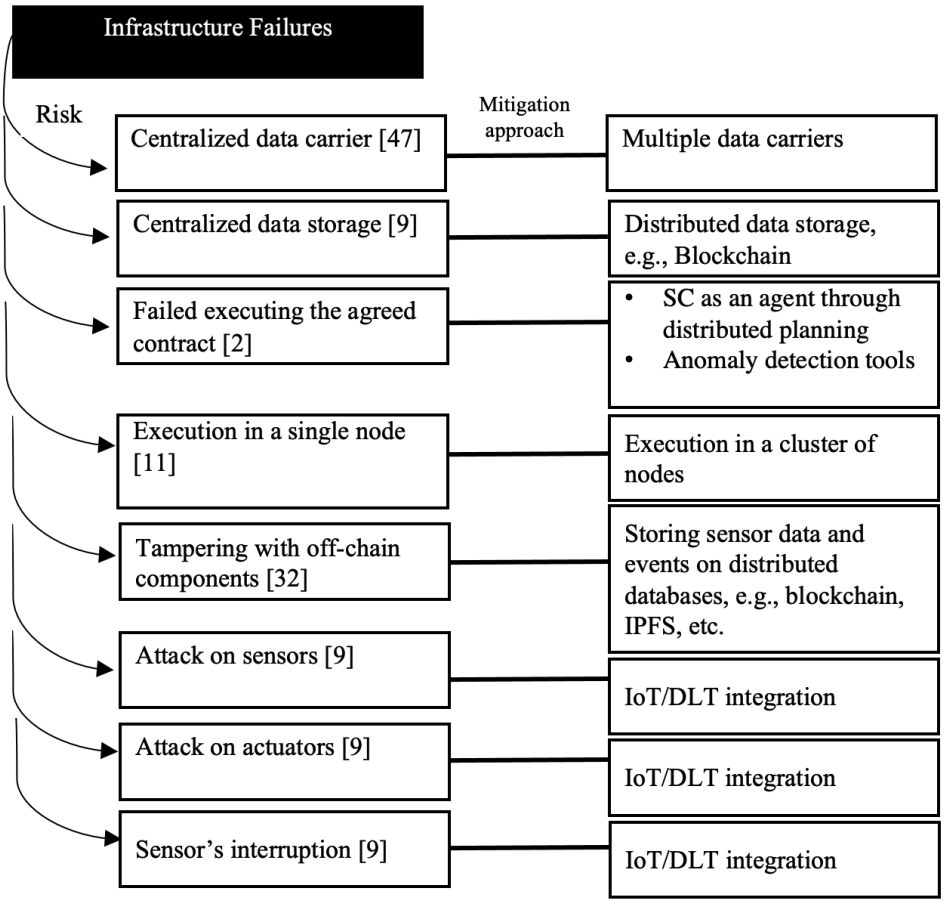


Figure 3.9: Common CPSC infrastructure failures with mitigation approaches.

Attack[3] in 2018, where a ransomware incident targeted the centralized infrastructure of the city, causing disruptions to critical services. This incident serves as a stark reminder of the risks associated with relying solely on centralized services in CPSCs. To enhance resilience and mitigate the impacts of such infrastructure failures, it becomes crucial to explore decentralized and distributed storage solutions, such as blockchains, to ensure the availability and integrity of CPSC operations [46].

Additionally, monitoring processes that involve multiple agents entails the risk of failures resulting from agents not following the agreed protocol of execution. Shukla et al. [111] propose CPSCs that monitor process execution and detect anomalous executions (centralized or decentralized). For instance, the Binance Smart Chain exploit in 2022 serves as a relevant example[4]. In this incident, an attacker exploited a vulnerability in the Binance Smart Chain's infrastructure, manipulating a smart contract's code to steal a significant amount of "FOO" tokens. Such incidents emphasize the importance of implementing robust security measures and conducting careful code reviews to prevent unauthorized access and manipulations of smart contracts.

[3] https://l8.nu/qh9M
[4] https://decrypt.co/111433/binance-smart-chain-hack

Moreover, running a CPSC on a single node can result in system failure if the node fails. Hang and Kim [47] utilize blockchain as back-end storage for compliance monitoring. However, their system runs on a single node, which makes it fault-intolerant. The authors do point out that the usage of a single (solo) node is suitable for running smart contracts for testing purposes only. However, for production purposes, they suggest using clustering nodes for implementation, such as those used by Kafka[5] or Raft[6] to avoid system failures.

Furthermore, components that reside off-chain are vulnerable to tampering. In this context, Lopez-Pintado et al. [77] suggest the use of distributed data storage, such as blockchain, to store sensor data and related events. This mitigation approach provides reliable and secure storage, so all compliance monitoring processes happen on-chain.

Typically in CPSs, IoT contains multiple sensors that are responsible for producing and exchanging massive quantities of data. Such sensors are themselves vulnerable to cyberattack and hence compromise the integrity of the data. IoT/DLT integration is suggested as a paradigm to handle such attacks [46].

Intercepting and altering the order of received data is another risk that can cause major losses. IoT/DLT integration can provide a level of authenticity of IoT devices, such as actuators, to ensure the integrity of transmitted data [46].

Finally, good network connectivity is crucial for IoT devices to perform properly and transmit sensor data to the smart contract; failure to do so can result in significant losses of sensor data and events [46].

3.5 Technical Challenges

3.5.1 Security

Smart contracts cannot directly access off-chain data accumulated by monitoring the outside world. Instead, access needs to be provided by a third-party data carrier [34, 52, 63, 71, 75, 86, 119, 121, 125]. This is because smart contracts have been designed to operate in a closed environment, disconnected from the outside world for security reasons [71, 75, 123]. Oracles have been used as a data carrier solution to fetch data from different data sources, verify them, and send them to the smart contract [3, 34, 71, 75, 119]. However, the trustworthiness of oracles and the integrity of provided data have become challenging. Therefore, the lack of trusted data carriers and the absence of a robust environment of reliable data sources impede the applicability of CPSCs.

Another concern is the lack of security measures for the communication between physical and cyber components used to link smart contracts with the outside world [69]. For example, the communication between REST APIs and physical components (e.g., IoT) [47, 48, 55]. Similarly, as more heterogeneous physical and cyber components and

[5] https://kafka.apache.org/
[6] https://raft.github.io/

services become connected to cyber-physical smart contract systems, more security risks, insecure connections, and bugs are expected to happen [30].

To deal with those smart contract vulnerabilities and take advantage of the CPSC paradigm, blockchain-based smart contracts have been proposed as a solution because they provide distributed security with their cryptographic mechanisms, especially when sensor data and events are collected and shared among different physical and cyber components [30]. Additionally, security analysis tools for contract vulnerabilities are constantly being developed to find potential security bugs and check compliance and potential violation of contract behavior.

Two relevant open-source audit tools are Securify[7] and Manticore[8]. *Securify* is a security audit tool for Ethereum smart contracts. It uses symbolic analysis on the dependency graph of the contract to obtain accurate semantic details from the code, and then it checks whether the smart contract behaves according to what is intended. *Manticore* is another symbolic execution tool capable of tracing smart contract transaction inputs to detect potential violations.

Furthermore, an Intrusion Detection System has been developed to handle other vulnerabilities and cyber and physical attacks as a result of the increasing incorporation of cyber and physical components where sensor data and events are collected and used to monitor different features of a CPSC [15, 67–70]. An Intrusion Detection System is a software tool that can be used to detect attacks automatically using machine learning, deep learning, or other techniques [67]. It is responsible for detecting attacks, triggering warnings, or taking action [67]. For example, Bagozi et al. [15] developed an anomaly detection service to monitor sensors and check if identified measures of sensor data are above or about to reach pre-identified thresholds during a transportation journey. Kumar et al. [67] propose a scalable blockchain-based smart contract for secure data transmission. Also, A deep learning architecture combining Deep Sparse AutoEncoder with Bidirectional Long Short-Term Memory is proposed for intrusion detection in a healthcare network. Kumar et al. [69] discuss IoT-based zero-touch networks for secure data sharing. They also suggest a deep learning approach for intrusion detection. Kumar et al. [70] suggest a two-level approach for data security. The first level uses blockchain and smart contracts to ensure secure data exchange, while the second level utilizes deep learning to encode data into a new format that prevents attacks. Kumar et al. [68] suggest a new framework for secure data sharing in Intelligent Agriculture that utilizes deep learning and smart contracts enabled by the Internet of Things to detect intrusions.

Note that the above technological solutions to security concerns are those that focus on cyber-physical smart contracts, as scoped by our mapping review questions and search query. Other solutions to security vulnerabilities targeting other types of systems can be found in the work of Leka et al. [73].

[7] https://github.com/eth-sri/securify2

[8] https:https://github.com/trailofbits/manticore

3.5.2 Availability

Availability in cyber-physical smart contracts relates to the capacity of smart contracts to maintain their state and be accessible to entities and physical and cyber components, regardless of the circumstances, e.g., power outages, network outages, attacks, or resource limitations [76]. For example, using Wireless Sensor Networks as a network infrastructure constitutes a risk because of limitations with respect to connectivity, data stream, energy, storage, and capacity [120]. Also, the current CPSC paradigm relies on transmitting sensor data and events to off-chain centralized servers that represent single points of failures before delivering them to smart contracts for compliance monitoring [76].

Maintaining CPSC availability is challenging, especially in scenarios where it is critical to ensure data is always available. For instance, such scenarios include monitoring sensor data and events in healthcare systems or food supply chains, which can be addressed by ensuring continuous functioning of both cyber and physical components even if some of them are damaged, and by implementing solutions to handle potential single points of failure [76].

To tackle the challenges related to availability, the CPSC paradigm can integrate various recommended methods, such as redundancy and load-balancing capabilities. An IoT/DLT architecture has often been used as a back-end infrastructure as it maintains redundancy [76]. Blockchain-based architectures allow cyber and physical components to be distributed over multiple nodes handling the same tasks; there is no single point of failure. Thus, when one node fails, the other nodes can take over [48]. Another approach is to use load-balancing, which allows deploying physical and cyber components over different resources to avoid overloads [63].

3.5.3 Robustness

Smart contract robustness is crucial for maintaining contact with the physical world via actuators and sensors. Currently, there is a significant rise in the adaptation of physical components such as IoT devices due to their capabilities to provide real-time data and enable networking among various CPS applications [33]. Also, the amount of generated real-time data by IoT sensors is increasing, which requires massive storage, processing techniques, and networks allowing them to interact with other physical and cyber components and exchange data for compliance monitoring conducted by smart contracts [33]. Thus, since CPSCs involve growing cyber and physical components and data, it is hard to maintain robustness and expect how the systems can react to different conditions, e.g., power constraints, network outages, IoT-limited resources, and so on. Also, such systems adopt a centralized architecture that is exposed to data loss at any given time, as such architecture is not fault-tolerant [33].

To address these challenges, many studies, such as [33, 76], report on architectures that achieve fault-tolerance through redundancy, which ensures that if one component stops responding for any reason, other components can carry out. Another robust mechanism is the use of artificial intelligence and expert systems to make decisions about an CPSC's

behavior and take actions accordingly, e.g., redistribute resources to avoid overwhelming IoT devices of the monitored environment [33].

3.5.4 Privacy

As indicated earlier, many CPSCs studied in the literature adopt an IoT/DLT architecture. However, maintaining sensory data on-chain is expensive. Therefore, several studies have turned to alternate hybrid approaches for storing sensor data and physical device information. Such practices are prone to privacy violations, particularly in terms of data privacy for smart contracts and the privacy of data carriers that contain sensitive data concerning the monitored process [16, 76].

Currently, public blockchains represent a real challenge as they do not support access control, resulting in unrestricted access to contract-related data and sensor data, as access to sensitive data can not be prevented. Thus, a variety of solutions have been suggested, such as (1) zero-knowledge proofs [51,64,67], (2) homomorphic encryption [128], and (3) secure multi-party computation [128]. Kosba et al. [64] proposed a model called Hawk that allows developers to write a privacy-preserving smart contract using zero-knowledge proofs that can keep its privacy to prove the validity of transactions to contract parties without exposing its content. Also, Kumar et al. [67] suggested a privacy-preserving scheme to protect and prevent data leakage during the transmission of data from/to smart contracts. This privacy-preserving scheme involves verifying IoT devices using zero-knowledge proofs. Hewa et al. [51] utilized zero-knowledge proof to maintain anonymity of identities stored in blockchains. Other privacy-preserving primitives suggested by Zhang et al. [128] include homomorphic encryption and secure multi-party computation. Homomorphic encryption keeps data confidentiality by conducting operations on encrypted data [125], whereas secure multi-party computation is another type of encryption that allows parties to come together to conduct computation on off-chain functions without disclosing their data [115].

For data carriers, they do not all have the same level of privacy and data access because sensor data and events are collected and shared among different off-chain data sources in CPSs. Therefore, the privacy and integrity goals of data carriers are challenging. Town Crier [129] is an approach that provides controlled access to off-chain data provided by different data carriers.

It is worth noting that these privacy-preserving approaches are resource intensive in a cyber-physical smart contract environment full of IoT devices with limited capacity.

3.5.5 Legal and Regulatory

As stated earlier in Section 2, a smart contract is a program created using a programming language, such as Go, JavaScript, or Solidity, that can independently enforce, verify and control the execution of a legal contract. However, a smart contract may not always be considered a completely enforceable legal contract, depending on how well it satisfies the requirements of laws and legal standards [83]. Thus, there could be an inconsistency between the legal contract and its corresponding digital representation, which may affect the

codification of laws [83]. Recent research initiatives related to this matter [45, 109, 127] aim to enable the development and verification of legal contracts and their corresponding smart contracts through the use of domain-specific languages with various levels of formality.

There could also be legal issues with the data manipulated by smart contracts. For example, in European countries and in the US, people's medical data are protected under privacy protection regulations and governance [55], which allows individuals to ask for their personal information to be removed; this is the so-called "right to be forgotten". Such provision conflicts directly with the immutability feature of smart contracts that leverage blockchain technology.

In addition, the execution of smart contracts is a dynamic process that cannot occur in isolation, as it is often influenced by various factors and external forces [40]. These factors involve rules and regulations (the law in a particular jurisdiction) from other contracts and external events from IoT devices that may affect the contract's outcome.

Thus, legal and regulatory challenges include, but are not limited to:

- Determining which legal rules and regulations would apply to transactions being executed in CPSC applications;
- Deciding on a strategy for the modification and deletion of data from the blockchain (e.g., through access removal or blockchain forking), as required by applicable legislation;
- Determining who takes responsibility for the consequences (disputes, claims, and financial penalties) when the contract's outcome does not align with the legal requirements that must be met.

Collaboration among legal experts, engineers, and stakeholders is essential to tackle the legal and regulatory aspects of smart contracts. Such collaborative effort can help ensure the integration of legal requirements into technical design and implementation, as well as help guarantee code correctness, compliance, and regulatory alignment during the deployment and execution of smart contracts. Approaches such as formal verification and others surveyed by Wang et al. [123] can help smart contracts become better aligned with existing jurisdictions.

3.6 Discussion

This section explicitly answers the four mapping review questions identified in this mapping review, together with relevant threats to the validity of our work.

3.6.1 Answers to Mapping Review Questions

The review highlights a growing interest in using CPSCs to oversee contract execution and ensure contract compliance. Not all reviewed approaches deployed smart contracts to

monitor legal contracts; instead, many have utilized smart contracts to monitor and control sensor data and events and react to violations. Here are the findings of each mapping review question and some insights about the conducted mapping review:

For **MRQ1** (*What are the current platforms that support the implementation of CPSCs for event-based monitoring?*): the findings of the literature review, which are reported in Section 3.3, demonstrate that the development, deployment, and invocation of CPSCs involve a variety of platforms, development tools, and data carriers. Table 3.3 presents a summary of the CPSC literature with architectural characteristics themes. The centralized, decentralized, and hybrid platforms were proposed as different CPSC approaches for compliance monitoring. Few studies proposed and explored the hybrid and centralized architecture for CPSCs, with most studies using a decentralized implementation for distributed execution of smart contracts. The decentralized approach is favored because it offers the benefits of blockchain technology, such as transparency and immutability.

Also, the results of the review indicate that smart contract execution and monitoring can occur on the blockchain, while also some parts of the smart contracts may be executed off-chain, and some monitoring procedures may remain off-chain as well. The prevalent method described in the literature for enforcing compliance is to conduct monitoring outside of smart contracts, where smart contracts can utilize feedback from external sources to respond to monitoring outcomes. In reality, monitoring using blockchain-based smart contracts can be challenging due to the high cost of execution and monitoring on the blockchain. Therefore, several approaches proposed for storing sensor data/events and conducting monitoring off-chain to reduce costs.

The literature also revealed that smart contracts are typically created manually and customized to fit the targeted programming language and platform, making their creation an exceedingly challenging task for developers in various CPS fields. Finally, the proposed layered architecture of CPSC, shown in Figure 3.8, is based on a layered architecture that may consist of several distinct components, and it is the responsibility of the developers to remove, add, or modify them according to the requirements of the targeted field in CPSs.

For **MRQ2** (*How do CPSCs produce and consume events from/to the outside world for monitoring?*): the results from Section 3.3 show that smart contracts cannot access and monitor sensor data and events from the outside world directly. External components and technologies are needed to verify, consume, and produce these data and events for compliance monitoring. Tables 3.4 and 3.5, together with Figures 3.7 and 3.8, summarize the needed components, technologies, and communication patterns for consuming and producing events. An oracle is used as a third-party agent to check the veracity of sensor data and events that cannot be accessed by the smart contract or cannot be reached by the physical world. Also, additional storage could be used as a conduit for consuming and producing sensor data and events, e.g., a cloud environment could store sensor data and events collected from an oracle. Sensor data and events are either made available to anyone to ensure transparency or are only accessible to certain parties in order to preserve privacy. Three methods are used to store sensor data and events, including storing them directly in the blockchain, in third-party storage (off-chain), or dividing them between the blockchain and multiple third-party storage entities.

For **MRQ3** (*What are current techniques for mitigating CPSC execution failures in event-based monitoring?*): the results reported in Section 3.4 show that the components and technologies needed for supporting CPSCs could come with multiple types of infrastructure failures. Figure 3.9 summarizes failure types and corresponding mitigation approaches. Also, the existing literature focuses more on run-time smart contract execution failures than on infrastructure failures and solutions (e.g., limitations of IoT devices). This could be a significant barrier hindering the development of smart contracts in CPSs. Therefore, infrastructure failures require further research in order to better address challenges related to integrating smart contracts with CPSs in practice.

For **MRQ4** (*What are the main technical challenges faced in the development of CPSCs for event-based monitoring?*): the results presented in Section 3.5 reveal that the role of CPSCs in compliance monitoring brings multiple technical challenges. Multiple aspects need to be considered while making architectural decisions related to CPSCs. Important technical challenges relate to security, availability, robustness, privacy, and legal and regulatory aspects.

3.6.2 Threats to Validity

As for any literature review, this mapping review is prone to several limitations and threats to validity, which are listed in Table 3.6 together with related mitigation strategies.

Table 3.6: Mapping review limitations and related mitigation approaches.

Limitations/Threats	**Mitigation Strategies**
Papers written in languages other than English, which may contain useful information on the topic, were not included.	None.
Having mainly one person (the first author) involved in the review, selection, and filtering of papers might bias the validity of the mapping review content.	Having multiple mapping review authors agree on the relevance of borderline papers helped reduce the risk of bias.
The selected databases and the query used may have left out relevant papers.	Snowballing was used as a complementary paper selection strategy.
Deciding the relevance of a paper based on the abstract only may result in the exclusion of relevant papers.	The irrelevant papers have been excluded after reading title, keywords, abstract, and conclusion.
The results of the mapping review are limited to information collected in academic publications (peer-reviewed journals and conference papers), which limits external validity and might not reflect industrial reality.	None.

In addition, as this is a literature review, there was no experiment-based or empirical assessment of the various approaches discussed in the literature review.

3.7 Conclusion – First Part of the Literature Review

This first chapter part presented the findings of a mapping review on the use of CPSCs for compliance monitoring, with the aim to provide insights on existing architectures, patterns, infrastructure failure types, and technical challenges. To our best knowledge, this mapping review is the first literature review focused on smart contract development for event-based monitoring in cyber-physical systems. Since the publication of this mapping review in 2023, another literature review was published in 2025 on blockchain-enabled SCs and the IoT [26]; the latter covers more papers but does not go deep on architectural concerns for CPSs. Another recent review focused on one architectural aspect (privacy) for a given domain (transactive energy domain) [116].

The main conclusions of this mapping review are summarized below, together with corresponding suggestions for potential elements of solutions and future research opportunities:

1. Existing research on smart contract compliance monitoring in CPSs is limited to the blockchain as a dominant execution environment with back-end storage and a mediator for transferring sensor data and events because this architecture maintains secure, immutable, and redundant storage. In the future, research should be extended to cover other approaches to overcome the limitation of blockchains in terms of cost, speed, and power usage. For example, off-chain execution could be used as an alternative approach to blockchain for executing smart contracts [40, 83]. This strategy requires shifting data and/or smart contracts "off" the blockchain and to a different platform that performs better in terms of execution speed, storage, regulatory compliance, and so on. This approach can decrease blockchain costs and provide better scalability by improving transaction processing times and storage [40]. Off-chain execution, however, may jeopardize some of the benefits of security and transparency that come with using blockchain [40]. Another option to explore could involve Directed Acyclic Graph based Distributed Ledgers, which offer better scalability than blockchain at the expense of more limited security due to the absence of consensus algorithms [72].

2. Regarding the lack of trusted data carriers, external data and events from the physical world are necessary for smart contracts execution and monitoring. The connection between the cyber and physical worlds is made possible by reliable data carriers (e.g., oracles) to verify the data generated by physical components and deliver them to smart contracts. However, the trustworthiness and security of data carriers are seen as major impediments to the use of smart contracts within CPSs. Therefore, a robust ecosystem of reliable data carriers is necessary to improve the implementation of CPSCs. Many potential solutions could be utilized to establish a robust ecosystem to ensure the accuracy of data and events that are roaming around several physical and cyber components in various CPSC applications, including but not limited to:

    - *Data Provenance*: Tracking the source of data, including its origin, usage, and history can help minimize data tampering. Blockchain provides an immutable

data record where data origins can be traceable to ensure data accuracy and consistency [16].

- *Data Protection*: Taking necessary privacy and security measures to protect the shared ecosystem's data and events is also something to be considered. This involves applying measures such as preventing unauthorized access or other privacy and security practices described in Section 3.5 of this paper.
- *Data Standardization*: The use of standardized data formats would also facilitate the sharing of data and events while supporting interoperability amongst the smart contract and other systems [40].

3. Few studies have focused on monitoring infrastructure failures; instead, most studies have focused on failures related to the execution of smart contracts. There is a need to extend the research on infrastructure failures of CPSs due to the increased adoption of IoT, cloud, and other technologies, which is crucial for the accurate and reliable monitoring and execution of smart contracts. As mentioned previously in Section 3.4, hardware failures can occur with actuators/sensors for a variety of reasons (e.g., attacks, damages, or degradation [46]), potentially resulting in major data loss. Potential solutions aiming to reduce the impact of hardware failures could involve the use of IoT/DLT integration that supports redundancy, e.g., distributing multiple actuators/sensors across various nodes such that if one fails, another can take over the task. Such solutions can help prevent single points of failure [46].

4. For compliance monitoring, existing studies have often used an extra layer in the monitoring architecture between smart contracts and the external world, where usually the monitoring happens outside the smart contracts. Also, based on the monitoring decision, a suitable function within the SC will be called. There is however a need for a reference monitoring model of generated data and events to identify and deal with them efficiently.

5. The body of smart contract functions is often coded manually using languages such as Solidity, JavaScript, Go, and Java. There is no method for directly converting a real contract into an enforceable smart contract, which further increases the complexity of adopting smart contracts in other CPS areas. There has been some research on this matter, such as the use of domain-specific languages, template-based code generators, and verification. Template-based code generators provide templates to assist developers in creating smart contracts efficiently. These templates consist of predefined common smart contract constructs code, as proposed by Hamdaqa et al. [45]. Domain-specific languages can also represent a potential solution [100, 109, 127]. By using domain-specific languages, developers can use domain concepts, rules, and high-level coding abstractions related to contracts, which can simplify the process of creating smart contracts, hence reducing coding errors, development time, and overall complexity. Such languages can hence help address some of the challenges discussed in our review. Verification is another potential solution to address the complexity of coding smart contracts [123]. This usually involves the use of (formal) verification

methods to ensure that a smart contract complies with its intended specification (including legal obligations) and can prevent vulnerabilities. These solutions have their own challenges in terms of additional time and domain expertise needed to exploit them properly [45, 127].

6. Common CPSC challenges related to availability, robustness, privacy, and legal and regulatory aspects have not been investigated extensively yet, which again provides many opportunities for further research.

3.8 Role-Based Access Control (RBAC)

The previous mapping review highlighted several challenges related to legal smart contracts as CPSs. This thesis focuses on a subset related to *privacy/security* concerns, including data protection, as well as *compliance monitoring* concerns (conclusions 2 and 4 in the previous section). This section describes important concepts and mechanisms pertaining to privacy and security through access control.

Access Control (AC) [32, 108] is a security mechanism designed to regulate and manage access to resources (i.e., objects) within a software system, ensuring that only authorized parties or entities can interact with those protected resources. The essence of access control concepts centers around giving specific **Permissions** to **Subjects** (i.e., any system entities), enabling them to execute **Actions** (such as read, write, etc.) on **Resources** (i.e., class, file, etc.). A subject refers to entities that request to access a resource or data within a resource to accomplish a task. This access request may be blocked if the subject does not have an access right to the resource or any part of it. A resource refers to objects where access control policies are defined to manage who and what actions can be performed. A permission is the access right specifying an action that is permissible to be carried out on a resource. This access control framework ensures that only authorized entities can perform permitted actions, safeguarding the system's integrity and security. By delineating access rights, the flow of information and activities within systems can be managed and regulated, minimizing the risk of unauthorized access or misuse of resources and fulfilling access control policies.

Permissions are maintained by security engineers/administrators or anyone in similar positions. They often use the concepts of **Rule** and **Policy** in access control frameworks to assign permissions instead of assigning permissions directly to individual subjects, facilitating more efficient management and scalability [32].

An access control policy is a collection of rules that serves as a blueprint for the system's security requirements, defining who can access what and under which conditions [22]. A rule is a mechanism to assign permissions to a subject [61]. A rule consists of conditions on one side (subject, action, and resource) to which the policy applies, along with an effect on the other side that can be either 'GRANT' or 'REVOKE' [61]. Thus, a rule can have the following form: *Rule R = {Effect, Action, Subject, Resource}*. Policy *specification* is the static level where each subject will be granted specific access rights that govern the security requirements of the system. On the other hand, policy *enforcement* is the

dynamic level that involves the mechanism of implementing defined rules that the system must follow [32].

Access control policies can be categorized into different models such as role-based access control (RBAC) [22, 36, 107], attribute-based access control (ABAC) [36, 53], discretionary access control (DAC) [22, 60], mandatory access control (MAC) [22, 36, 60], and many more. In this thesis, we adapt the RBAC model where access rights are assigned to roles instead of individual subjects. Also, RBAC allows for clear, static role assignments that can be validated through certificates in permissioned blockchains such as Hyperledger Fabric [13]. Furthermore, it offers distinct benefits compared to alternative models such as hierarchical roles, where subordinate roles inherit privileges and permissions from higher-level roles [107].

3.9 A Review of Important Work on Access Control

Security concerns, including RBAC, have been a significant focus in conceptual modeling. In this context, we review various suggestions for implementing Role-based Access Control model. We compare our work to a similar study in another domain [2, 95], as well as with general works [18, 19, 59, 62, 66] and those within the same domain [39, 127]. Table 3.7 presents a comparison of our approach and other approaches regarding their access control over protected resources and support for code generation.

3.9.1 Similar Studies in Another Domain

Planas et al. [95] integrate access control in a bot definition language. Their approach involves integrating RBAC protocols into the Conversational User Interface (CUI) of chatbot resources. Similar to our work, their approach focuses on restricting access to resources such as events and states. However, it neither provides a fine-grained control that includes attributes of events, nor covers a broad range of resources, such as assets, which they do not explicitly support restricting access to, such as products in their case and their attributes. Al-Azzoni et al. [1, 2] propose an access contact model that extends the metamodel of iContractML [45], with similar access control concepts to ours. However, that model only restricts access to assets and does not provide fine-grained control over those assets or other resources such as events.

3.9.2 General Studies

Other Domain-Specific Languages (DSLs) focusing on access control modeling, such as [18, 19, 59, 62, 66], employ RBAC primitives similar to ours. Basin et al. [18] introduce a Model-Driven Security framework that allows system models and security requirements to be specified, with access control infrastructures. By merging Unified Modeling Language (UML) system modeling with security modeling, the approach formalizes access control needs. The system is implemented in a UML-based tool and is designed for server-based applications.

Ben Fadhe et al. [19] introduces GemRBAC-DSL, a new specification language aimed at closing the gap between expressive RBAC models and policy specification languages. With a natural language-like syntax for ease of use, GemRBAC-DSL includes semantic checks to resolve policy conflicts. It leverages Object Constraint Language (OCL) formalizations to support model-driven policy enforcement. Kashmar et al. [59] propose a generic AC metamodel that unifies various AC concepts and plans to support IoT in the future. Their metamodel addresses the challenges found in traditional AC models by incorporating features from multiple AC frameworks, providing a higher level of abstraction that meets security and privacy standards. Kim et al. [62] present a method to embed RBAC policies into UML design models by using reusable patterns expressed through UML template diagrams. These patterns are instantiated into specific application models to incorporate RBAC policies. Additionally, the method provides tools for identifying policy violations, illustrated with a banking system example. Kuhlmann et al. [66] propose a UML-based DSL for developing and validating RBAC policies, particularly focusing on simplifying the handling of authorization constraints. The DSL abstracts complex mathematical structures and includes features for handling dynamic constraints. Validation is supported through a UML and OCL tool for better policy analysis and enforcement.

The above works include approaches focusing on restricting access to states or assets only. However, these approaches remain at the modeling stage and do not extend to code generation for implementing access control policies (e.g., in smart contracts, or in other CPSC components).

3.9.3 Studies in the Same Domain

None of the legal contract DSLs, such as those in [39,127], provides access control modeling. While these works highlight the need to integrate security aspects, they do not propose primitives to define access control policies. To our knowledge, our work is the first approach to integrate access-control primitives into a legal contract language, with code generation.

Table 3.7: A comparison of our SymboleoAC approach and other approaches regarding their access control over protected resources and support for code generation.

Article	Domain	Type of AC model	Onto.	Modeling language	Code generator	Language/tool	Protected resources
Close approaches, from different domains							
Planas et al. [95]	GUI	centralized	Yes	Xtext/textual	Yes	pre-existing RBAC library (Casbin)	event, state, transition
Al-Azzoni et al. [1, 2]	General	centralized	Yes	Obeo/graphical	Yes	Acceleo	asset
General approaches, from different domains							
Basin et al. [18]	General	centralized	Yes	N/A	No	N/A	state
Ben Fadhe et al. [19]	General	centralized	Yes	N/A	No	N/A	state
Kashmar et al. [59]	General	centralized	Yes	N/A	No	N/A	asset
Kim et al. [62]	General	centralized	Yes	N/A	No	N/A	state
Kuhlmann et al. [66]	General	centralized	Yes	N/A	No	N/A	state
Approaches *from the same domain (contracts)*							
Frantz & Nowostawski [39]	Legal Contract	N/A	No	N/A	N/A	N/A	N/A
Wöhrer&Zdun [127]	Legal Contract	N/A	No	N/A	N/A	N/A	N/A

3.10 Conclusion – Second Part of the Literature Review

This second part of the literature review covered the main concepts of Role-Based Access Control, including permissions, subjects, actions, as well as resources, and explored their use in various domains. Building upon these foundational principles, we will customize the RBAC model specifically for legal contracts within our own DSL, SymboleoAC (Chapter 7). By tailoring RBAC to address the unique requirements of legal contracts, we aim to provide more robust access control mechanisms that go beyond traditional approaches, ensuring precise control over contractual obligations, roles, assets, and IoT events. This customization will not only strengthen security but also enable automated and fine-grained access control within legal contracts.

Also, the reviewed literature has demonstrated that while many domain-specific languages provide essential RBAC features, they remain largely focused on modeling access control without extending into the realm of code generation. Our work with SymboleoAC advances the field by integrating RBAC not only at the modeling level but also by supporting code generation for smart contracts in Hyperledger Fabric (Chapter 9), as well as related CPSC components (CEP, message broker, and interfaces to IoT devices), the latter being unique. Unlike other approaches that limit their scope to controlling access over specific types of resources, such as states, events, or assets, SymboleoAC offers a more comprehensive access control framework that includes IoT data.

Chapter 4

Methodology

This chapter presents the methodology used to answer the four research questions of this thesis. This thesis' research was conducted in several steps that are divided into two main phases: (1) theoretical aspects and (2) the design, demonstration, and evaluation of the research artifacts. The chapter explains the steps involved in the development and evaluation of the proposed solutions and artifacts.

4.1 Methodology Overview

The research methodology followed in the thesis is Design Science Research (DSR), initially proposed by Hevner et al. in 2004 [50]. DSR is a problem-solving approach that has been used mainly in Information Science, but also in other disciplines, including Software Engineering [124]. DSR is a constructive approach (in contrast to explanatory science research) that focuses mainly on the design and performance assessment of *artifacts* such as:

- constructs (e.g., DSLs);
- models (e.g., architectures and ontologies);
- methods (e.g., processes, procedures, and algorithms); and
- instantiations (e.g., libraries and tools).

The literature review in the previous chapter highlighted important gaps in the support of legal smart contracts in relation to different qualities (availability, robustness, privacy, security, and legal compliance) and tool support (for code generation and deployment on a Cyber-Physical System (CPS) architecture). This thesis focuses on security, privacy, legal compliance, and tooling issues, in the context of the SYMBOLEO ecosystem.

The primary artifacts in this thesis include the extension of a DSL to support actions, notifications, and RBAC-based privacy/security qualities (SYMBOLEOAC); an extended

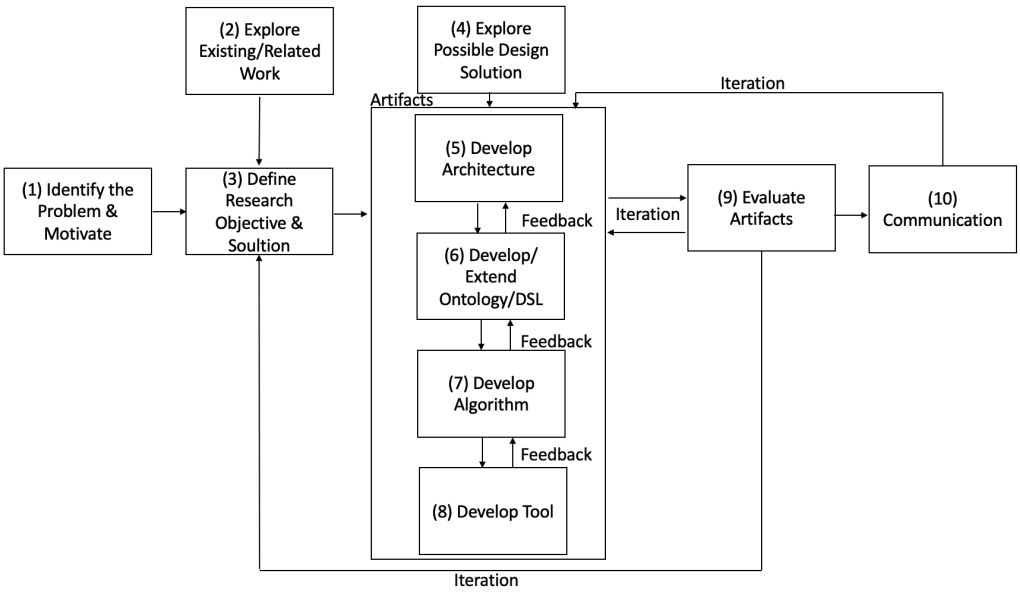


Figure 4.1: Thesis methodology adapted from Peffers' DSR process model [94].

ontology with JavaScript implementation (SymboleoACJS); and an code generation tool (SymboleoAC2SC) enabling the deployment of legal smart contracts on a CPS architecture composed of a specific DLT platform (Hyperledger Fabric) and other important components such as a message broker and a CEP engine.

As another primary artifact, an API (i.e., the SymboleoAC App), implemented in JavaScript and Java, was designed to support secure interaction with all SymboleoAC architecture runtime components including the generated smart contract, IoT sensors, message brokers, the CEP engine, and other on-chain/off-chain services (e.g., for validating identities). This API handles real-time event publishing, subscribing, filtering, smart contract invocation, and so on. It acts as the core bridge connecting blockchain-based contract logic with external cyber-physical data flows. The capability and boundary of the solution have been determined by their utility in different case studies (real contracts).

This thesis methodology includes several steps, shown in Figure 4.1. In addition, an iterative approach has been applied from the evaluation phase (Step 9) back to the artifacts design and development phase (Steps 5-8) to identify the limitations and vulnerabilities of the proposed language and generator and enhance the conversion accordingly until we obtain a new version where the limitations are absent, and the generated smart contract code functions effectively as an actual contract. Feedback received from reviewers of the publications has also affected our solution.

These steps are broken down into two main phases. Steps 1 to 4 in the thesis methodology are the first phase, representing the theoretical part on which the thesis has been built. Steps 5 to 10 are the second phase and are aimed to design, demonstrate, and evaluate the artifacts. In addition, the artifacts are designed and evaluated using rigorous methods, following the seven guidelines proposed by Hevner et al. [50]. Table 4.1 summarizes the seven DSR guidelines. The subsequent sections elaborate on how these guidelines were

applied in this thesis.

Table 4.1: Hevner's seven Design Science Research guidelines [50]

Guidelines	Descriptions
(1) Design as an artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
(2) Problem relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
(3) Design evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
(4) Research contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
(5) Research rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
(6) Design as a search process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
(7) Communication of research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

4.2 Artifacts

According to guideline (1), each research question contributed to the following artifacts:

- Construct:
  - an extension of the SYMBOLEO language that supports advanced functionality: actions, notifications, RBAC-based privacy/security qualities (SYMBOLEOAC), and Cyber-Physical System (CPS) elements (e.g., for IoT device).
- Models:
  - an architectural model of CPS components (e.g., a message broker and a CEP engine), a blockchain (namely Hyperledger Fabric), and smart contracts, specifying how different components interact with each other (generated using SYMBOLEOAC).
  - an RBAC-based SYMBOLEOAC ontology, specified using Umple, that extends the earlier SYMBOLEO ontology

- Methods (Procedures):
  - a procedure for configuring the SymboleoAC architecture elements (CEP, message broker, and IoT devices), to support interactions with smart contracts.
  - a procedure for verifying that access control rules are correctly enforced at runtime.
  - extensions to existing procedures for converting SymboleoAC specifications to smart contract.
- Instantiations (Tools):
  - SymboleoAC2SC, an extended code generation tool for SymboleoAC specifications with built-in features enabling the deployment of legal smart contracts on a CPS architecture composed of a specific DLT platform (Hyperledger Fabric) and other important components such as a message broker and a CEP engine. This tool instantiates the procedures for verifying access control rules and for converting specifications.
  - an improved IDE for SymboleoAC specifications.
  - an implementation of the ontology and semantics of SymboleoAC (including controller and preauthorization rules) into a reusable library called SymboleoACJS.
  - SymboleoAC App, an API written in JavaScript and Java, which instantiates the SymboleoAC architecture model and procedure to enable interaction between the generated smart contract and configured architectural components.

4.3 Problem Relevance

Following guideline (2) of DSR, part of the problem relevance was mentioned in Sections 1.1 and 1.2. Also, Chapter 3 presents a comprehensive literature review to highlight the significance of this study and clarify the existing gaps in the current state-of-the-art. Additionally, this research develops artifacts to address relevant problems:

- With Industry 4.0/5.0 initiatives driving digital transformation, and the need to better comply with contractual obligations, integrating smart contracts into CPSs is essential for monitoring transactions across interconnected devices, ensuring adherence to contractual clauses and reporting violations and other anomalies early [114].
- As industries in various sectors (including finance, energy, healthcare, and supply chain) face increasing demands for data privacy and protection due to sensitive information and stringent regulations, there is an increasing need for secure data handling in smart contracts. This includes the use of encryption, access controls, secure storage, and other protective measures [126].

4.4 Research Process

Following guidelines (5) and (6) of DSR, we selected a research process inspired by Peffers [94] for Information Systems. Also, to provide explicit and reproducible systematic literature reviews (including mapping reviews), we followed Okoli's approach [88] to learn about related work and to enable a comparison with this thesis work. In this thesis, we build on the previous work of Sharifi [110] and Rasti [99], and we extend the approach taken by Rasti [99] for generating smart contract code. Also, we address the research questions and objectives of this thesis. The detailed research processes of each research question are explained below.

For **RQ1**, first, we conducted a systematic mapping review [6], whose results were reported in Chapter 3, to examine the state of the art on how CPSCs are typically realized for compliance monitoring. It provides background and identifies gaps related to architecture frameworks. The review identifies several limitations in existing CPSCs, particularly a lack of tools to support code generation and deployment within CPS architectures. By analyzing various state-of-the-art approaches, it became clear that a reliable messaging infrastructure and event-driven processing mechanisms are important for supporting the integration of CPSs with smart contracts. Drawing on insights gained from the mapping review and inspired by the work of Rosa-Bilbao et al. [104] and the work of Boubeta-Puig et al. [24], a new architectural framework is developed in Chapter 5. Next, technologies for event processing mechanism (specifically, CEP) and messaging infrastructure (specifically, message broker) were selected and installed. A smart contract was generated to connect and exploit these components as a proof of concept. The code generator has been augmented with the necessary interfaces to produce the additional configuration, communication, and rule handling code automatically. The generated smart contract can now interact with off-chain components by handling event-driven invocation and emitting notifications aligned with predefined contractual rules. To support this integration, an API (the SymboleoAC App) was implemented in JavaScript and Java to manage authenticated and encrypted communication between the generated smart contract, the message broker, the CEP, and IoT devices. This API facilitates identity validation, event filtering, publishing and subscribing operations, and the forwarding of off-chain events that trigger on-chain contractual actions.

For **RQ2**, the process began with identifying the limitations of the existing Symboleo language as specified in Section 1.2. In the next step, we used Eclipse's Xtext [20] to define the grammar and syntax of SymboleoAC by extending the existing Symboleo grammar [109]. Additionally, we updated the Symboleo IDE [92] with static semantic rules for checking the well-formedness of SymboleoAC specifications. We have incorporated language and tool support for assignment expressions (addressing **RQ2-1**), allowing for dynamic variable assignments during contract execution [90, 101]. Additionally, we have added RBAC constructs addressing privacy and security concerns, enabling SymboleoAC to declare policies and rules (addressing **RQ3-3**). We extended the language to include a controller for each resource, derived from the semantics of legal concepts, and pre-authorization rules that specify resource access for various roles within a legal contract (addressing **RQ3-4**). Automated notification features were also implemented to trigger

notification events (e.g., violations of contractual obligations) during contract execution. Furthermore, the IDE support integration with external data sources (i.e., IoT devices), enabling smart contracts to access real-world information and events.

For **RQ3** on SYMBOLEO extensions related to privacy and security, we studied the popular Role-Based Access Control (RBAC) model and its fundamental concepts along with significant works on RBAC, as reported in the second part of Chapter 3, to identify fundamental concepts and interrelations, particularly focusing on how these concepts can be customized and applied in legal contract modeling. Next, an access control model was developed as an ontology (addressing **RQ3-1**). This RBAC ontology was then integrated with the SYMBOLEO ontology [92], resulting in SYMBOLEOAC's ontology (addressing **RQ3-2**) [7]. Additionally, we introduce two new classes in SYMBOLEOAC's ontology to support access control over resources: (1) StateTransition, capturing contract state changes with timestamps, and (2) DataTransfer, which enables access to IoT data like temperatures or locations. Also, as SYMBOLEOAC is tailored for contract execution, we defined two sets of rules: (1) rules that determine a controller for every resource, and (2) pre-authorization rules that determine who has access to what, on the basis of the role they play in contract execution.

Additionally, our approach to utilizing the SYMBOLEOAC access control mechanism supports two core security modes: the *detection* and *prevention* of unauthorized access to resources. In our architecture (see Figure 5.1), *authentication* ensures that every entity whether a role, IoT device, listener, message broker, or CEP, is validated using an authentication mechanism (e.g., certificate-based authentication) before it can participate in the system. Once authenticated, *authorization* determines whether the entity's assigned role has the appropriate permissions to access, read, write, publish, subscribe, or invoke specific contract operations. In this way, authentication provides prevention by blocking unverified entities from entering the system, while authorization provides detection and prevention by verifying that even authenticated entities only perform actions they are explicitly permitted to perform according to the SYMBOLEOAC policy.

Lastly, still for **RQ3**, we have adopted the approach of supporting new Non-Functional Requirements (NFRs) by integrating them directly into the original SYMBOLEO DSL. This decision was driven by several key factors discussed by Ameller et al. [11] and summarized in Table 4.2. By embedding NFRs such as security and privacy directly into the language, we can ensure that these requirements are consistently addressed throughout the modeling and code generation stages, which aligns with the metrics of expressiveness and code generation efficiency highlighted in the comparison. Furthermore, this approach minimizes the risk of misalignment between functional and non-functional requirements, as they are both handled within the same tool-supported framework. While this integration may increase the complexity of the DSL and may pose challenges for maintainability concerns identified in Table 4.2, we believe that the benefits of a unified specification and the ability to enforce NFRs at every stage outweigh these concerns.

For **RQ4**, to support the new SYMBOLEOAC concepts, we extended the JavaScript library of the original SYMBOLEO ontology (SYMBOLEOJS [101]) by formalizing the SYMBOLEOAC ontology as an Umple model [38, 74], from which we automatically generated

Table 4.2: Comparison of alternative approaches for integrating NFRs in DSL-based code generation

Aspect	**(a) Adding NFR syntax and semantics to the original DSL**	**(b) Creating a separate DSL for the new NFRs**
Code Generation	Code generation is straightforward since everything is defined in one place.	Code generation may become complex as it needs to merge two DSLs, which may lead to unexpected errors.
Expressiveness	Models and variables can be updated as soon as new NFRs are needed, making it easier to ensure consistency between generated code and models.	Can lead to inconsistency between the specifications and NFR variables. Updates are necessary to maintain consistency.
Applicability	Easy to achieve in practice, as NFRs are directly incorporated into the DSL.	Harder to achieve in practice due to the need to manage and synchronize two separate DSLs.
Maintainability	NFR changes may require changing the entire DSL.	NFR changes can be handled independently without impacting the original DSL.
Separation of Concerns	Lower, as different concerns are mixed.	Higher, as each DSL focuses on specific and different concerns.
Reusability	Less reusable if it is closely tied to specific functionalities	Can be reused with other DSLs.
Change Oversight	Changes to both functional and non-functional requirements can be managed within a single DSL, making it easier to track updates.	Hard to track changes between original models and new NFR specifications, as updates must be done both in the original model and in the new specification.

an Ecore model visualizable as a UML class diagram (illustrated in Fig.7.3) and Java files with many utility methods for instantiating, navigating, and modifying concept instances[1]. Then, we manually converted the generated Java files into their equivalent JavaScript form[2], since Umple does not generate JavaScript code directly. This resulted in the SymboleoACJS reusable library (4,491 LOC) for the code generator (**RQ4-1**). The JavaScript versions of several ontology classes are discussed in Chapter 9, where

[1] https://shorturl.at/s6Gtn
[2] https://shorturl.at/XIhRc

each ontology class has a corresponding JavaScript class. In parallel, we extended the SymboleO2SC code generator (written in Xtend [20]), which generates executable smart contracts for Hyperledger Fabric[3] from SymboleO specifications [101]. The updated tool, SymboleoAC2SC, supports the new SymboleoAC access control concepts via the expanded SymboleoACJS library (**RQ4-2**). SymboleoAC2SC now integrates (1) default security settings (controller and pre-authorization rules) and (2) access control policies specified at design time. Tested with multiple instances version of a meat sale contract case study [7], SymboleoAC2SC successfully generated smart contracts that specify each resource's controller and access rules. We have also expanded SymboleoACJS and SymboleoAC2SC to incorporate additional CPSC architectural components and notification (Figure 5.1) and evaluated them with multiple instances of another case study (vaccine procurement contract).

4.5 Evaluation

Guideline (3) of DSR suggests creating evaluation methods to assess the feasibility, effectiveness, and quality of the research artifacts. The above artifacts have been evaluated iteratively, through several case studies, for the four RQs. These case studies include multiple instances (variants) of:

1. Our meat sale contract [7], extended to incorporate IoT data and more policies and rules;

2. A new vaccine procurement contract, used to demonstrate the integration of multiple IoT data streams in a complex supply chain agreement.

Steps 5 to 9 in Figure 4.1 have been performed iteratively to continuously assess and refine the identified artifacts. Feedback from the evaluation has been used to incrementally enhance the architecture, ontology/DSL, method, and tool, ensuring they were progressively improved based on practical insights. Table 4.3 shows the iterative evaluation process for the artifacts. The developed artifacts have been evaluated in terms of correctness (i.e., the generated functions reflect the architecture and ontology), executability in the CPS environment, and access control enforcement, with an assessment of their limitations.

Additionally, unit tests have also been used for more conventional software testing. These tests have been developed to ensure that the implemented classes of the ontology artifact effectively enforce access controls and align with the intended behavior of the proposed architectural framework. Furthermore, we have developed multiple test scenarios for the generated smart contracts to evaluate their ability to enforce access control, detect violations, and validate behavior. The tests have confirmed that the generated smart contract code adheres to the specified security policies, restricts access as intended, and is capable of securely producing and consuming events from and to the outside world.

[3] https://www.hyperledger.org/projects

Table 4.3: Iterative evaluation process for the artifacts

Iteration #	Evaluation	Result
Iteration 1	Initial implementation of variable assignment using Eclipse's Xtext framework for grammar definition, and Xtend for generating the smart contract code (via SYMBOLEO). Tested with the new vaccine contract.	Modifications and enhancements to the tools, including SYMBOLEO's Xtext grammar, IDE, and code generator.
Iteration 2	Refined the tool based on feedback from the first iteration, improving the Xtext grammar. Re-tested with other contract case studies.	Modifications and enhancements to the tools.
Iteration 3	Initial design of the proposed architectural framework with basic components (message broker, CEP, smart contract generated using SYMBOLEO). Manual implementation of the notification mechanism. Tested with the meat sale contract.	Identified that some components were missing or needed adjustments, such as refined communication between the message broker and the smart contract in the proposed architecture.
Iteration 4	Refined the architecture based on feedback from the previous iteration.	Modifications and improvements to the architecture.
Iteration 5	Designed the access control ontology using the Eclipse Modeling Framework (EMF), followed by the design of the SYMBOLEOAC ontology, which integrates the access control ontology with the earlier SYMBOLEO ontology.	Modifications and improvements to the ontology/DSL.
Iteration 6	Refined the SYMBOLEOAC ontology based on feedback from the previous iteration.	Modifications and improvements to the ontology/DSL.
Iteration 7	Implemented the SYMBOLEOAC reusable library. Tested with multiple unit tests.	Modifications and improvements to the tools.

Iteration 8	Implemented the automated configuration of the message broker, CEP engine, and extended smart contract. Updated SYMBOLEO's Xtext grammar, and used Xtend for generating code to provide the necessary API for the smart contract to communicate with the message broker and CEP. Tested with the meat sale contract with simple IoT data to assess the architecture's ability to handle event processing and efficient contract monitoring.	Modifications and improvements to the architecture, procedures, ontology/DSL, and tools.
Iteration 9	Refined the tools based on feedback from the previous iteration.	Modifications and improvements to the ontology/DSL, procedures, architecture, API, and tools.
Iteration 10	Case study 2: vaccine procurement: tested and covered different aspects of SYMBOLEOAC, including access control, control actions, notification feature, and integration with external data sources (message broker and CEP), with more IoT data.	Modifications and improvements to the procedures, architecture, API, and tools.
Iteration 11	Testing of multiple instances of multiple contracts, at the same time, with some shared parties that have different permissions.	Modifications and improvements to the procedures, API, and tools.

4.6 Contributions

Regarding guideline (4) of DSR, the primary and secondary contributions of this thesis were already detailed in Section 1.6, and illustrated in Figure 1.1.

4.7 Communication Through Publications

According to guideline (7) of DSR, it is crucial to communicate research findings in various ways. This thesis has led to 11 publications (already enumerated in Section 1.6), co-authored with my co-supervisors and other members of our research laboratory. These include:

- Four journal publications [6, 8, 90, 101] (two as main author),
- Three conference publications [4, 7, 81] (two as main author), and
- Four workshop publications [9, 12, 85, 113] (one as main author).

Most of these publications were helpful in collecting feedback that influenced later iterations of the thesis work.

Chapter 5

SymboleoAC Architectural Framework

This chapter presents the proposed architecture of a Cyber-Physical Smart Contract (CPSC) designed to facilitate the integration of smart contracts with a message broker, a Complex Event Processing (CEP) engine, and IoT devices. This architecture illustrates how event-driven smart contracts can be modeled and connected to data sources and sinks (through the message broker) as well as to the CEP (for event aggregation), and then transformed into executable code (as will be detailed in Chapter 9) for development within a Cyber-Physical System (CPS).

5.1 SymboleoAC Architecture

This section introduces the SymboleoAC architecture, which facilitates the integration and collective use of smart contracts, the CEP, and the message broker. The architecture contains four complementary parts: i) the Design-Time Layer, ii) the On-Chain Run-Time Layer, iii) the Off-Chain Run-Time Layer, and iv) the Interaction Flow Dimension.

Figure 5.1 shows this SymboleoAC architecture. This architecture integrates the SymboleoAC modeling environment, smart contract generation, and secure run-time components to enable end-to-end enforcement of cyber-physical smart contracts. Our architecture separates design-time assets (ontology, specifications, rules, grammar, and code generation) from run-time enforcement on the blockchain platform. We chose Hyperledger Fabric [13] as our blockchain-based distributed ledger because this is an enterprise-level, open-source platform that supports permissions (most useful in an access control context), as well as different programming languages (JavaScript/Node.js, Go, and Java). To achieve this integration, the architecture is structured into distinct parts, each addressing a specific phase in the smart contract lifecycle from specification and code generation to deployment and event-driven execution:

The Design-Time Layer. In this layer, SymboleoAC specifications and rules are authored and transformed into executable JavaScript smart contracts through the SymboleoAC2SC code generator and the SymboleoACJS library (the later implementing

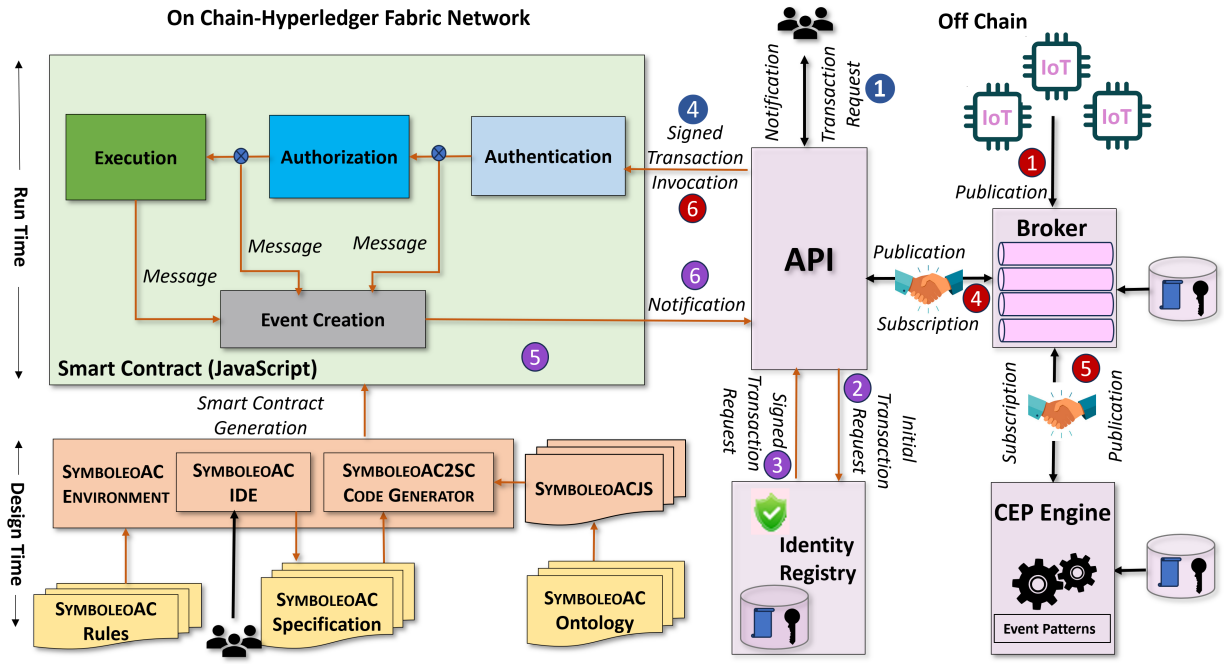


Figure 5.1: CPSC architecture for the integration of smart contracts (generated using SymboleoAC) with the API, a digital identity registry, a message broker, a CEP engine, and IoT devices.

the SymboleoAC ontology). This layer provides the modeling environment and tools necessary to define contractual elements, permissions, and policies, before deployment.

The On-Chain Run-Time Layer. This layer is deployed on the distributed ledger (Hyperledger Fabric network), where the generated smart contracts execute contractual operations. It includes authentication, authorization, and execution modules that ensure secure and policy compliant contract enforcement. Upon successful transaction execution, blockchain events are generated and propagated to the off-chain components.

The Off-Chain Run-Time Layer. This layer connects the blockchain to external entities through an API gateway, identity registry, message broker, CEP engine, and IoT devices. It handles real-time event publication, subscription, filtering, and transaction invocation. The message broker mediates communication between IoT devices and the CEP engine, while the API and identity registry manage transaction requests and digital identities.

The Interaction Flow Dimension. This dimension defines the communication sequence among layers, covering IoT data publication, subscription, event filtering, message exchange through the message broker, smart contract invocation, and run-time notifications.

Together, these four parts provide an event-driven architecture for integrating formal contract specifications with decentralized execution environments. The architecture separates on-chain and off-chain responsibilities in order to reduce coupling between components and support integration with external services and heterogeneous IoT devices.

Simpler alternatives, such as direct API calls, oracle-based approaches, or basic filtering mechanisms, could potentially be used instead of advanced and mature components such as message brokers and CEP engines. However, both the message broker and the CEP engine are incorporated because our architecture focuses on decoupled communication and support for handling multiple IoT data streams.

- The message broker separates communication between components, allowing sensors, the CEP engine, and smart contracts to interact without directly depending on each other. It also provides built-in publish/subscribe communication, message queuing, and message routing capabilities without requiring these mechanisms to be implemented as part of the smart contract, on a slower and more expensive platform.
- The CEP engine is used to process raw sensor data, detect meaningful patterns, and generate abstract events (relevant at the level of a legal contract) rather than sending all sensor readings directly to the smart contract. This introduces an abstraction layer between IoT devices and the blockchain, reducing unnecessary smart contract invocations and allowing the smart contract to focus primarily on contractual enforcement and execution logic.

The following subsections explain the components included in each part (additional components exist beyond those illustrated in the architecture shown in Figure 5.1).

5.1.1 The Design-Time Layer

The design-time layer contains the SymboleoAC environment, which includes the following components:

- SymboleoAC IDE: an Eclipse-based integrated development environment, built using Xtext, for the SymboleoAC language (Chapter 8). It enables users to model contracts with access control extensions, including new domain concepts (e.g., **DataTransfer**) for IoT data and attribute qualifiers, as well as policies and rules, while providing features such as syntax highlighting, auto-completion, and well-formedness validation.
- SymboleoAC2SC: a code generator that transforms validated SymboleoAC specifications into executable smart contracts for Hyperledger Fabric (Chapter 9). It embeds contractual elements and access control policies and rules into blockchain ready chaincode. It also generates a ready-to-use SymboleoAC API that connects with the external environment via a message broker and a CEP to handle IoT data and event-driven interactions securely (Chapter 6).

- SymboleoACJS: a JavaScript core library used by SymboleoAC2SC that implements the SymboleoAC ontology to support the run-time execution of generated smart contracts (Chapter 7 and Section 9.1). It provides reusable functions for managing contract states, evaluating rules, and enforcing access control policies during interaction with Hyperledger Fabric.

5.1.2 The On-Chain Run-Time Layer

The on-chain platform is a key component of our architecture and is responsible for receiving and processing events, as well as for sending notifications that come from the off-chain run-time layer through the SymboleoAC API. The main components of this layer are:

- Two security components:
  - *Authentication*: in our architecture, authentication is managed by the underlying blockchain platform, which verifies the digital identity of each participant before granting access. When a transaction is initiated, the platform ensures that the user's credentials are valid, issued by a trusted authority, and securely stored in the digital registry. This mechanism upholds the confidentiality of user identities, preserves the integrity of authentication records, and enforces accountability by ensuring that only authorized participants who are registered and enrolled by the administrator or the SymboleoAC policy controller (e.g., the regulator) can access the network and invoke contract operations.
  - *Authorization*: once the user is authenticated, authorization is enforced by the SymboleoAC access control code embedded within each smart contract transaction. It verifies whether the authenticated user has the appropriate permissions to perform the requested action using the `hasPermission()` function, and also evaluates any overriding policies – either explicit or implicit (i.e., preauthorization rules in Section 7.6) – through the `isValid()` function. This ensures that access to contract resources strictly adheres to the defined policies and rules.
- Smart Contract Listener (receiver): listens for events from the message broker and triggers functions in the smart contract to process the data.
- Chaincode Event Listener (sender): listens for specific events or state changes within the smart contract. Once these events occur, it triggers actions to send data or notifications to the outside world, through a message broker and delivered to the intended subscribers.
- Smart Contract: includes all types of smart contract transactions to perform different operations based on the information received by the smart contract listener.
- Hyperledger Fabric Network: provides the blockchain infrastructure that supports the execution of the smart contracts. It includes the consensus mechanism, peer nodes, orderers, and other components necessary for the secure execution and recording of transactions in a distributed ledger.

5.1.3 The Off-Chain Run-Time Layer

This layer is composed of components outside of the distributed ledger environment:

- API (receiver): acts as an entry point for external applications or services to send requests or data to the Hyperledger Fabric Network or smart contracts.
- API (sender): serves as an exit point, allowing external systems to retrieve data from the Hyperledger Fabric Network or smart contracts.
- Message Broker (sender): responsible for routing and transmitting raw data received from a data source (e.g., IoT sensors) to the appropriate destinations, such as the CEP or a smart contract. It acts as an intermediary, ensuring efficient and reliable message delivery.
- Message Broker (receiver): receives processed data from the CEP and routes it to the smart contract for further execution, ensuring proper communication and data flow between components.
- Digital Registry: stores SymboleoAC client identities (e.g., users, administrators, sensors, message broker, and CEP engine). The SymboleoAC API loads these identity files and retrieves each client's certificate and private key as part of the authentication process.

Another key component of the off-chain run-time layer is the CEP, responsible for analyzing and aggregating events that come from IoT devices or sensors thought the message broker. In a CEP system, there are typically three main responsibilities involved: (1) Event capturing, which gathers the generated data from IoT devices through message brokers, (2) Event processing, which includes CEP engine for analysing such data, and (3) Event responding, which sends/publishes processed (often more abstract) events to smart contracts though the message broker. Correspondingly, the main CEP-related components in our architecture are:

- Data Listener (Input): listens for new data received from IoT devices through the message broker. The latter acts as an intermediary, delivering data from IoT devices to the CEP system for further analysis.
- CEP Engine: analyzes and monitors received data based on predefined patterns/rules, and generates (more abstract) events, e.g., understandable by a smart contract.
- Pattern Trigger Listener (Output): once the CEP engine has processed the data, the output component sends/publishes the resulting events or processed data to the smart contracts through the message broker. This output triggers actions in the smart contract.

Please note that the specific technologies used for the digital identity registry (e.g., a Certificate Authority and/or a blockchain wallet), message broker, and CEP components can be replaced with alternatives other than those mentioned in the SymboleoAC API implementation (Chapter 6) of the SymboleoAC architecture. However, the following requirements must be satisfied by the selected technologies:

1. **Requirements for the Trusted Digital Identity Registry**
   (a) The Digital Identity Registry shall serve as a trusted entity responsible for issuing, validating, and revoking digital certificates for all participants in the network.
   (b) The Digital Identity Registry shall ensure the authenticity and integrity of digital identities through secure credential management and verification mechanisms.
   (c) The Digital Identity Registry shall support role-based registration and enrollment processes to establish trust among heterogeneous entities (e.g., users, administrators, sensors, and services).
   (d) The Digital Identity Registry shall maintain a secure and tamper-evident registry of issued certificates to enable traceability and accountability across the system.
2. **Requirements for the Complex Event Processing Engine**
   (a) The CEP engine shall allow for the definition and deployment of event patterns to analyze and correlate data in real time, enabling rapid, informed decision-making.
   (b) The CEP engine shall support integration with data streams (inputs and outputs) via message brokers, ensuring seamless interaction between various system components of different origins.
   (c) The CEP engine shall be capable of transmitting the results of event processing through message brokers, enhancing system integration and responsiveness.
3. **Requirements for the Message Broker**
   (a) The message broker shall support the publish/subscribe model.
   (b) The message broker shall support the AMQP messaging protocol to support reliable and standardized message delivery between system components and IoT devices.
   (c) The message broker shall support message routing using AMQP (e.g., exchanges, bindings, and routing keys) to deliver messages to the appropriate consumers.

5.1.4 The Interaction Flow Dimension

The execution flow in our run-time architecture (Figure 5.1) can be summarized by the following steps:

1. Sign the transaction
   - (1) (1) A user or backend system initiates a request to invoke smart contract transactions, such as `trigger_delivered` or `trigger_paid`.
   - (2) The API sends the initial request to the digital identity registry.
   - (3) The registry cryptographically signs the request using the private key and attaches the certificate to prove the origin of the transaction, as well as the user's identity and role.
2. Processing and analyzing IoT data
   - (4) The IoT data are published to a queue in the message broker, awaiting subscription by a consumer component, such as the CEP engine.
   - (5) The CEP engine subscribes to IoT data from the message broker, applies the defined rules, and publishes the resulting event back to the message broker when a condition is met.
3. Submit the signed transaction request to the blockchain.
   - (4) (6) The API validates the signed transaction request and forwards it to the blockchain.
4. (5) Execute the smart contract
   - The smart contract *authenticates* the request by verifying the signature against role attributes derived from the contract specification. If valid:
     - The smart contract *authorizes* the action by checking rules and constraints generated from the contract specification and SymboleoAC rules (see Section 7.6). If valid:
       - The smart contract *executes* the transaction and updates the ledger.
   - For the authentication, authorization, and event creation, the smart contract *creates events* according to each message, in case of success or failure, and notifies the API.
5. Deliver the result through the API
   - (6) The API communicates the execution outcome to the user or backend system.

The architecture also identifies policy enforcement points (PEPs) at both the chaincode level (for transaction execution and event creation) and the API level (during user registration and request submission). The policy decision point (PDP) evaluates access requests against SymboleoAC policies before contract execution or event emission.

A detailed approach for end-to-end encryption and communication flow of the SymboleoAC architecture is explained next.

5.2 End-to-End Secure Communication Approach of the SymboleoAC Architecture

The communication between the architecture components is divided into five phases: enrollment (including registration), connection, publishing, subscription, and notification. UML sequence diagrams (Figures 5.2 and 5.3) illustrate the message exchanges for these phases.

5.2.1 Enrollment Phase

During this phase, illustrated in Figure 5.2, all entities capable of event publishing or subscribing – such as the message broker, the CEP engine, SymboleoAC roles, IoT devices, and API listeners – are registered and enrolled through a digital identity registry (e.g., trusted certificate authority); the certificate issuance for roles is not shown in the sequence diagram for readability purposes, but it follows the same process as for the other entities shown. A unique key is generated and securely stored in the digital registry to identify each entity in subsequent interactions. Once the smart contract is instantiated, the API listener (4 in Figure 5.1) establishes a mutual Transport Layer Security (TLS) protocol connection with the message broker, verifying both identities to ensure secure mutual communication. The same mechanism is applied between IoT devices, smart contract listeners, and the CEP engine to guarantee end-to-end authentication and confidentiality. Additionally, all queues, topics, and their authorized publishers and subscribers (i.e., sensors, CEP, and listeners) are generated and recorded in the message broker's metadata store, where user accounts and permissions are managed internally (locally) – except for SymboleoAC roles, whose permissions are defined and enforced by the policy stored on the ledger.

During the registration sub-phase, the list of roles defined in the SymboleoAC contract specification is extracted and stored on the blockchain ledger, ensuring that they cannot be altered or tampered with. This process is executed through a dedicated smart contract transaction (`storeRolesPolicy()`), ensuring that the defined roles cannot be altered or overwritten by unauthorized entities. Only assigned participants, specifically, the blockchain Administrator or the SymboleoAC Policy Controller (e.g., Regulator) are authorized to invoke this transaction, which stores a signed hash of the roles list for verification and auditability. Another retrieval transaction (`getRolePolicy()`) allows only these trusted authorities to access the stored policy and use it during user enrollment via the digital identity registry (e.g., a trusted certificate authority), ensuring that all registered participants adhere to the original role definitions established by the SymboleoAC specification. Together, these steps ensure the **integrity**, **accountability**, and **traceability** of all identities and roles across the system.

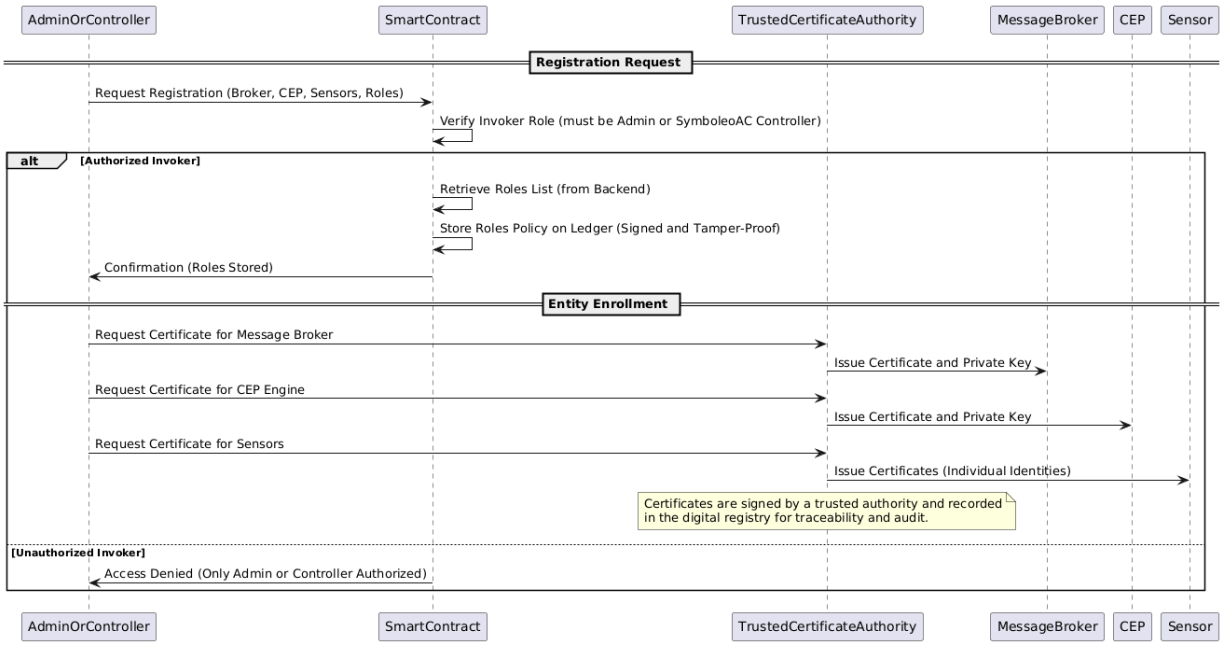


Figure 5.2: Enrollment phase, including the registration sub-phase.

5.2.2 Connection Phase

For components such as the CEP engine, API listeners, or other back-end roles, this certificate-based mutual authentication is sufficient because these components operate within controlled, server-side environments that are protected by secure network boundaries and managed access policies.

However, IoT devices differ significantly in their deployment. They are often distributed at the network edge, in untrusted environments, making them more susceptible to key extraction, device spoofing, or other such attacks. To address this risk, IoT clients are required to provide an additional **credential-based authentication** step, where the device's username and password, stored locally in the message broker's secure datastore, are verified alongside its certificate. This dual mechanism follows the principle of **defense in depth** [105], ensuring that authentication remains robust even if a device's private key is compromised.

5.2.3 Publishing Phase

During this phase, the *message broker* manages publish requests from authenticated producers for specific queues or topics. After establishing a mutual TLS connection in the previous phase, each producer (e.g., an IoT device or the CEP engine) submits an encrypted message to `sendToQueue(queueID, Message)` operation. The message broker verifies publish authorization before accepting a message to ensure that the publisher has the appropriate permissions to write to the targeted queue or topic, which can be abstracted as `checkRight(publisherID, WRITE)`. In practice, this check is enforced internally using

built-in permission metadata as explained in the next chapter, rather than being implemented as an explicit function in our case.

The *IoT device* acts as a producer that publishes encrypted sensor readings to the designated `sensor_data` queue. The *CEP engine*, upon detecting a rule violation or a condition defined by its event patterns (e.g., exceeding a temperature threshold multiple times), publishes a processed event or alert to the `alert_events` queue ❺. Both publishing actions occur over secure, certificate verified connections to maintain the **confidentiality** and **integrity** of transmitted data.

The message broker enforces a reputation-based security model [58], where any unauthorized attempt to publish to a restricted queue results in a **reputation penalty** for the offending publisher. If repeated violations occur and the publisher's reputation score drops below a predefined threshold specified by the message broker administrator, the entity is **automatically blacklisted**, preventing further interactions. This mechanism ensures continuous compliance with access control policies and strengthens **accountability** and **non-repudiation** within the SymboleoAC architecture.

As illustrated in Figure 5.3, this phase provides a high level of confidence that only trusted entities with verified identities and valid permissions can generate or propagate events, thereby preserving the **security and reliability** of end-to-end communication within the SymboleoAC architecture.

5.2.4 Subscription Phase

In this phase, the **message broker** manages subscription requests from authenticated consumers for specific queues or topics. Once the mutual TLS session has been established, each subscriber sends a request to `consume(queueID, Message)` operation. The message broker performs a local authorization check through the `checkRight (subscriberID, READ)` function to ensure that the subscriber is permitted to receive messages from the designated queue or topic.

The **CEP engine** subscribes to the `sensor_data` queue ❺ to receive IoT sensor readings. It continuously evaluates these events using its defined event processing logic and correlation rules. Upon detecting a rule violation or fulfillment of a defined condition, the CEP engine publishes a corresponding alert to the `alert_events` queue ❹. The API's **Smart Contract Listener** then subscribes to this queue to receive the alerts and trigger corresponding smart contract transactions ❻. Each transaction is executed on-chain (❺) using two security layers mentioned earlier, ensuring tamper-avoiding of the contractual execution.

Throughout this phase, the message broker enforces a reputation-based access control mechanism, as in the publishing phase. Any unauthorized attempt to subscribe to a restricted topic results in a reduction of the subscriber's reputation score, and repeated violations may lead to blacklisting. This dynamic mechanism maintains the **integrity**, **availability**, and **accountability** of the event-driven data flow across the SymboleoAC architecture's components.

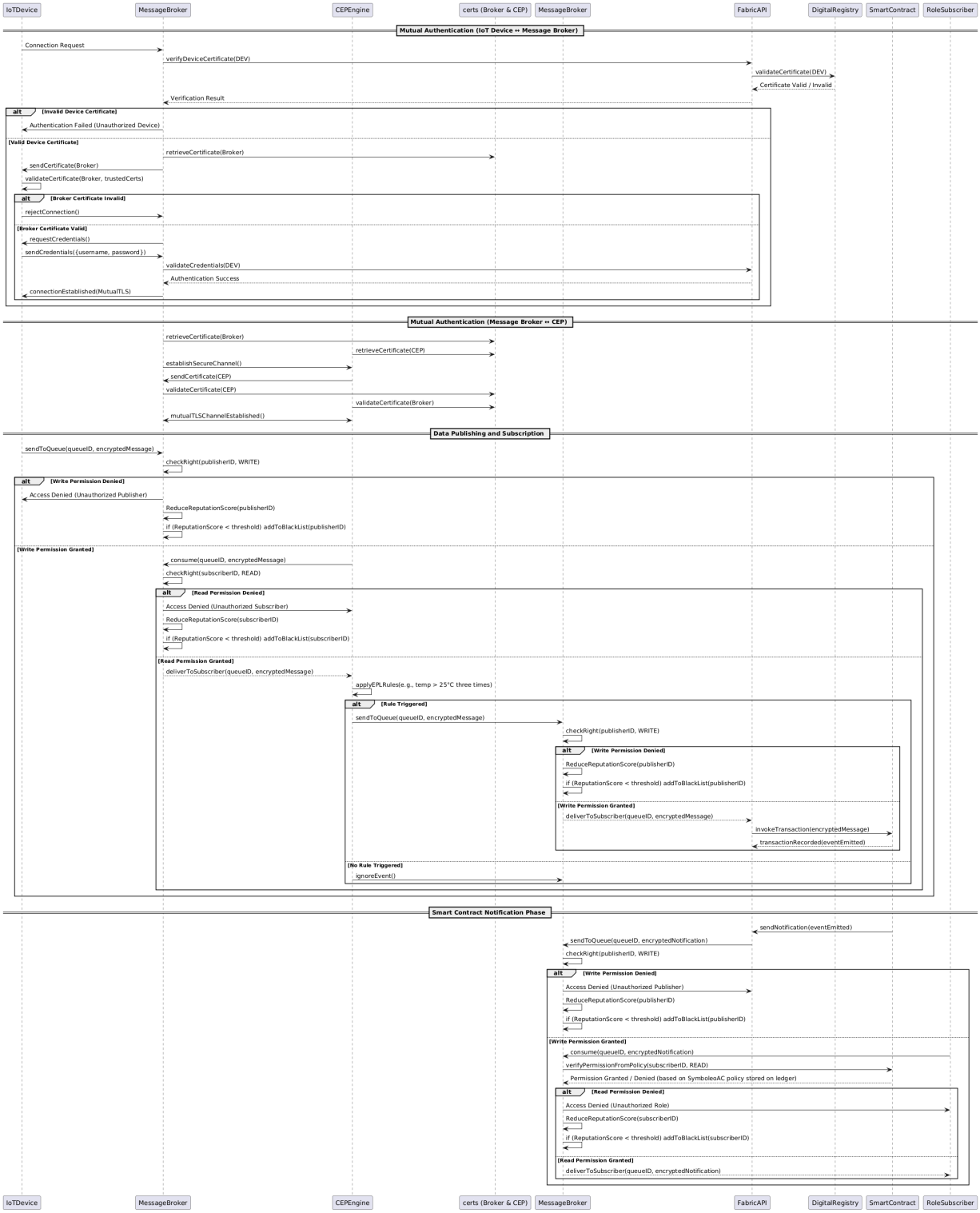


Figure 5.3: Connection, publishing, subscription, and notification phases.

5.2.5 Notification Phase

In this phase, the **Smart Contract** emits a notification event (6) following the successful execution of a transaction, such as the fulfillment or violation of a contractual obligation. The **API listener** captures the emitted event and publishes it to the message broker using the `publishToQueue(queueID, encryptedNotification)` operation. Interested roles (e.g., *seller*, *buyer*) then subscribe to receive the corresponding notifications through the message broker.

Unlike earlier phases, read permissions in this phase are not enforced solely through the message broker's local metadata but are instead derived from the SymboleoAC policies stored on the ledger. the message broker consults the smart contract to verify the subscriber's access rights using the SymboleoAC policy-based authorization mechanism (`hasPermission ()` and `hasPermesstionOnLegalPosition()`). This ensures that only roles explicitly authorized in the formal contract specification, or those holding specific privileges (i.e., preauthorized), such as the *performer*, *rightholder*, or *controller* of SymboleoAC resources, are permitted to access and read the emitted events notifications.

This final phase closes the loop in the event-driven architecture, enabling secure, policy-compliant, and auditable dissemination of contractual state changes. By combining certificate-based authentication, policy-driven authorization, and reputation-aware event management, the system upholds end-to-end **confidentiality**, **non-repudiation**, and **integrity** across all communication layers.

5.2.6 Additional Notes

It is important to note that all these activities are executed through a separate API composed of many functions, ensuring that they do not affect the performance of the smart contract on the blockchain. For instance, the publishing component includes an API function that contains a listener that listens to smart contract events and forwards it to the message broker. On the subscriber side, another API function that contains a listener that listens to broker messages and enforces access control, allowing only authorized users to retrieve the information.

All authorization decisions for roles are driven by the policies and rules of the smart contract (generated from the SymboleoAC specification) and consumed by these external components. Since there is no direct interaction between the API and the smart contract beyond event emission and rule retrieval, the security, integrity, and execution performance of the on-chain logic are preserved.

5.3 Conclusion

This chapter presented the SymboleoAC architectural framework, which integrates formal contract specifications, secure message-driven communication, and blockchain-based enforcement into a cohesive end-to-end system. The architecture was organized into four

parts that collectively enable the modeling, generation, and execution of cyber-physical smart contracts. Each part was designed to ensure confidentiality, integrity, accountability, and policy compliance through mechanisms such as certificate-based authentication, mutual TLS connections, and reputation-aware access control. Furthermore, the interaction phases (enrollment, connection, publishing, subscription, and notification) define how trusted components (IoT devices, CEP engine, message broker, and smart contracts) collaborate to securely exchange data and enforce contract state transitions.

The next chapter defines the architecture's API and provides an implementation of the architecture with specific technologies that satisfy the requirements described earlier. While this architecture defines the structural and procedural foundation for secure event-driven interaction, its effectiveness depends on a precise and formal representation of roles, permissions, and policies. Chapter 7 will then introduce the *SymboleoAC Ontology and Access Control Model*, which provides the conceptual and semantic underpinnings of the framework. It formalizes how roles, resources, and privileges are defined, related, and reasoned about within the SymboleoAC environment, establishing the foundation for policy derivation and enforcement in both design-time and run-time contexts.

Chapter 6

SymboleoAC API and Architecture Deployment

For the deployment of the SymboleoAC architecture (Figure 5.1), this chapter presents the Application Programming Interface (API) that has been developed. The SymboleoAC API provides the middleware required to connect the generated smart contract (generated using SymboleoAC2SC) with the off-chain components, including the message broker, the CEP engine, the identity registry (or wallet), and IoT devices. This chapter explains the technologies selected for an example implementation of the SymboleoAC architecture, the configuration of its components, and the implemented functionalities that form the backbone enabling secure end-to-end communication of Cyber-Physical Smart Contract (CPSC) within the proposed architecture.

6.1 SymboleoAC Architecture Development Overview

This section gives an overview of an implementation of our secure, end-to-end and event-driven SymboleoAC architecture (Figure 5.1) and its deployment.

For the deployment of the SymboleoAC architecture, Docker[1], a platform designed to automate the deployment of applications in lightweight, portable containers, is used. The blockchain platform (namely Hyperledger Fabric[2]), including the generated smart contract, is deployed in Docker containers to ensure a consistent, isolated, and reproducible environment. While Docker provides a simple deployment environment, using blockchain platforms other than Hyperledger Fabric is possible; however, this would require re-implementing the smart contract logic (e.g., in Solidity for Ethereum[3]) and adapting identity registration and management according to the requirements of the new blockchain platform.

[1] https://www.docker.com/
[2] https://hyperledger-fabric.readthedocs.io/en/release-2.1/build_network.html
[3] https://www.soliditylang.org/

Communication with the message broker and the CEP engine, i.e., the off-chain run-time layer, is implemented as a Node.js[4] application running outside the Hyperledger Fabric containers. This application, which implements part of the SymboleoAC API, connects to the Fabric network through the Fabric SDK[5] for Node.js, allowing it to register off-chain entities (clients), including IoT devices, the CEP engine, the message broker, and participants. It also manages their identities, invokes transactions on the smart contract, receives notifications, and handles other run-time interactions.

We are using *virtual* devices that represent real IoT devices, in order to provide abstraction and enable simpler testing.

The message broker is deployed and running as its own server, and the CEP is executed as a separate run-time process. However, both are hosted on the same machine (for testing purpose) and are independent from the Hyperledger Fabric and Node.js instances. The SymboleoAC Node.js application interacts with the message broker and the CEP through their APIs and protocols. Thus, each component can run on its own dedicated server, allowing better scalability, reliability, and resource isolation.

This implementation combines multiple complementary technologies: the blockchain provides immutability and transparency for stored data, the message broker ensures role-based communication between these components, and the CEP engine processes and detects real-time events derived from IoT data received though the message broker. Together, these components form a robust runtime environment for executing Cyber-Physical Smart Contracts (CPSCs).

When selecting a CEP system and a message broker for our architecture, it is essential to ensure they meet the criteria mentioned in Section 5.1 to enable efficient real-time data processing and system integration. A detailed implementation of these off-chain components is described below.

6.2 IoT Devices

For this thesis, we support virtual IoT sensors that mimic the real sensors of our case studies. Each virtual sensor periodically produces readings based on a defined *base value* and *variation* to simulate environmental changes. This approach allows testing the SymboleoAC architecture without using physical IoT devices.

Each IoT device is treated as an off-chain client in the SymboleoAC Node.js application and must be authenticated before sending data to the message broker queue. IoT devices without proper authentication are rejected before they can publish any data to the message broker. Authentication for IoT devices occurs at two levels:

- **Certificate-based authentication**: The IoT device is validated through its digital certificate stored in the Hyperledger Fabric *wallet*. Thus, each sensor is assigned a

[4] https://nodejs.org/en
[5] https://hyperledger-fabric.readthedocs.io/en/release-2.2/fabric-sdks.html

unique X.509[6] certificate signed by the Fabric Certificate Authority (CA). When an IoT device (sensor) publishes data to the message broker (e.g., RabbitMQ), it signs the message using its private key. If the sensor's identity does not exist in the system, or if the signature is invalid (e.g., expired or corrupted), the sensor is rejected.

- **Credential-based authentication**: The message broker verifies the IoT device's credentials (i.e., username/password) before allowing it to publish to a queue.

Additionally, for IoT devices and the message broker, we use mutual Transport Layer Security (TLS) and the Advanced Message Queuing Protocol (AMQP) to ensure encrypted message delivery and secure communication.

After successful authentication, the sensors can send data payloads, including *sensorId, value, and timestamp* to the message broker queue *sensor_data*. This data is subscribed and processed by the CEP engine and published back to the message broker, which delivers it to the smart contract. In this way, the sensors enable secure and realistic data flow.

For virtual IoT sensors to be replaced with real IoT sensors in a real deployment, the latter must hence support the above protocols (mTLS and AMQP) as well as X.509 for credentials checking.

6.3 Message Broker: RabbitMQ

6.3.1 Why RabbitMQ?

During the selection of a message broker for our sample implementation of the architecture, several platforms were evaluated, some of which support common messaging protocols (e.g., MQTT, AMQP) discussed in Chapter 3, including Apache Kafka[7], ActiveMQ[8], and RabbitMQ[9]. To narrow the selection, Apache Kafka and RabbitMQ were examined in detail, as these two brokers are discussed and evaluated in several academic and technical reviews [35, 118, 122]. After this evaluation, RabbitMQ was selected for the following reasons:

Kafka is designed for high-throughput data streaming and is open source. However, it does not support the SymboleoAC architecture requirements:

- Granular per-message routing for roles is not supported by Kafka;
- Kafka clients are heavy and not suitable for small IoT devices;
- Kafka does not support MQTT or AMQP natively; it uses its own protocol and requires a bridging component to interoperate with these standard messaging protocols;

[6] https://datatracker.ietf.org/doc/html/rfc5280
[7] https://kafka.apache.org/
[8] https://activemq.apache.org/
[9] https://www.rabbitmq.com/

```
  UW PICO 5.09                    File: /usr/local/etc/rabbitmq/rabbitmq.conf

# TLS Listener
listeners.ssl.default = 5671
ssl_options.cacertfile = /Users/sfuhaid/RunBlockchain/symboleoAC-app/certs/ca-cert.pem
ssl_options.certfile   = /Users/sfuhaid/RunBlockchain/symboleoAC-app/certs/rabbitmq-server.crt
ssl_options.keyfile    = /Users/sfuhaid/RunBlockchain/symboleoAC-app/certs/rabbitmq-server.key
ssl_options.verify = verify_peer
ssl_options.fail_if_no_peer_cert = false

# Enable client certificate authentication
ssl_cert_login_from = common_name
auth_mechanisms.1 = EXTERNAL
auth_mechanisms.2 = PLAIN
loopback_users = none
```

Figure 6.1: RabbitMQ configuration enabling mutual TLS, server certificate verification, and client certificate-based authentication.

- Kafka does not provide built-in message routing.

On the other hand, **RabbitMQ** is free and open source, and meets our SymboleoAC architecture requirements:

- Supports both MQTT and AMQP messaging protocols;
- Provides built-in message routing, enabling per-role message subscription;
- Lightweight and suitable for small IoT devices;
- Supports multiple queue policies (e.g., durable, exclusive, auto-delete).

Thus, **RabbitMQ** is used in our case studies as a messaging middleware to support secure communication among IoT devices, the CEP, the Node.js application, and the generated smart contract (SymboleoAC2SC).

6.3.2 Mutual TLS Configuration and Authentication

For mutual TLS (mTLS) communication, RabbitMQ is set up as a stand-alone AMQP server, so all clients (e.g., roles, IoT devices, CEP, listeners) are required to publish or subscribe over encrypted channels. Thus, *both* the server and the client are required to authenticate each other before any message exchange occurs. The configuration used for RabbitMQ to support mTLS is shown in Figure 6.1.

Additionally, RabbitMQ is set up with a server certificate (X.509) signed and issued by a reliable Certificate Authority (CA) by Fabric CA, allowing clients to authenticate the message broker before publishing or subscribing. The RabbitMQ server runs on port 5671 and supports two security steps (see Figure 6.1):

1. **Authentication** via the `EXTERNAL` mechanism used to validate the client's certificate;

2. **Authorization** via the `PLAIN` mechanism using RabbitMQ's permission system, which determines whether the authenticated identity is allowed to read from or write to specific queues.

Each mTLS handshake performs two identity checks:

1. **Client authentication:** RabbitMQ verifies the client's X.509 certificate (e.g., IoT sensor, role subscriber, CEP engine, and listeners).
2. **Server authentication:** The client verifies the RabbitMQ server certificate to ensure it is communicating with a trusted message broker.

For IoT devices and other off-chain clients, authentication occurs through certificate validation against identities stored in the Hyperledger Fabric *wallet*. If a client identity is missing or invalid, communication with RabbitMQ is rejected. This certificate-based authentication ensures that only trusted CPS components can publish sensor readings or subscribe to alert and notification queues.

6.3.3 Authorization – Clients Permission via RabbitMQ

As explained earlier, two security steps are enforced by RabbitMQ. First, it checks the client's certificate, and second, it verifies whether the authenticated client is allowed to read from or write to specific queues (subscribe/publish). These permissions can be defined using the `rabbitmqctl` tool via the `set_permissions` command in the RabbitMQ terminal. Listing 6.1 shows how to use `rabbitmqctl` commands to assign `queue` and *exchange level* permissions to the virtual IoT sensors, the CEP engine, and the Fabric subscriber. In this sample configuration:

- **IoT sensors** (e.g., `temperature_sensor` and `humidity_sensor`) are allowed to **write only** to the `sensor_data` queue.
- **CEP engine** (`cep_bridge`) is allowed to **read** from `sensor_data` and **write** to `alerts`.
- **Fabric smartcontract listener** (`fabric_listener`) is allowed to **read only** from the `alerts` exchange.

```
# Sensors: write ONLY to sensor_data
rabbitmqctl set_permissions -p / temperature_sensor_tempRule "" "^(sensor_data)$" ""

rabbitmqctl set_permissions -p / humidity_sensor_humidityRule "" "^(sensor_data)$" ""

# CEP: read sensor_data, write alerts
rabbitmqctl set_permissions -p / cep_bridge "^(sensor_data|alerts)$" "^(sensor_data|alerts)$" ""

# Fabric subscriber: read alerts ONLY
rabbitmqctl set_permissions -p / fabric_listener "^(alerts)$" "" ""
```

Listing 6.1: RabbitMQ permissions for `sensor_data`, `alerts`, and role specific users

6.3.4 Authorization – SymboleoAC Policy in RabbitMQ

To subscribe to notifications generated by the smart contract, we generate a `rules.json` file that retrieves the list of roles associated with each SymboleoAC contract specification and dynamically create a dedicated RabbitMQ queue for each of them. RabbitMQ then checks whether a role subscriber has the required permissions to access its corresponding queue by inspecting the notification event payload, which contains the list of authorized roles derived from the SymboleoAC policy specification. This payload cannot be tampered with by off-chain clients because it is stored on the blockchain, which is immutable.

For notifications, we adopt a *per-role queue* design rather than a shared queue with filtering to enforce several security and architectural principles:

- **Confidentiality**: messages are isolated per role, preventing any unauthorized visibility across subscribers.
- **Least privilege**: each role receives only the notifications it is authorized to access, without requiring additional filtering logic.
- **Scalability**: queues can be independently scaled per role or user.
- **Auditing and monitoring**: per-role queues enable clear and traceable message flows for accountability and compliance.

6.3.5 Queuing

RabbitMQ allows the definition of multiple queues to support the workflow between IoT devices, the CEP engine, and the smart contract listener (Fabric listener). We configure each queue differently based on the needs of the SymboleoAC architecture. Three categories have been designed for our implementation: (1) `sensor_data`, (2) `alerts`, and (3) `per-role notification queues`.

1. `sensor_data` Queue (durable, point-to-point) Virtual IoT devices publish raw IoT measurements (e.g., temperature, humidity, and so on) to this queue. The queue is declared as shown in Listing 6.2. The `durable`[10] choice means the queue does not need to be redeclared and will not be deleted if the message broker is restarted. Also, the queue will be received by only one subscriber, i.e., the CEP engine in our case.

```
1 await channel.assertQueue('sensor_data', { durable: true });
```

Listing 6.2: A queue `sensor_data`

2. `alerts` Exchange and Queue (durable, fanout) The CEP engine publishes to the `alerts` exchange using the `fanout` type, as shown in Listing 6.3. The `fanout`[11] option is a

[10] https://www.rabbitmq.com/docs/queues
[11] https://www.rabbitmq.com/docs/exchanges

type of exchange that creates temporary queues and broadcasts a message to all of them; it does not require a routing key. Once a new alert is received, the Node.js subscriber (i.e., listener) consumes the alert payload, which contains `sensorId`, `avgValue`, `sensorTime`, and `alertTime`. The `sensorId` is then used to retrieve contract information from `rules.json`, which returns an array containing `contractId`, `chaincodeFunction`, and `chaincodeName`. Using this information, the listener dynamically invokes the corresponding smart contract transaction after being verified as an authenticated listener.

```
await channel.assertExchange('alerts', 'fanout', { durable: true });
```

Listing 6.3: An exchange `alerts`

As shown in Listing 6.4, we do not need to specify a queue name as no routing keys are required, i.e., every listener receives all alerts from the CEP engine.

```
const { queue } = await channel.assertQueue('', {
    exclusive: true,
    autoDelete: false,
    durable: false
});
await channel.bindQueue(queue, 'alerts', '');
```

Listing 6.4: An exchange `alerts` with random queue

3. Per-Role Notification Queues (non-durable, direct routing) When the generated smart contract emits a notification events using Fabric `setEvent()` that include an embedded list of authorized roles, the Node.js publisher (i.e., listener) forwards it to the `eventExchange` using a `direct` exchange. The exchange is declared as shown in Listing 6.5. The `direct`[12] exchange type means that the message will be routed to the queue whose routing key exactly matches the message's routing key.

```
await channel.assertExchange('eventExchange', 'direct', { durable: false });
```

Listing 6.5: An exchange `eventExchange`

For each SymboleoAC contract role (e.g., a buyer or a seller), a dedicated queue is created dynamically. For the per-role queues, we select the `non-durable` option, which means the queues do not need to persist through broker restarts and exist only for the duration of the session, and once it is consumed by the subscriber, it is deleted for security purposes. Using `direct` routing ensures that each queue receives only the messages whose routing key matches `role.<role>`, as shown in Listing 6.6.

```
const queueName = 'queue.role.${role}';
await channel.assertQueue(queueName, { durable: false });
await channel.bindQueue(queueName, exchangeName, 'role.${role}');
```

Listing 6.6: A per-role queue `queue.role.${role}` and binding via `role.${role}`

[12] https://www.rabbitmq.com/docs/exchanges

Only authorized roles as identified by the `permissionValid()` function in the SymboleoACJS library and encoded in the notification event payload are permitted to subscribe to their respective queues under this per-role queuing approach, which is consistent with the SymboleoAC access control model.

6.4 CEP Engine: Esper

6.4.1 Why Esper?

Complex Event Processing (CEP) plays an important role in the SymboleoAC architecture (Figure 5.1) by analyzing, filtering, and aggregating streams of real-time raw IoT data before it reaches the smart contract. In our architecture, CEP is used to detect threshold violations across multiple events in a continuous stream of IoT data. Without CEP, the smart contract would be forced to process large volumes of sensor readings, which is costly, inefficient, and unsuitable for a blockchain environment. Placing the CEP in the middle allows simple raw IoT data to be transformed into more abstract events that the smart contract can understand and consume. Additionally, from a security perspective, the CEP can also act as a gatekeeper that reduces the risk of spamming the smart contract with malformed or corrupted data. It also reduces the risk of sensor spoofing, as anomalies can be detected at the CEP level before triggering a smart contract, for example by verifying the IoT signature.

For selecting a CEP engine, as discussed earlier in Chapter 2, CEP engines use an Event Pattern Language (EPL), and we required one that supports stream types suitable for IoT data. We narrowed our selection to two open-source platforms that support stream-oriented EPLs: Apache Flink SQL[13] and Esper EPL[14].

Both support EPL as a query language, which is an SQL-like language. However, we decided to use Esper because it is lightweight compared to Apache Flink, the latter being cluster-based and requiring additional dependencies to run.

6.4.2 Esper EPL Rules

An EPL statement has the following structure:

```
select <fields> from <Event>(<condition>).win:<window> having <aggregate>
```

Each rule describes:

- A **condition** that filters incoming sensor events (e.g., `value > 30`);
- A **window** defining how many events or how much time should be considered (e.g., `win:length(3)` or `win:time(10 sec)`);

[13] https://nightlies.apache.org/flink/flink-docs-master/docs/libs/cep/
[14] https://www.espertech.com/

- A **having clause** that evaluates aggregated results (e.g., average, count);
- A **select field** that specifies one or more fields to include/return in the output alert.

Because of this, EPL is particularly effective in IoT environments, where violations are often defined over multiple readings rather than single events. Examples include "temperature exceeds 25°C three times in a row" or "humidity average drops below 40% over a 10-second window," and so on. Listing 6.7 shows an example of an EPL rule, which is generated by SymboleoAC2SC, for temperature sensor readings that triggers a threshold violation when the temperature exceeds 25°C and at least one matching event occurs within a 10-minute sliding window.

```
select
    sensorId,
    sensorTimestamp,
    count(*) as cnt,
    avg(value) as avgValue
from
    SensorEvents(value > 25).win:time(10 min)
having
    count(*) >= 1
```

Listing 6.7: An example of generated EPL statement for `Temperature IoT device` from SymboleoAC2SC's `rules.json`

All IoT information specified in the Symboleo specification under the domain type Datatransfer is extracted, generated with the SymboleoAC2SC, and stored in `rules.json` within the SymboleoAC API. This makes the information difficult to tamper with because the original source is recorded on the immutable blockchain. Listing 6.8 shows a snippet of the generated `rules.json` file, highlighting the fields that will be used to dynamically generate EPL rules for each sensor for a contract instance and to invoke the corresponding smart contract transactions accordingly.

```
{
  "rules": [
    {
      "id": "tempRule",
      "contractId": "MeatSale_202581716",
      "chaincodeName": "meatsale",
      "eventType": "SensorEvent",
      "sensorType": "temperature",
      "sensorId": "temperature_sensor_tempRule",
      "condition": "value > 25",
      "window": "time(10 min)",
      "having": "count(*) >= 1",
      "select": "sensorId, sensorTimestamp, count(*) as cnt, avg(value) as avgValue",
      "chaincodeFunction": "trigger_temperatureAlert"
    }
    // ... other rules omitted
  ]
}
```

Listing 6.8: Example rule from generated `rules.json` file (temperature rule)

6.4.3 CEP Integration in SymboleoAC

In this SymboleoAC architecture implementation, the CEP engine is operationalized as a standalone Java process program called `EsperBridge.java`. The bridge conducts four key functions:

1. **Loading DataTransfer monitoring rules**: As stated earlier, a `rules.json` file is created by SymboleoAC2SC to configure the CEP with the rule identifier, condition, window, selection fields, and alert logic for each sensor type, as well as corresponding smart contract transaction that will change contract state. These rules are dynamically compiled into EPL statements using the Esper compiler as shown in Listing 6.9.

```
// Deploy EPL rules dynamically from rules.json
for (Map<String, Object> rule : rules) {
    String condition = (String) rule.get("condition");
    String window    = (String) rule.get("window");
    String having    = (String) rule.get("having");
    String select    = (String) rule.get("select");

    String epl = String.format(
        "select %s from SensorEvents(%s).win:%s having %s",
        select, condition, window, having
    );

    EPCompiled compiled =
        compiler.compile(epl, cargs);

    runtime.getDeploymentService()
           .deploy(compiled);

    // ... omitted code
}
```

Listing 6.9: A snippet illustrating the compilation and deployment of EPL rules inside `EsperBridge.java`

2. **Mutual TLS – Receiving IoT readings from RabbitMQ securely**: The CEP engine uses mutual TLS authentication (`EXTERNAL`, see Figure 6.1) to listen on the `sensor_data` queue. Thus, before any data is consumed, two validation checks are conducted:

   - The CEP engine verifies the RabbitMQ server certificate, and RabbitMQ verifies the CEP client certificate. Both certificates must be issued by the same trusted Certificate Authority (CA), ensuring that each side recognizes and trusts the other. See Listings 6.10 and 6.11.
   - The CEP engine checks whether the IoT device has a valid certificate issued by a trusted authority. The CEP rejects any data that is not authenticated. See Listing 6.12.

```
/* --- Load CA certificate and build trust store (server authentication) --- */
CertificateFactory cf = CertificateFactory.getInstance("X.509");
X509Certificate caCert;
try (FileInputStream fis = new FileInputStream(CA_PATH.toFile())) {
    caCert = (X509Certificate) cf.generateCertificate(fis);
}

KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
trustStore.load(null, null);
```

```
trustStore.setCertificateEntry("fabric-ca", caCert);

TrustManagerFactory tmf =
    TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmf.init(trustStore);

/* Enable hostname verification to ensure the server certificate matches */
factory.enableHostnameVerification();
```

Listing 6.10: RabbitMQ server authentication: CEP verifies the message broker's X.509 certificate

```
/* --- Load CEP client certificate and private key from PKCS#12 keystore --- */
KeyStore keyStore = KeyStore.getInstance("PKCS12");
try (FileInputStream ksFile = new FileInputStream(P12_PATH.toFile())) {
    keyStore.load(ksFile, P12_PASSWORD);
}

KeyManagerFactory kmf =
    KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
kmf.init(keyStore, P12_PASSWORD);

/* Use EXTERNAL mechanism for mutual TLS authentication */
factory.useSslProtocol(sslContext);
factory.setSaslConfig(DefaultSaslConfig.EXTERNAL);
```

Listing 6.11: CEP client authentication: RabbitMQ verifies the CEP engine certificate

```
/* --- Step 1: ensure the sensor is registered in the Fabric wallet --- */
if (!WalletUtil.isSensorRegistered(sensorId)) {
    System.err.println(" Unauthorized sensorId (not in wallet): " + sensorId);
    return;
}

/* --- Step 2: verify the sensor certificate CN matches the sensorId --- */
String certPem = WalletUtil.getSensorCert(sensorId);
if (!CertificateUtil.verifySensorBinding(certPem, sensorId)) {
    System.err.println(" Certificate CN mismatch for sensorId: " + sensorId);
    return;
}

/* --- Step 3: forward validated event to Esper --- */
runtime.getEventService().sendEventBean(
    new SensorEvents(sensorId, value, sensorTimestamp),
    "SensorEvents"
);
```

Listing 6.12: IoT device identity validation: wallet check & certificate Common Name (CN) binding

3. **Evaluating EPL rules and generating complex events**: Every time an EPL rule is triggered, Esper generates a complex event that summarizes the condition or violation found in several raw readings. Timestamps and aggregated values are included in the event payload as shown in Listing 6.13.

```
String alert = String.format(
    "ALERT %s: %s, alertTimestamp=%s}",
    rule.get("id"),
    newData[0].getUnderlying(),
    alertTimestamp
);
```

Listing 6.13: Constructing an alert message with timestamp in `EsperBridge.java`

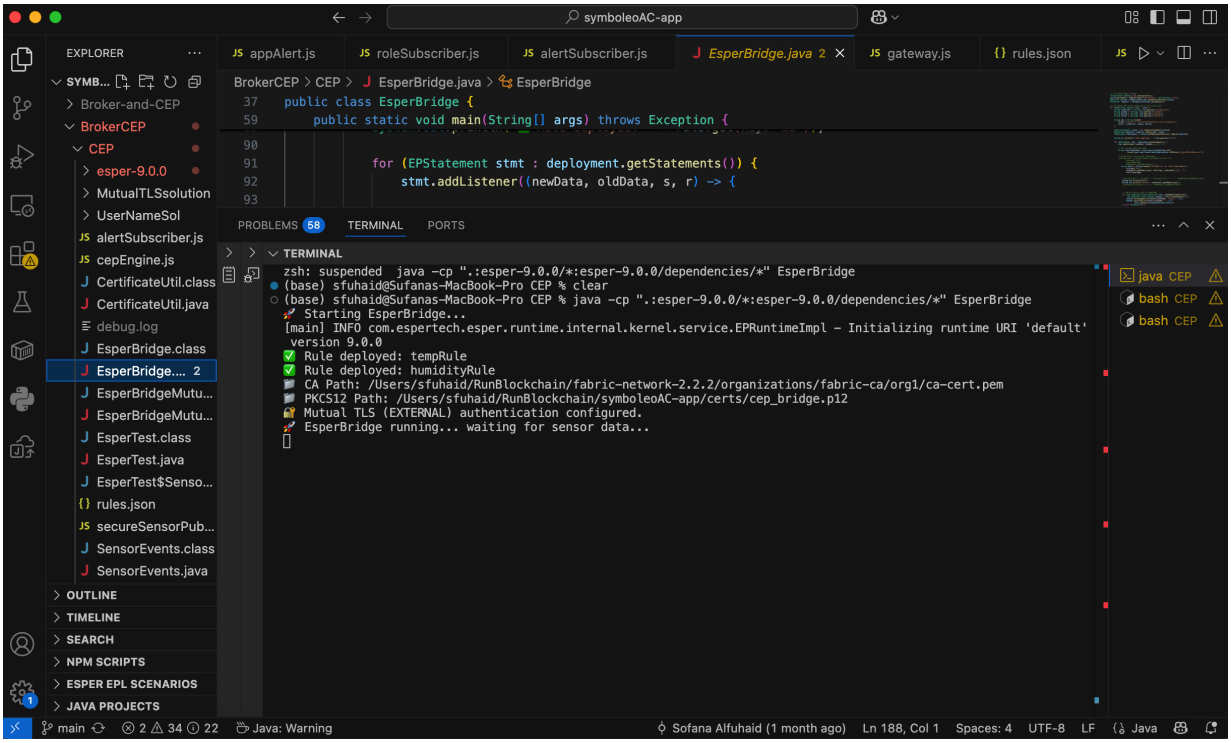


Figure 6.2: EsperBridge runtime showing successful deployment of CEP rules (`tempRule` and `humidityRule`) and awaiting incoming sensor readings over the `sensor_data` queue.

4. **Publishing alerts back to RabbitMQ**: The complex event is then sent via a `fanout` routing pattern to the `alerts` exchange (Listing 6.14).

```
channel.exchangeDeclare("alerts", "fanout", true);
channel.basicPublish("alerts", "", null, alert.getBytes());
```

Listing 6.14: Publishing a CEP-generated alert to the `alerts` exchange (fanout) in `EsperBridge.java`

At this point of the example, as shown in Figure 6.2, the Esper CEP engine is operational and has been successfully integrated with RabbitMQ. Two rules have been deployed correctly, and the engine is listening securely via mutual TLS.

6.5 Node.js Application

The core off-chain run-time that connects Hyperledger Fabric and generated smart contract (from SymboleoAC2SC), RabbitMQ, Esper CEP, and IoT devices is the SymboleoAC API that targets Node.js run-time. The SymboleoAC API is implemented primarily to enable communication with external services. It was developed manually and is designed to operate with any smart contract generated by SymboleoAC2SC through the Fabric

API, enabling dynamic interaction across different contract instances, as demonstrated in Chapter 10. It uses a `rules.json` file, which is automatically generated from the contract specification.

It provides all middleware logic (i.e., the required JavaScript modules and functions to interact with Hyperledger Fabric through the Fabric SDK) for client enrollment and registration (roles, IoT devices, CEP, RabbitMQ, and event listeners) through a Fabric Certificate Authority (CA), authentication, transaction submission, smart contract event consumption, publishing and subscribing to RabbitMQ queues, and dynamic run-time enforcement of SymboleoAC policies.

6.5.1 Gateway Connection to Hyperledger Fabric

The Hyperledger Fabric SDK for Node.js is used by the SymboleoAC application to communicate with the blockchain network. It creates a secure connection to the blockchain network, loads the organization's connection profile, and confirms the user's identification from the wallet. It returns a contract object connected to the particular chaincode and channel for the contract instance after authentication.

For example, `gateway.js` in SymboleoAC API provides a contract object connected to the `contract` chaincode deployed on the Fabric channel called `mychannel` when interacting with one of our case study, e.g., the *MeatSale* contract instance. This allows the SymboleoAC API to submit transactions (e.g., `trigger_paid`) or evaluate queries securely and in accordance with the access control rules enforced by SymboleoAC.

6.5.2 Identity Management (Wallet)

All the identities of off-chain entities (i.e., roles, IoT devices, listeners), except the message broker and the CEP engine, are stored in a FileSystem *Wallet*. This wallet acts as the digital identity registry in Figure 5.1. Each identity is represented by an `*.id` file that contains the X.509 certificate and private key issued by the Fabric Certificate Authority. We use a separate folder (named `cert`) for message broker (RabbitMQ) and the CEP (Esper) certificates, distinct from the Fabric wallet used for users, admins, sensors, and listeners. RabbitMQ and Esper require conventional Privacy Enhanced Mail (PEM) files[15] (i.e., `.crt` and `.key`) because they do not understand the JSON based wallet identity format used by Hyperledger Fabric (i.e., `*.id` files). These identities are used for:

- Authenticating off-chain components through certificates.
- Imposing mutual TLS communication through certificates.
- Signing smart contract transactions.
- Validating roles and permissions embedded in SymboleoAC policies.

[15] https://www.rfc-editor.org/rfc/rfc1422

There are several enrollment scripts available in SymboleoAC's application and API[16], including `EnrollSensors.js`, `EnrollCEPServer.js`, `EnrollRabbitMQ.js`, and `EnrollRoles.js`, each of which is in charge of registering and enrolling the relevant user or component with the Fabric CA.

6.5.3 Broker and CEP Integration

The `BrokerCEP` directory contains the logic for message-driven interaction with RabbitMQ and the CEP engine. To name a few:

- `rabitMQ-Publish.js` implements the publisher for role-based messaging.
- `roleSubscriber.js` and `alertSubscriber.js` implement secure subscribers using mutual TLS.
- `sensorPublisher.js` publishes signed IoT readings to the `sensor_data` queue.
- `eventListeners.js` listens to Fabric chaincode events and forwards them to RabbitMQ queues that correspond to authorized roles.

Collectively, the SymboleoAC Node.js application and its API provide secure and event-driven communication between the run-time off-chain and on-chain components of SymboleoAC architecture. All files are available online.[17]

6.6 Blockchain Network: Hyperledger Fabric

Hyperledger Fabric [123], a permissioned distributed ledger technology intended for enterprise grade applications, serves as the foundation for the blockchain layer of the SymboleoAC architecture. The channel configuration files, Membership Service Provider (MSP) directories, peer and orderer certificates, and connection profiles (e.g., `connection-org1.json`) are all essential configuration artifacts needed for safe on-chain execution and are included in the Fabric network. These files specify the identities, policies, and communication parameters for every on-chain participants and are produced by Fabric's configuration tools (`cryptogen`, `configtxgen`).

Using our SymboleoAC2SC compiler, the executable smart contract is generated from the SymboleoAC specification and deployed as a chaincode package on this Fabric network. In accordance with Fabric's chaincode lifecycle, each contract instance is installed on peers of the relevant organization and authorized. After the contract is committed to the channel, it may be securely invoked via the Node.js gateway, guaranteeing that all contract-related transactions are executed in an authenticated, authorized, and verifiable manner. The created Hyperledger Fabric network is available online.[18]

[16] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-Application-API
[17] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-Application-API
[18] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-HyperledgerFabric-Test-Netwrok

6.7 Conclusion

The SymboleoAC API, including the Node.js application, offers the core middleware layer that operationalizes the entire SymboleoAC architecture by safely integrating the on-chain run-time layer, including the generated smart contract, with all off-chain components, including the CEP engine, IoT devices, and the message broker. The application facilitates end-to-end authenticated communication, real-time event handling, and secure transaction invocation through its Fabric gateway, wallet management, enrollment tools, event listeners, and publishers/subscribers.

In accordance with the SymboleoAC architecture, the API ensures that IoT readings are verified, processed through the CEP, routed via the message broker, and finally reflected in the blockchain and smart contract state. Additionally, the SymboleoAC API demonstrates the practical execution backbone of Cyber-Physical Smart Contracts (CPSCs), showing how formal contract specifications can be implemented and enforced in the real world by integrating these independent technologies into a unified run-time system.

Chapter 7

SymboleoAC Ontology – Access Control Model for Symboleo

This chapter proposes a new role-based access control model for Cyber-Physical Smart Contracts (CPSCs), treating all contract elements as resources and ensuring regulated access by designated parties. The access control model extends the Symboleo ontology for legal contracts with new modeling concepts inspired by Role-Based Access Control (RBAC), tailored for the legal contract domain, resulting in the SymboleoAC ontology (1 in Figure 1.1). Specifically, we: (i) model a set of access control concepts, including resource, access rule, and access policy, thereby extending the Symboleo ontology, (ii) define controller rules that specify who can authorize access to each resource, and (iii) present pre-authorization rules that specify who has access to what.

7.1 Introduction

We are interested in making legal contract executions secure by including safeguards against physical and social attacks. For the meat sale contract example, such attacks include stealing some of the meat during its transportation or damaging its quality, both *physical* attacks concerning unauthorized access to the meat [102]. Other attack examples also include having an intruder who poses as an assessor and sends a report that misrepresents the quality of the meat, a *social* attack that reflects known vulnerabilities of smart contract enabled processes to social engineering [43]. In addition, we consider *privacy* concerns, such as what information is available to each participant in a contract execution, particularly regarding IoT generated data [65]. For the meat sale contract, this information may involve the happening of events (When was the meat delivered?) and data generated by IoT devices (What was the temperature in the container throughout the trip?). However, the scope of our interests does not include cyber attacks that exploit vulnerabilities in blockchain technology or IoT devices.

Our proposed solution consists of an access control model for legal contracts that views all things that participate in a contract execution as *resources*: contracting parties, assets,

the contract itself, obligations and powers, as well as all information generated during contract execution (events, state transitions for legal concepts, other data, such as meat container temperature). Every resource has associated operations through which the state of the resource can be changed during contract execution. Moreover, every resource has one or more parties as *controller* who collectively decide who else has access to the operations and information of the resource. For example, an obligation has as controller its *debtor*, i.e., the party who is responsible for fulfilling the obligation. On the other hand, the *creditor* of an obligation, i.e., the party who stands to benefit from its fulfillment, has access to the current state of the obligation. In other words, the creditor's right to see the obligation satisfied includes the right to know what the current state of the obligation is. Access rules such as the above are derived from the semantics of the legal concepts of contract, obligation, and power [42].

Our security approach does not go beyond RBAC-based protection: if an attacker gains valid credentials, such as a password or private key, outside of the system, RBAC cannot prevent misuse. While some forms of social attacks can be mitigated through role restrictions and policy enforcement, others, particularly those involving credential compromise, remain outside the protective scope of our framework and are discussed among the limitations in Chapter 11. The proposed access control model is limited to the support of several core functions identified in the NIST Cybersecurity Framework 2.0 [93], particularly the *Govern*, *Identify*, and *Protect* functions, through identity management, authentication, authorization, and policy enforcement.

7.2 Access Control Ontology

Our proposed access control model, shown in Figure 7.1, is based on the RBAC security mechanism [32, 108] designed to regulate and manage access to resources within a software system, ensuring that only authorized parties can interact with resources.

The RBAC ontology consists of **Permissions** to **Subjects** (aka Roles), i.e., system users, enabling them to execute **Actions** (aka Operations in SymboleoAC), such as read, write, etc., on **Resources** (also Resources), such as classes, files, etc. A role can only apply operations on a resource if it has permission assigned by an access rule.

7.2.1 Policies and Rules

An **Access control policy** (aka Policy) contains Rules that must hold in priority over other rules). For example, a policy by a Food Industry regulator may be that the buyer has full access to all relevant information about the transportation of the meat, whereas only the assessor of the transaction has access to meat quality/quantity attributes. Policies define constraints on what is (dis-)allowed for access rules. Accordingly, they need to be checked against the initial set of access rules (aka *preauthorized access rules*) as well as after every incremental insertion or deletion of access rules. This access control framework ensures that only authorized roles can perform operations on a resource, safeguarding the

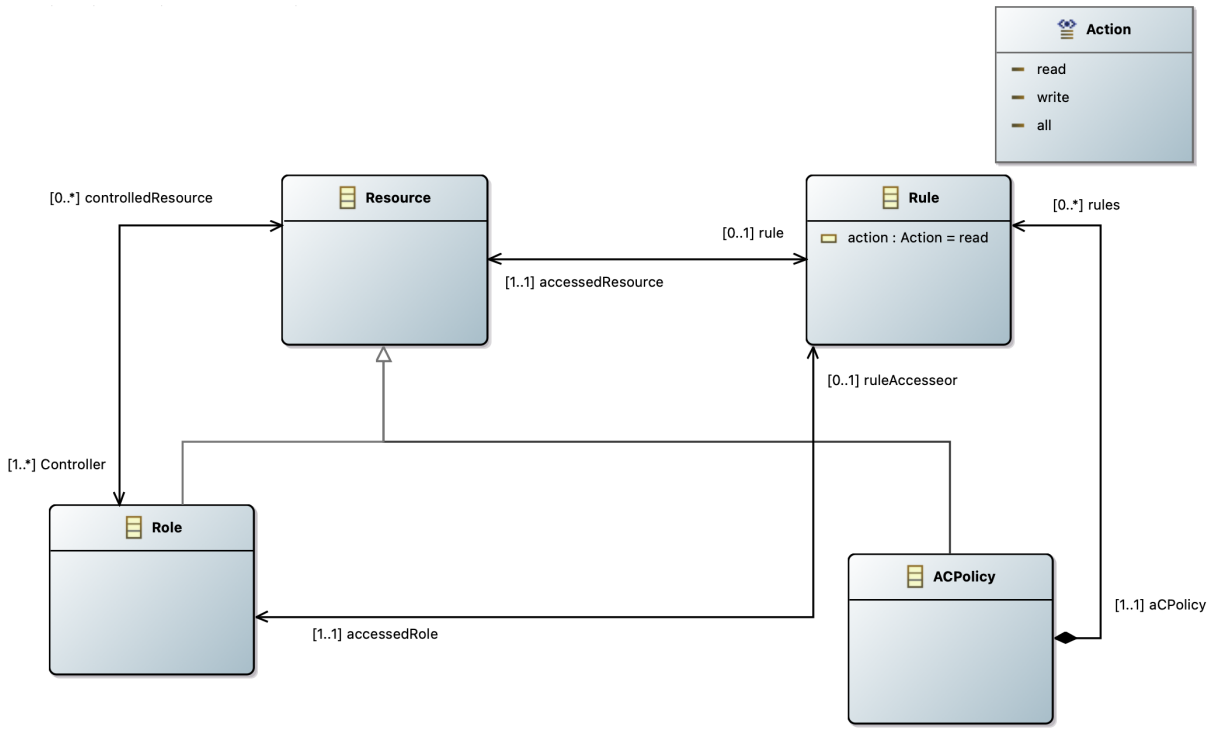


Figure 7.1: Access control ontology.

system's integrity and security. By delineating access rights, the flow of information and activities within systems can be managed and regulated, reducing the risk of unauthorized access or misuse.

In traditional RBAC proposals, the assigned permissions are maintained by the security engineer/administrator or anyone in a similar position. In our case, access rules are embedded in smart contract code to only recognize as legitimate permissible access to operations and information about a contract execution.

7.2.2 Access Control Operations

For each of the access control classes in Figure 7.1, we used Umple [38, 74] to automatically generate all the methods and attributes required for the associations connecting these classes. For example, a class ACPolicy contains a collection of Rules, and Umple generates all the methods required to access and modify related objects. These methods include *addRule* and *removeRule*, used for adding or removing rules to/from the policy. However, some utility functions at run-time are not addressed by such automatically generated methods. Custom access control methods were added to the class Role to add additional features, allowing for greater flexibility and customization in the design of the access control policy. Details of some custom access control operation at run-time are shown in Table 7.1.

Table 7.1: Access control execution-time operations for Roles

Operation	Description
identify(r)	Takes as argument a role r and issue an ID for it.
authenticate(r)	Authenticates a role r through an issued ID.
authorize(r, re)	Grants to a role r full authorization to access operations and attributes of that resource *re*.
authorize(r, re, att, R/W/All)	Provides fine-grained authorization by specifying particular attributes *att* and access types *(R/W/All)* that a role r is authorized to access or modify on a given resource *re*.
authorize(r, re, op)	Grants the role r permission to execute specific operations *op* on a resource *re*.
deauthorize(r, re)	Revokes all authorization previously granted to the specified role r on a resource *re*.
deauthorize(r, re, att, R/W/All)	Revokes previously granted authorizations by specifying particular attributes *att* and access types *(R/W/All)* that a role r is no longer authorized to access or modify on a given resource *re*.
deauthorize(r, re, op)	Revokes the authorization for the role r to execute specific operations *op* on a resource *re*.

7.3 Overview of Access Control Integration Approach

In this section, we introduce our approach to integrate the access control ontology with the existing SYMBOLEO ontology and artifacts, following the process shown in Figure 7.2.

The proposed approach for generating secure smart contracts consists of two main components:

1. **SYMBOLEOAC access control model:** This step involves integrating the RBAC-inspired access control ontology shown in Figure 7.1 with the main SYMBOLEO ontology from Figure 2.3 [92, 109]. The objective is to produce the SYMBOLEOACJS library, which will be utilized by the SYMBOLEOAC2SC compiler to generate secure smart contracts (Chapter 9).

2. **SYMBOLEO IDE and code generation – SYMBOLEOAC2SC:** This phase entails updating the Xtext grammar and validation rules of the original SYMBOLEO IDE with SYMBOLEOAC concepts. This enhancement enables developers to specify legal contracts along with their access control rules and domain restrictions more effectively. Moreover, the SYMBOLEO2SC tool, which was originally designed to generate smart contracts from contract specifications, has been extended to incorporate the SYMBOLEOACJS library and its concepts. Additionally, the improved tool now supports SYMBOLEOAC controller rules and pre-authorization access rules (see Section 7.6), resulting in the full SYMBOLEOAC2SC tool (Chapter 9). These

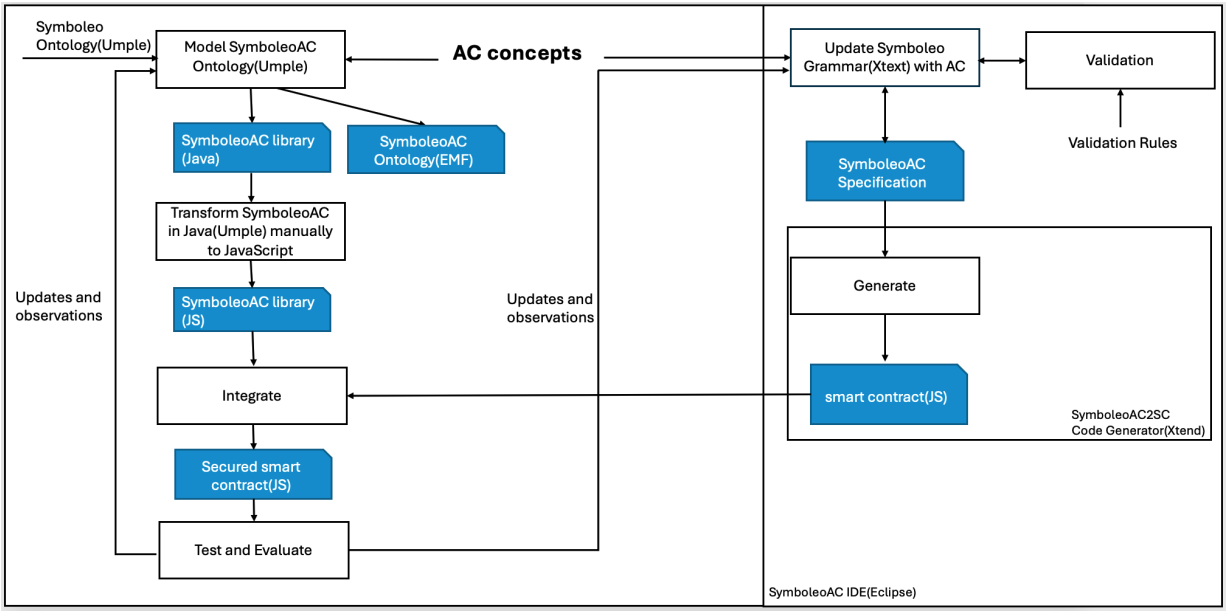


Figure 7.2: Approach taken for integrating AC concepts into SYMBOLEO.

updates facilitate the generation of secure smart contracts capable of safeguarding their resources from unauthorized access during run-time (Chapter 9).

7.4 SYMBOLEOAC

In this section, we integrate the SYMBOLEO ontology [90, 109] with the RBAC ontology composed of five concepts (Policy, Rule, Resource, and Role from Figure 7.1, and Operation from Section 7.2.2), resulting in the SYMBOLEOAC ontology depicted in Figure 7.3.

7.4.1 SYMBOLEO-RBAC Integration

The integration is founded on a simple principle: The concepts in the core SYMBOLEO ontology (except Party, Situation, TimePoint and TimeInterval) – shaded in yellow in Figure 7.3 – are resources and can have associated access rules. This means that an Obligation is a Resource, and Operations on it, such as enacting that Obligation, can be performed only by a Role who has access to such enact Operation. In other words, based on our running example, if the smart contract receives notice of meat delivery, it checks that the sender is the seller. If not, this delivery is invalid.

A Role is a Resource too. For example, for a TransportCo role, the seller needs read access to the profile and reviews of TransportCo candidates. So is the Asset meat, since the Assessor who checks the quantity and quality of the meat must have physical access to it for inspection. The Assessor also needs write access to the two Attributes (qualityFound and quantityFound), which no one else should have. All these access Rules help make a contract execution meaningful and valid.

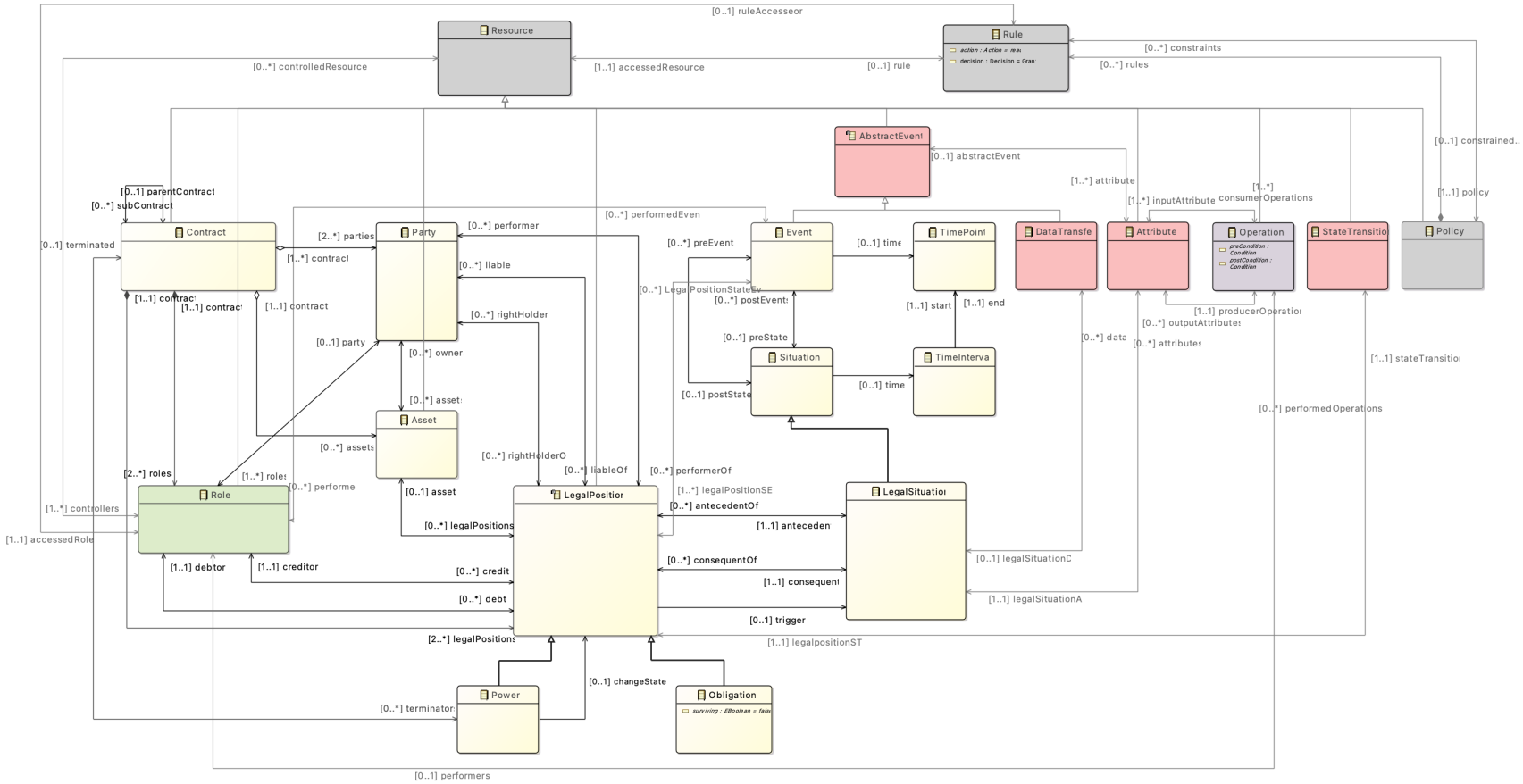


Figure 7.3: SymboleoAC ontology, with the original Symboleo ontology (in yellow), access control ontology (in blue), merged classes from both ontologies (in green and purple), and newly added classes as resources (in red).

To be able to talk about access to information about some of the SymboleoAC resources, we also need two more classes, in addition to the class Event that is already part of the Symboleo core ontology:

- Class StateTransition, whose instances are collections of tuples (fromState, toState, and timePoint) and can be specialized to a particular obligation or power, e.g., Odelivery .
- Class DataTransfer, whose instances are data generated by IoT devices, and can be specialized, for example, to TempLocTime (temperature, location, time) for the data generated during the enactment of Odelivery.

The RBAC ontology class Policy has particular policies as instances, such as "Buyer has read access to all data about food delivery" (assigned by, e.g., a *regulator* role), which can be represented with SymboleoAC's syntax as:

```
Grant read To buyer On temploctime by regulator
```

The above policy means: if temploctime is a **DataTransfer** (e.g., part of the Odelivery resource), then the buyer is authorized to access temploctime and its attributes, including the temperature of the meat, its timestamp, etc. It also means that there cannot be a new or existing rule that revokes read access to the buyer.

Another example: "Only the assessor has write access to the inspection quality attributes of the meat". In SymboleoAC, this is equivalent to:

```
Grant write To assessor On inspectedQuality by regulator
```

The assessor is the performer of the resource event inspectedQuality and, by default, has write access to its attributes, i.e., qualityFound and quantityFound. If this rule is defined as a policy, other access rules must comply with that policy. In other words, other roles such as the seller (controller of the asset meat) or TransportCo, will have their write access to these attributes revoked as shown below.

```
Revoke write To seller On inspectedQuality by regulator
Revoke write To transportCo On inspectedQuality by regulator
```

Finally, for uniformity purposes, we also treat Attribute as a resource class with two associated operations, Read and Write. This addition allows us to deal with Assessor access needs, as discussed earlier. In addition, we treat Operation as a resource class as well with preconditions and postconditions, along with two associations: inputAttributes (parameters) and outputAttributes (return values).

Figure 7.3 presents the SymboleoAC ontology. For simplicity, it only shows some of the attributes. The root ontological element is Resource, with access-protected classes

as subclasses. The merger between the SYMBOLEO ontology (shaded in yellow) and the RBAC ontology (in blue) has resulted in the merging of two classes, Role (in green) and Operation (in purple). Additionally, new classes (in red) have been added as resources, as explained earlier. New associations (colored in blue in Figure 7.3) have also been added to the classes AbstractEvent, Event, LegalPosition, LegalSituation, Operation, StateTransition, and DataTransfer.

In SYMBOLEOAC, our goal is to manage access to instances of particular resources and attributes. Therefore, we assume that access to instances of all resources and attribute values is denied by default to prevent unintended security breaches. To this end, roles must explicitly be granted access to resources, including their respective operations and attributes. However, there are some exceptions: the *controller* of a resource, by default, has full access to that resource and is allowed to change its policy and authorize other roles to use it. Additionally, if a role is *pre-authorized*, then it has access without the need to request permission. Details on the controller of each resource and pre-authorization rules are found in Section 7.6.

7.4.2 SYMBOLEOAC Ontology: New Concepts and Relationships

The main elements of the SYMBOLEOAC ontology and the additional steps performed to secure SYMBOLEO are explained below. From RBAC, we integrate:

- **Policy**: A collection of access Rules. Each policy is a constraint on access rules. Policies are enforced automatically both at resource instantiation time (i.e., pre-authorization rules) and incrementally at run-time after each addition/deletion of access rules. Policy, as a resource, has operations such as *isValid(accessRules)*, *updatePolicy(accessRules)*, and *updateRule(accessRules)*.

- **Rule**: The authorization of a Role to execute a particular Operation on a Resource (e.g., deliver meat, addPerformer for meat delivery or read/write for an attribute). Each rule specifies that a role has access to a resource, one or more operations associated with that resource, and read/write permissions for certain attributes of the resource. A rule can either **Grant** to allow access or **Revoke** to deny/remove access. The possible actions a role can perform are READ, WRITE, and ALL. SYMBOLEOAC includes CRUD operations (create, update, and delete) in the WRITE operation.

- **Resource**: An object that must be protected and to which the Rules apply. Inheritance is used to connect the class Resource from the RBAC ontology and the original SYMBOLEO classes that require restriction and protection. Each resource will be assigned a *controller* role consisting of one or more users, which may change over time.

- **Role**: An active party that interacts with protected Resources for which controllers can grant/revoke access permission.

Other extensions made to the SYMBOLEO ontology, in addition to those mentioned earlier for the purpose of protecting access to resources, include new associations:

- SymboleoAC already includes associations between the classes Party and LegalPosition (performer, rightHolder, and liable), depending on the role a party plays in the legal position [91]. The performer is the party performing legal positions, akin to execution in conventional access control. In SymboleoAC, we have added performer relationships to the Event and Operation classes, as we must also restrict who can generate events and perform operations.

- Some resources, such as obligations and powers, comprise informational components sourced from other resources – specifically, events that occur during the fulfillment of legal positions, along with associated data, attributes, and state transitions. In SymboleoAC, and from a security standpoint, we address this aspect of LegalPosition by establishing generic relationships with legal situations (conditions) that encompass Event, DataTransfer, Attribute, and StateTransition. This approach ensures that the controller or performer of the corresponding legal position can access only the outcome and its timestamp, rather than the underlying resources themselves, enhancing security and privacy.

7.5 SymboleoAC Contract Specification Example

An extended version of the meat sale contract initially proposed in [90, 109] and specified in Listing 7.1 will serve as a running SymboleoAC example for illustrating access rules and generating a smart contract with access control aspects.

```
Domain meatSaleDomain
  // Controller by default is the role itself
  Seller isA Role with returnAddress: String, name: String;
  Buyer isA Role with name: String, warehouse: String;
  // thirdParty added to differentiate third party role from contracting parties
  TransportCo isA Role thirdParty with name:String;
  Assessor isA Role thirdParty with name: String;
  Regulator isA Role thirdParty with name: String;
  Storage isA Role thirdParty with name: String;
  MeatQuality isAn Enumeration(PRIME, AAA, AA, A);
  // Controller by default is the owner
  PerishableGood isAn Asset with quantity: Number, quality: MeatQuality, barcode:String, owner: Seller;
  Meat isA PerishableGood;
  // For delivered event, the controller is its performer
  Delivered isAn Event with deliveryAddress: String, delDueDate: Date, performer: TransportCo, controller: Seller;
  InspectedQuality isAn Event with Env quantityFound: Number, Env qualityFound:MeatQuality, Env barFound: String,
     performer:Assessor;
  TempLocTime isA DataTransfer with Env value: Number, Env sensorTimestamp:String, condition: String, window: String,
     count: String, controller:Seller, performer:Regulator; // Other events exist but are skipped here.
endDomain
  // ... Omitted code describing the contract signature and local definitions
Obligations
  // Controller by default is the debtor (seller) of obligation delivery
  delivery: Obligation(seller, buyer, true, WhappensBefore(delivered, delivered.delDueDate) and not Happens(
     temploctime) and delivered.deliveryAddress == buyer.warehouse) with Controller seller;
  // Controller by default is the debtor (assessor) of obligation delivery
  inspectMeat: Happens(delivered) -> Obligation(assessor, buyer, Happens(passwordNotification), Happens(
     inspectedQuality) and inspectedQuality.barFound == goods.barcode and inspectedQuality.qualityFound == goods.
     quality  and inspectedQuality.quantityFound == goods.quantity);

Powers
   // Controller by default is the creditor of power terminateContract, i.e.,buyer
    terminateContract: Happens(Violated(obligations.delivery)) -> P(buyer, seller, true, Terminated(self));

ACPolicy with Controller regulator // Controller of policy is the regulator who can override rules and pre-
     authorization rules
 Rule1: Grant read To buyer On goods.quantity by seller; // Access to specific asset attribute
 Rule2: Grant read To assessor On obligations.delivery by seller; // Access to obligation
 Rule3: Grant read To transportCo On inspectedQuality by assessor;
 Rule4: Grant read To seller On inspectedQuality by assessor;
 Rule5: Grant read To buyer On temploctime.value by regulator;
 Rule6: Grant write To assessor On inspectedQuality by regulator;
 Rule7: Grant write To transportCo On powers.suspendDelivery by seller;
```

```
38   Rule8: Grant write To transportCo On powers.resumeDelivery by seller;
39  endContract
```

Listing 7.1: Meat sale contract specification in SymboleoAC, adapted from [90,109]. Note that objects (specific events, roles, or assets) instantiating the domain classes start with a lowercase letter; see full specification online for details[1].

This contract specification has multiple roles for contractual parties (seller and buyer) and third parties (a transportation company – TransportCo, an assessor, and a regulator). The contract also includes an obligation for the seller to deliver meat (asset) of a specified quantity, quality, and temperature to the buyer, and another obligation requiring the buyer to pay a specified amount for the meat before it is shipped. Additionally, there is an obligation for the assessor to inspect the quality and quantity of the delivered meat. The occurrence of delivery, payment, and meat inspection is indicated by the events delivered, paid, and inspectedQuality, respectively. Moreover, the seller has the right to suspend their delivery obligation in case of a violation of the payment obligation.

Each of the resources (roles, assets, events, data, obligations, and powers) has information that should not be available to all roles. For example, we do not want anyone to modify the quality and quantity of meat during meat inspection. In the inspectedQuality event, the assessor is the performer (Listing 7.1, line 7) and the only role with write access to its attributes (quantityFound, qualityFound). Therefore, proper access control is added to Symboleo to manage who is entitled to do what on which resource and protect others. To this end, each of the roles (lines 3–9) can interact with different resources according to the permissions detailed in the policy (lines 30–38). A complete example is available online[1].

7.6 SymboleoAC Rules

As SymboleoAC is tailored for contract execution, it differs from other access control approaches (i.e., centralized) designed for other software. We aim to protect the execution of the contract through a distributed access control mechanism. In this context, there are two sets of SymboleoAC rules: (1) rules that determine a controller for every resource, and (2) pre-authorization rules that determine who has access to what, based on the role they play in contract execution.

7.6.1 Controller Rules

A controller is a role that sets access rules to its controlled resource for other roles. The controller can access all operations and attributes of that resource, and can authorize and deauthorize other roles to use that resource or part thereof. Controller rules determine who

[1] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-JS-Core/blob/main/MeatSale.symboleo

is the controller of every resource and they are founded on two principles: *responsibility* and *informational composition*. The controller of a legal position is the role legally responsible/liable for that legal position. The informational composition principle defines the informational parts of each obligation/power, i.e., the different kinds of information that are generated during the enactment of an obligation/power, and assigns as a controller for all of these the controller of the legal position. Applying these principles, we arrive at the following default rules for controllers, which are core to SymboleoAC and do not need to be specified explicitly in a contract:

- **Contract:** The controller of a contract is the set of contracting parties who jointly authorize permissions to contract operations and state. For example, the meat sale contract involves five roles. The seller and the buyer are the contracting parties (see Listing 7.1, lines 3–4), and the other third parties.
- **Role:** The controller of a role is the role itself (lines 3–9). In the meat sale example, for the seller role, the seller grants access to its activities such as personal profile information.
- **Asset:** Its controller is its owner. For the meat sale contract, the seller is the owner of the meatand can access its operations and attributes, and determine access permission to that asset (line 12).
- **Obligation:** The role responsible for an obligation serves as its controller. When an obligation is instantiated, the debtor is both performer and responsible (i.e., liable) for it and, therefore, is its controller. For the delivery obligation (line 22), the seller is responsible and, therefore, the controller.
- **Power:** The creditor, being a power's rightholder and performer, is its controller. For terminateContract (line 28), the buyer is the controller.
- **Policy:** Its controller is one of the contracting or third parties (line 30). The policy controller has more authority than the controllers of other resources and can override any access permission established by other controllers.
- **Event:** The controller of an event is its performer (e.g., delivered on line 15).
- **Attribute:** Its controller is the one of the containing resources.
- **Operation:** Its controller is the one of the containing resources.
- **StateTransition:** Its controller is the one of the containing obligation, power, or contract (along the partOf relationship in the ontology).
- **DataTransfer:** Its controller is the containing obligation/power (e.g., temploctime on lines 17 and 22).

7.6.2 Pre-Authorization Rules

Pre-authorization refers to automated authorizations that take place when a resource is instantiated, serving as the initialization of access rules for the resource instance. However, The policy controller can override these pre-authorization rules with a newly added policy. This ensures that the pre-authorizations align with broader governance, which is always there during the execution of a legal contract. Pre-authorization rules in SymboleoAC (with examples) cover:

- **Contract**: The contracting parties that jointly control a contract have access to the contract execution operation and state transition information during each execution. For the meat sale contract (Listing 7.1), this is the case for the seller and the buyer.
- **Obligation**: The debtor (i.e., performer and liable) is pre-authorized and has access to the obligation's operations and informational parts (i.e., antecedent, consequent, and events that determine the status of the trigger). The creditor (i.e., rightHolder), on the other hand, is pre-authorized and has access to state transitions only. For example, in the meat sale contract (Listing 7.1, line 22), the seller , as the debtor and performer of the obligation, has access to all information generated during the fulfillment process, including temperature data and related events, and all state transitions. Conversely, the buyer, as the creditor and rightholder of the delivery obligation, is pre-authorized to access state transitions such as when was delivery initiated.
- **Power**: The creditor (i.e., performer and rightHolder) is pre-authorized and has access to the power's operations and all its informational parts (i.e., antecedent, consequent, and events that determine the status of the trigger). The debtor (i.e., liable) of a power has access to state transitions only. For example, as the performer of the terminateContract power shown in Listing 7.1 (line 28), the buyer has access to state transitions and trigger events, and can execute the antecedent and consequent's events, whereas the seller has only access to the state transition information.

7.7 Conclusion

In this chapter, we extended a legal contract ontology with an RBAC-based access control model that enables defining rules and restricting access to contract resources (including assets, events, and their attributes and operations). The semantics of SymboleoAC also includes rules that determine the controller of each resource, and default pre-authorization rules that come into effect when a resource is instantiated. In the next chapter, we introduce the SymboleoAC language, which extends the original Symboleo grammar with access control constructs and other elements that exploit the ontology defined in this chapter.

Chapter 8

SymboleoAC Language

In this chapter, we present the new syntax and semantics made for SymboleoAC, outline the new grammar (❸ in Figure 1.1) and newly introduced validation rules for its Integrated Development Environment (IDE) (❹), and explain their implementation.

8.1 SymboleoAC Language and IDE

The initial development of the Symboleo language and IDE was carried out by Sharifi et al. [109], using Eclipse's Xtext framework[1]. Subsequently, Rasti et al. [101] and Parvizimosaed et al. [90] built upon previous work by enhancing the Xtext grammar, resulting in the existing Symboleo IDE [101], which also integrates the Symboleo2SC compiler to produce code that monitors and verifies legal contract execution.

In our work, we build the SymboleoAC IDE by improving and extending the existing Symboleo Xtext grammar. We first focus on aspects directly related to smart contracts, which include mechanisms such as *variable assignment* in event that support the execution of contract obligations and powers. Additionally, our extensions involve the implementation of control mechanisms for contract execution, including *notifications*. We also add security aspects to the language by building *access control* mechanisms along with controller rules and pre-authorization rules. Table 8.1 summarizes the new features of the SymboleoAC language.

We also leverage the Application Programming Interface (API) provided by Xtext to develop the SymboleoAC IDE, define validation rules, and enable code generation. Specifically, we use two interface classes: `AbstractDeclarativeValidator` and `AbstractGenerator`. Building upon previous work, particularly the `SymboleoGenerator` and `SymboleoValidator` classes, we developed the compiler for SymboleoAC2SC (an enhanced version of Symboleo2SC), and the code generator for extended Hyperledger Fabric smart contracts, the message broker configuration, and the CEP configuration. Details regarding the code generation process for SymboleoAC2SC will be discussed in Chapter 9.

[1] https://eclipse.dev/Xtext/

Table 8.1: New features of the SymboleoAC language

Feature	Description
Assignment Expression (Section 8.2)	Supports variable assignment in events used by contractual obligations and powers.
Attribute Qualifiers (Section 8.3)	Supports the use of the thirdParty keyword before attributes of the domain role to differentiate between contracting parties and other roles, for the purpose of access control.
Access Control Primitives (Section 8.4)	Support defining and enforcing access control policies, as well as controller rules for each resource and pre-authorization rules that give access to resources for different roles.
Automated Notifications (Section 8.5)	Support notifications to enable contract parties and other roles to track updates to the state of the contract, obligations, and powers.
External Data Integration (Section 8.6)	Integrates external data sources to access real-world information, e.g., from IoT devices. This enables the contract to make informed real-time decisions and trigger actions based on external events or data.

8.2 Assignment Expression

To enhance the expressiveness and functionality of the Symboleo language (a feature hence inherited by SymboleoAC), we have made significant updates to its grammar. These updates include the incorporation of a new assignment statement that can be utilized in both the antecedent and consequent of obligations and powers. This addition allows specification writers to define more dynamic and context-aware behaviors within the specifications. The implementation of this new grammatical rule introduces two distinct assignment functions, which provide flexibility in how assignments are executed in relation to specified events. Listing 8.1 shows the grammar of the assignment. Assignments are used in two new predicate functions:

- HappensAssign(e, ae): This function has two parameters, event e and assignment expression ae. Event e must happen before evaluating the assignment expression and assigning the new values to the variables.
- Assign(ae): To provide more flexibility for specification writers, this function has only one argument, which is the unconditional assignment ae itself. Using this function, the events used to trigger or activate the obligation or power that contains the function must happen before evaluating the assignment expression.

As with any predicate function, these two functions will be evaluated according to their location in obligations or powers. Note that the ae argument can be a composition of

several assignment expressions. An example of the new assignment expression syntax is shown in the consequent of the oAssign obligation at line (22) in Listing 8.2.

```
1  OAssignment:
2    {OAssignExpression} name2= VariableDotExpression op=":=" (value=Expression);
3
4
5  PredicateFunction:
6    {PredicateFunctionAssignment} name='HappensAssign' '(' event=Event ',' (assignment+=OAssignment (';' assignment+=
       OAssignment )*)?')' |
7    {PredicateFunctionAssignmentOnly} name='Assign' '(' (assignment+=OAssignment (';' assignment+=OAssignment )*)?')'
8      ;
```

Listing 8.1: New assignment expression grammar rule.

```
1  Domain covidVaccineProcurementD
2    Manufacturer isA Role;
3    Government isA Role;
4    Requested isA Event with Env reqID: String, Env dosage: Number, Env date: Date;
5    Delivered isA Event with Env reqID: String, Env dosage: Number, Env delAddr: Location, Env date: Date, Env
       temperature: Number;
6    Remain isA Asset with value:Number, owner: Government;
7    PaidAmount isA Asset with value:Number, owner: Government;
8  endDomain
9
10 Contract VaccineProcurementC (pfizer: Manufacturer, mcdc: Government, approval: Boolean,
11   unitPrice: Number, minQuantity: Number, maxQuantity:Number, temperature: Number )
12
13 Declarations
14   requested: Requested;
15   delivered: Delivered;
16   remain: Remain with value:= maxQuantity, owner:=mcdc;
17   paidAmount:PaidAmount with value:=0, owner:=mcdc;
18   withdrewApproval:
19 Obligations
20
21   // Calculate the remaining doses and the price of the doses delivered when the required doses are delivered,
       fulfilling all the agreed-upon conditions
22   oAssign: Happens(requested)->O(mcdc,pfizer,Happens(delivered) and delivered.reqID==requested.reqID,
23            HappensAssign(Fulfilled(obligations.oDeliver), remain.value:=remain.value-delivered.dosage;
24            paidAmount.value:=delivered.dosage*vaccineDose.price) and delivered.reqID==requested.reqID);
25
26 endContract
```

Listing 8.2: An example snippet using the HappensAssign() expression syntax in a Vaccine Procurement contract specification in SYMBOLEO.

8.3 Attribute Qualifiers

An attribute qualifiers thirdParty in SYMBOLEOAC is added to differentiate between contracting parties and third-party roles in the code generation process, specifically for generating access control rules, controller rules and pre-authorization rules. By introducing this modifier, SYMBOLEOAC can assign specific permissions to main contracting parties, ensuring they are treated distinctly from the other third parties. This distinction is critical in access control scenarios, where third-party entities, such as assessor or external transportation company, need to interact with the contract but should not have the same level of control or access as the contracting parties. The thirdParty modifier helps tailor permissions to limit third-party actions, ensuring that they can only access and interact with contract as resource as specified by the access control rules, thus enhancing security in SYMBOLEOAC and the generated smart contract. The roles in Listing 8.3, lines (4-6),

exemplify third-party roles that are distinct from the contracting parties, i.e., the seller and buyer.

```
1  Domain meatSaleDomain
2    Seller isA Role with returnAddress: String, name: String;
3    Buyer isA Role with name: String, warehouse: String;
4    TransportCo isA Role thirdParty with name:String;
5    Assessor isA Role thirdParty with name: String;
6    Regulator isA Role thirdParty with name: String;
7    Currency isAn Enumeration(CAD, USD, EUR);
8    MeatQuality isAn Enumeration(PRIME, AAA, AA, A);
9    PerishableGood isAn Asset with quantity: Number, quality: MeatQuality, barcode:String, owner: Seller;
10 endDomain
11
12 Contract MeatSale (//omitted code
13 )
14
15 Declarations
16   seller: Seller with name:= sellerP.name, returnAddress := sellerP.returnAddress;
17
18 Obligations
19   delivery: Obligation(seller, buyer, true, WhappensBefore(delivered, delivered.delDueDate) and Happens(temploctime)
        and temploctime.temp <= 18 and delivered.deliveryAddress == buyer.warehouse);
20
21 Powers
22   suspendDelivery : Happens(Violated(obligations.payment)) -> Power(seller, buyer, true, Suspended(obligations.
        delivery));
23
24 endContract
```

Listing 8.3: An example snippet using **thirdParty** in a Meal Sale contract specification in SymboleoAC.

8.4 Access Control Primitive

To enhance the Symboleo language for better governance and compliance in smart contracts, we have introduced an access control primitive. This addition aims to define and enforce access control policies directly within the Symboleo language, allowing for precise control over Symboleo resources and actions associated with different roles. The syntax for this part of the grammar is defined by ACPolicy, as shown in Listing 8.4, line (1).

The grammar starts with the **ACPolicy** keyword, which indicates the beginning of the access control section. This section governs who has permission to perform certain actions on specific resources within the contract. It includes the designation of controllers, who are responsible for managing the rules associated with the policy. The rules form the core of the access control policy. Each rule defines decisions (grant or revoke) for specific roles to access resources in the contract. These rules dictate how entities (such as roles) interact with resources based on the controller's authorization.

The syntax is structured as shown in Listing 8.4, lines (8-10). Its definition includes, but is not limited to, the following elements:

- **To** *accessedRole*: the role for which access is being granted or revoked.
- **On** *accessedResource*: the resource on which access is being controlled.
- **Controller**: the entity or entities responsible for managing this access.

Lastly, the resource construct incorporates various resource types that can be accessed or controlled, as shown in Listing 8.4, lines (12-20). This diverse representation of resources allows for comprehensive control mechanisms, addressing various scenarios where access must be managed, such as obligations, powers, ontology types, attributes, and operations. An example of the new access control syntax is shown in Listing 8.5.

```
1  ACPolicy:
2     ('with' 'Controller' (controller+=Controller ',')* (controller+=Controller))
3    ;
4
5  Controller:
6      controllerType=VariableDotExpression;
7
8  Rule:
9    name=ID ':'  (('Grant' | 'Revoke') action=Action 'To' accessedRole=VariableDotExpression 'On' accessedResource=
       Resource  'by' controller=VariableDotExpression)
10    ;
11
12 Resource:
13   {ResourceObligation} ('obligations.' resourceOp = [Obligation]) |
14   {ResourcePower} ( 'powers.' resourcePo = [Power]) |
15   {ResourceOntologyType} ( resourceOn = OntologyType) |
16   {ResourceACPolicy} (resourceAc=ACPolicy)|
17   {ResourceAttribute} (resourceAt=Attribute) |
18   {ResourceDot} (resourceDot= VariableDotExpression)|
19   {ResourceOperation} (resourceOpe=Operation)
20 ;
21
22 Action:
23   name=('read' | 'write' | 'all');
```

Listing 8.4: Access control grammar rule.

```
1  ACPolicy with Controller regulator //controller of policy are the regulator who can override rule and pre-authrization
       rule
2    Rule1: Grant read To buyer On goods.quantity by seller; //access to specific asset attribute
3    Rule2: Grant read To assessor  On obligations.delivery by seller; //access to obligation
4    Rule3: Grant read To transportCo On inspectedQuality by assessor;
```

Listing 8.5: An example snippet using access control in a Meal Sale contract specification in SymboleoAC.

8.5 Notification

We implemented a notification mechanism (see 5 and 6 in Figure 5.1) to ensure that contracting parties are informed of important state transitions within the contract. This allows parties to respond promptly to changes, such as the fulfillment of obligations, violations, unsuccessful terminations, and so on. However, the SymboleoAC language does not introduce new grammar constructs for modeling events since it already supports the ontology type **Event**. Instead, the code generator (see Chapter 9) extends the underlying semantics to enable these events to participate in run-time execution. This extended foundation allows **Event** instances to be emitted to the outside world, interact with cyber-physical processes and components, and serve as notification events within the system.

A notified event will be triggered after each transaction in the smart contract and will iterate through the state of obligation and power to check if there is a violation or an unsuccessful termination and will report it. The notified event will include the following

information: contract identifier, instance identifier (of the contract, power, or obligation), instance name, start state, transitions name, end state, and timestamp. For example, when an obligation is transferred to the **UnsuccessfulTermination** state, we call the notifier function `trigger_notification()`, which triggers the Hyperledger Fabric event notified.

To integrate state change notifications into SymboleoAC, we began by enhancing the generation of Hyperledger Fabric smart contracts to emit events upon state changes. We implement the event emission in our smart contract using `ctx.stub.setEvent()` within our transaction functions, such as violateObligation_latePayment for the meat sale contract, where we define the event payload to capture the updated state. Next, we set up a client application (in Node.js, part of the SymboleoAC API seen in Chapter 6) using the Fabric SDK to listen for these events. The client must register for the specific chaincode events with the SDK's event hub, and handle incoming events by parsing the payload and invoking a notification function. For the notification process, we configure a topic-based publish-subscribe message queuing. Its architecture consists of three main components, which follow the corresponding abstract phases in Figure 5.3, with an implementation (per-role notification queuing) discussed in Section 6.3.5:

- A publisher, which transmits data associated with a specific topic/queue;
- A subscriber, which is subscribed to a particular topic/queue to receive updates when the topic/queue changes; and
- A message broker, acting as an intermediary server that collects data from various publishers and delivers it to subscribers already registered for that topic/queue.

In our client application (SymboleoAC API, specifically the Node.js application from Section 6.5), upon receiving an event, we publish the payload to the message broker, ensuring the message is properly formatted and sent to the appropriate topic. Lastly, subscribers to this broker will receive these notifications.

8.6 External Data Integration

To enhance the expressiveness of the Symboleo language and support the integration of IoT data and other cyber-physical information, we extended the language by introducing a new domain concept in SymboleoAC: the DataTransfer, briefly discussed in the ontology (Section 7.4.1). This addition enables the language to represent data originating from external sources, specifically IoT devices.

By modeling IoT-driven inputs as **DataTransfer** in SymboleoAC, the contract can evaluate conditions, trigger obligations or powers, and update its state in response to real-world changes. Each **DataTransfer** domain type must define three key attributes: condition, window, and count. These values are subsequently used by a CEP engine to formulate the rules that determine when an alert should be triggered.

The corresponding addition to SymboleO grammar in Xtext is shown in Listing 8.6. An example of using a **DataTransfer** is shown in Listing 8.7.

As the DataTransfer and Event ontology concepts are both subclasses of AbstractEvent (see Figure 7.3), DataTransfer inherits the same functionality as Event. Practically, in SymboleoAC, this means it supports the same predicate functions [99], such as **Happens**, as shown in Listings 8.7.

```

OntologyType:
  name=("Asset" | "Event" | "Role" | "Contract" | "DataTransfer" );
```

Listing 8.6: Grammar extension with the new DataTransfer ontology type.

```
Domain meatSaleDomain
  Seller isA Role with returnAddress: String, name: String;
  Buyer isA Role with name: String, warehouse: String;
  TransportCo isA Role thirdParty with name:String;
   // omitted code
  TemperatureAlert isA DataTransfer with Env sensorId: String, Env value: Number, Env sensorTimestamp:String,
     condition: String, window: Number, count: Number, controller:Seller;
   // omitted code
endDomain


Contract MeatSale (   // omitted code

)

Declarations
  seller: Seller with name:= sellerP.name, returnAddress := sellerP.returnAddress;
  buyer: Buyer with name:=buyerP.name, warehouse:= buyerP.warehouse;
  transportCo: TransportCo with name:= transportCoP.name;
    // omitted code
  temperatureAlert: TemperatureAlert with condition:= "value <= 18", window:=10, count:=1, controller:=seller;
    // omitted code

Obligations
  delivery: Obligation(seller, buyer, true, WhappensBefore(delivered, delivered.delDueDate) and  delivered.
     deliveryAddress == buyer.warehouse and not Happens(temperatureAlert));
     // omitted code

Powers
  suspendDelivery : Happens(Violated(obligations.payment)) -> Power(seller, buyer, true, Suspended(obligations.
     delivery)) with Controller seller;


ACPolicy with Controller seller
   // omitted code
 Rule9: Revoke read To buyer On goods.quality by seller;
 Rule10: Grant read To buyer On temperatureAlert.value by seller;
endContract
```

Listing 8.7: An example snippet using **DataTransfer** in a Meal Sale contract specification in SymboleoAC.

8.7 IDE-Based Validation

We implemented validation rules to ensure that SymboleoAC policies and contract level rules are correctly defined within the Eclipse-based IDE. The Xtext provides the `AbstractDeclarativeValidator` interface to support custom validation logic. We extended the existing `SymboleoValidator` class originally written by Rasti [99], which itself

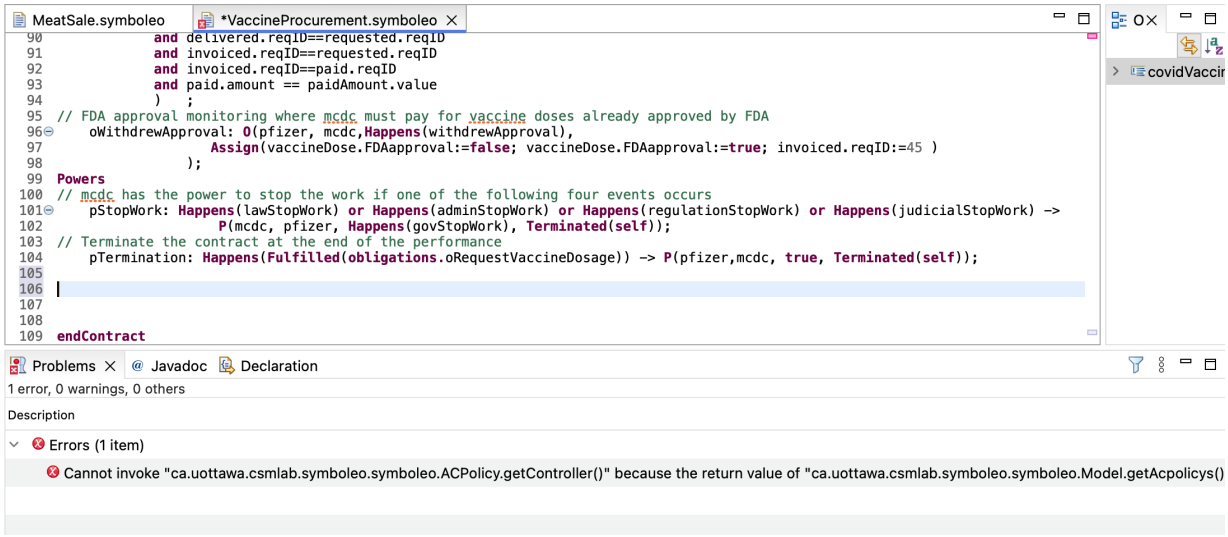


Figure 8.1: Example of a validation error when no access control policy is specified.

extends `AbstractDeclarativeValidator`, in order to incorporate additional validation rules. The updated validator is available online.[2]

To define a validation rule in Xtext, a method annotated with `@Check(CheckType.FAST)` must be implemented. Once the contract model is parsed, Xtext automatically triggers these validation methods based on their parameter types. Below, we outline and describe the validation rules developed for the SymboleoAC IDE.

Specifying the SymboleoAC ACPolicy is mandatory.

The SymboleoAC policy must be defined in each contract specification. The JavaScript code for the contract specification will not be generated unless the policy is specified. Figure 8.1 illustrates a validation error triggered when no access control policy is defined in the specification.

The accessedRole and controller parameters of rules must be Roles.

Both accessedRole and controller must be of type Role. This validation rule ensures that the assigned elements indeed conform to the expected Role type. Figure 8.2 shows an example where accessedRole is incorrectly assigned a value that is not a Role. The editor highlights the error and indicates the required correction.

8.8 Conclusion

In this chapter, we presented the extended syntax and semantics introduced in SymboleoAC, detailing the new language constructs and the validation rules added to the

[2] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-IDE

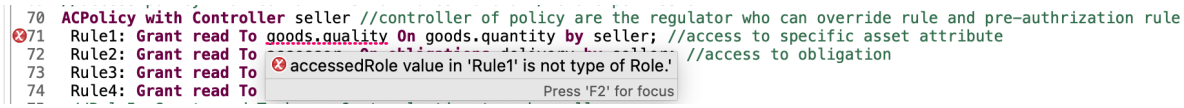


Figure 8.2: Example of a validation error where accessedRole is assigned a value that is not of type Role.

SymboleoAC IDE. We described how these extensions support access control, notification, the integration of external data, and other important features. Together, these additions greatly enhance the expressiveness of the original Symboleo language. In the next chapter, we describe how the SymboleoAC ontology and language constructs are operationalized through an automated code generation process.

Chapter 9

SymboleoAC2SC – Smart Contract Code Generator

In this chapter, we start by outlining how the SymboleoAC ontology is implemented as a reusable SymboleoAC JavaScript library (SymboleoACJS) (2 in Figure 9.1), which will be utilized by the code generated through the SymboleoAC to Smart Contract tool (SymboleoAC2SC) (5). Next, we explain how SymboleoAC2SC converts SymboleoAC contract specifications into secure Hyperledger Fabric smart contracts (6), and into configuration files for the message broker, the CEP, and IoT devices.

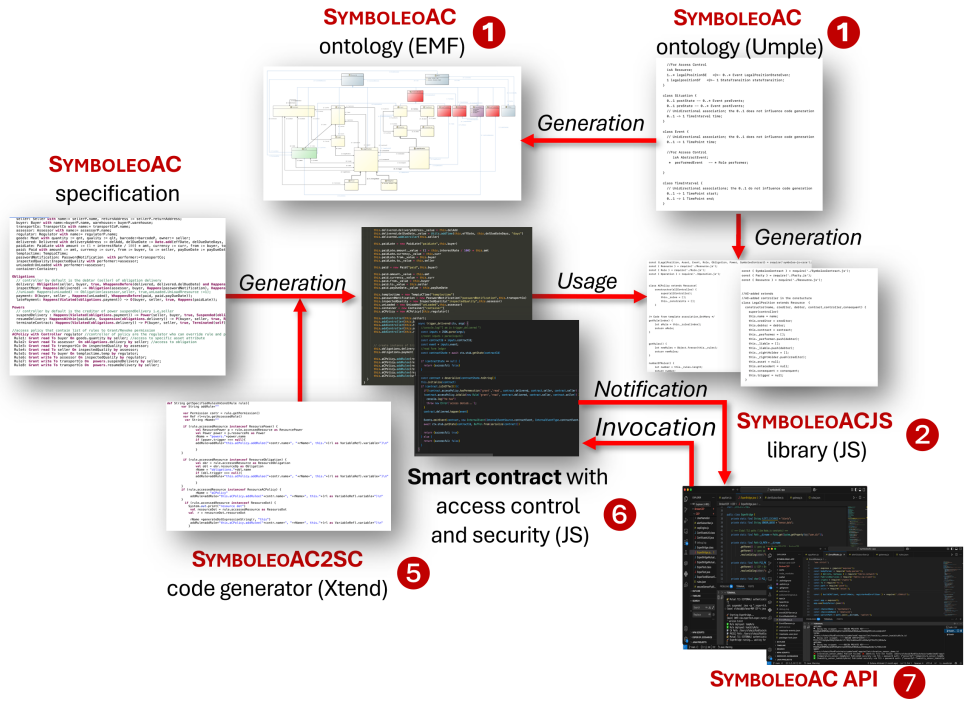


Figure 9.1: Subset of the overview from Figure 1.1 that is the focus of this chapter.

9.1 Implementation of SymboleoAC Ontology & Rules

To generate a robust and secure smart contract, we extended the JavaScript library implementation of the original ontology (SymboleoJS [101]) to cover the new SymboleoAC concepts introduced in Section 7.4, leading to a reusable and secure library called SymboleoACJS (2 in Figure 9.1). This library can then be used by JavaScript programs generated from a SymboleoAC specification, which can then be run on the Hyperledger Fabric[1] smart contract platform while interacting with the external world (IoT devices, CEP, and message broker) via the SymboleoAC API (7).

Note that JavaScript code is used directly in this section instead of more abstract algorithms as the code complexity is low and similar to what corresponding algorithms would look like.

The initial stage of the development of SymboleoACJS begins with the integration of RBAC ontology concepts and their relations with Symboleo's ontology concepts and relations (Chapter 7, with a formal representation specified with the Umple[2] modeling tool [38,74]. The resulting Umple model was used to generate EMF code (1 in Figure 9.1), which is then turned into the class diagram shown in Figure 7.3. Other tools built on EMF can reuse that code, but the latter is not used for purposes other than visualization in this thesis.

Following the approach taken by Rasti [101], we first added/updated the classes, attributes, and associations of the SymboleoAC ontology in the Umple model, from which we automatically generated corresponding Java files[3] with many utility functions for object navigation and updates. Since Umple does not support the direct generation of JavaScript code, we manually transformed the Java code into its corresponding JavaScript form.

Here, we are illustrating the JavaScript equivalent of some of the ontology classes (from the integrated ontology in Figure 7.3) and access control operations (Section 7.2), while the rest is accessible online[4]. Each ontology class has its equivalent JavaScript class. In particular, the `Resource` class has a list of controller roles (see Figure 7.3), and all access-controlled classes in the ontology extend `Resource`. This class also contains the necessary methods to manipulate controllers (`add/removeController()`); such methods are generated automatically from Umple for all attributes and associations.

Class constructors also handle the default access control rules discussed in Section 7.6. For example, Listing 9.1 shows the `Obligation` class, where the debtor is assigned as the controller by default in the constructor of its superclass (`LegalPosition`), which then sets it properly in its own superclass (`Resource`).

[1] https://www.hyperledger.org/projects
[2] https://cruise.umple.org/umple/
[3] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-JS-Core/tree/main/ontology/Java
[4] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-JS-Core/tree/main/ontology/core

```
class Obligation extends LegalPosition {
  constructor(name, creditor, debtor, contract, surviving) {
    super(name, creditor, debtor, contract, debtor);
  }
// ... omitted code
}

class LegalPosition extends Resource  {
  constructor(name, creditor, debtor, contract, controller) {
    super(controller)
  }
// ... omitted code
}
```

Listing 9.1: Snippets of the `Obligation` and `LegalPosition` JavaScript classes, where the debtor is passed as the default controller in the superclass constructor calls.

The `Policy` class serves as the core component in SymboleoAC containing rules that determine access permissions based on roles (Figure 7.3). It maintains a structured mechanism for dynamically evaluating, storing, and enforcing access control decisions. The class contains methods such as `addRule()`, `addPolicy()`, and `addRulee()`, each serving a distinct function in access governance, as well as other rule management methods such as `updateRule()` and `removeRule()`.

Listing 9.2 shows the `addRulee()` function, which dynamically classifies a new access request as either a rule or a policy, the latter being a collection of rules. If the requester (`aByRole`) has control privileges (i.e., a controller of `policy`), the rule is treated as a policy (`addPolicy(aRule)`). Otherwise, it is treated as a standard rule (`addRule(aRule)`).

```
function addRulee(aDecision, aPermission, aAccessedResource, aAccessedRole, aByRole) {
    let aRule = new Rule(aDecision, aPermission, aAccessedResource, aAccessedRole, aByRole, this);

    if (aRule !== null && typeof aRule !== 'undefined') {
        if (this.findController(aByRole)) {
            this.addPolicy(aRule);
        } else {
            this.addRule(aRule);
        }
    }

    return aRule;
}
```

Listing 9.2: SymboleoACJS library function in JavaScript: `addRulee()`.

Listing 9.3 shows `addRule()`, which introduces new access rules while ensuring they are valid and do not conflict with pre-authorization rules or constraints.

```
function addRule(aRule) {
    let wasAdded = false;

    if (this.hasPermesstion(aRule.decision, aRule.permission, aRule.accessedResource, aRule.accessedRole, aRule.byRole
     )) {
        return false;
    }

    if (this.isValid(aRule) && this.updateRule(aRule)) {
        this._rules.push(aRule);
    }

    wasAdded = true;
    return wasAdded;
}
```

Listing 9.3: SymboleoACJS library function in JavaScript: `addRule()`.

The `addPolicy()` function, shown in Listing 9.4, enforces rules restrictions, preventing unauthorized access while ensuring compliance with security policies. In other words, at contract instantiation time (pre-authorization rules), it imposes pre-existing constraints, while at run-time (incremental policy updates), as new rules are added (e.g., `addRule()` or `addRulee()`), policies dynamically adjust the access control model. Unauthorized changes are automatically rejected, ensuring that role-based constraints remain intact.

```
function addPolicy(aRule) {
    let wasAdded = false;

    if (this.findPolicy(aRule) || this.hasPermesstion(aRule.decision, aRule.permission, aRule.accessedResource, aRule
    .accessedRole, aRule.byRole)) {
        return false;
    }

    if (this.updatePolicy(aRule)) {
        this._constraints.push(aRule);
    }

    wasAdded = true;
    return wasAdded;
}

```

Listing 9.4: SymboleoACJS library function in JavaScript: `addPolicy()`.

The `authenticate()` function in Listing 9.5 is another SymboleoAC access control operation used at run-time to authenticate a participant's role using the identity information contained in their identifier, issued through certificate-based authentication.

```
function authenticate(inRole, inName, inOrg, inDept,aContract ) {
      const objRole = this.findObject(inName, inRole, aContract)
      if(objRole != null){
        if (inOrg === objRole.org._value && inDept === objRole.dept._value){
          return objRole
        }else{
          return null
         }
        }else{
          return null
        }
      }
```

Listing 9.5: SymboleoACJS library function in JavaScript: `authenticate()`.

Additionally, a set of utility functions were added to the class. Listing 9.6 shows a utility function `hasPermission()` that assesses whether the role holds controller privileges, or is pre-authorized. If any of these conditions is met, we do not need to grant specific permissions to that role.

```
function hasPermission(aDecision, aAction, aAccessedResource, aAccessedRole, aByRole) {
    let aRule = new Rule(aDecision, aAction, aAccessedResource, aAccessedRole, aByRole, this);
    if (this.findRule(aRule)) {
        return true;
    } else {
        if (aRule.accessedResource.findController(aRule.accessedRole)) {
            if (!(aRule.accessedResource instanceof Event) &&
                !(aRule.accessedResource instanceof Asset) &&
                !(aRule.accessedResource instanceof Role)) {
                return true;
            }  // ... omitted code
        }
        if (aRule.accessedResource instanceof Asset) {
            if (aRule.accessedResource._owners._value === aRule.accessedRole) {
                return true;
            }
        }
     // ... omitted code
```

```
}
```

Listing 9.6: Snippet of JavaScript utility function `hasPermission()` in class `Policy`.

Another utility function, `hasPermesstionOnLegalPosition()` (Listing 9.7), checks whether a given role is pre-authorized to perform a specific action on the consequent of a legal position.

```
function hasPermissionOnLegalPosition(aDecision, aAction, aAccessedResource, aAccessedRole, aByRole, aContract) {
    for (const obligationKey of Object.keys(aContract.obligations)) {
        if (
            (this.findObject(aContract.obligations[obligationKey].consequent, aAccessedResource) ||
                this.findObject(aContract.obligations[obligationKey].antecedent, aAccessedResource)) &&
            aContract.obligations[obligationKey].findPerformer(aAccessedRole)
        ) {
            let aRule = new Rule(aDecision, aAction, aContract.obligations[obligationKey], aAccessedRole, aByRole,
 this);
            return this.isValid(aRule);
        }
    }

    for (const powerKey of Object.keys(aContract.powers)) {
        if (
            this.findObject(aContract.powers[powerKey].antecedent, aAccessedResource) &&
            aContract.powers[powerKey]._performer.find(
                (obj) => obj._name === aAccessedRole._name && obj._type === aAccessedRole._type
            )
        ) {
            let aRule = new Rule(aDecision, aAction, aContract.powers[powerKey], aAccessedRole, aByRole, this);
            return this.isValid(aRule);
        }
    }

    return false;
}
```

Listing 9.7: JavaScript utility function `hasPermissionOnLegalPosition()` in class `Policy`.

Furthermore, another utility function, `permissionValid()`, iterates through the list of roles in a given contract and returns the roles that are authorized to access (i.e., read) a specific *resource*. It is used at run-time when emitting notifications; the list of authorized roles is attached to the notification event payload so that the message broker can determine which roles are allowed to subscribe to that notification (see Section 6.3.4).

```
function permissionValid(aAccessedResource, aAccessedRoles, aByRole, aContract) {
  const validRoles = [];

  for (const role of aAccessedRoles) {
    const rule = new Rule('grant', 'read', aAccessedResource, role, aByRole);

    if (((this.hasPermesstion('grant', 'read', aAccessedResource, role, aByRole) ||
    this.hasPermesstionOnLegalPosition('grant','read', aAccessedResource, role, aByRole, aContract)) &&
        this.isValid(rule))) {
      validRoles.push(role.name._value);
    }
  }

  return validRoles; // array with only valid roles
}
```

Listing 9.8: JavaScript utility function `permissionValid()` in class `Policy`.

Other utility functions (available online[5]) collectively form a robust access control management library for SymboleoAC that ensures policies and rules remain consistent, se-

[5] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-JS-Core/blob/main/ontology/core/ACPolicy.js

cure, and up to date: `isValid()` checks if a new rule is compatible with existing policy constraints, and `updatePolicy()` modifies the current set of constraints to align them with changes made by the policy controller. Function `updatePolicy()` is not limited to merely changing decisions from **Grant** to **Revoke** or vice versa; it also dynamically manages access rules by adding, removing, or updating parameters of constraints. Similarly, function `updateRule()` modifies the rules to reflect changes made by controllers of the resources. Listing 9.9 describes the utility function `isValid()`.

```
isValid(aRule) {
    let isValidVar = true;

    this._constraints.forEach(constraint => {
        if (constraint.accessedResource === aRule.accessedResource &&
            constraint.accessedRole === aRule.accessedRole) {

            switch (aRule.decision) {
                case 'grant':
                    switch (aRule.permission) {
                        case 'read':
                            if ((constraint.decision === 'revoke' &&
                                (constraint.permission === 'read' || constraint.permission === 'all'))) {
                                isValidVar = false;
                                return false;
                            }
                            break;
                        case 'write':
                            if ((constraint.decision === 'revoke' &&
                                (constraint.permission === 'write' || constraint.permission === 'all')) ||
                                (constraint.decision === 'grant' && constraint.permission === 'read')) {
                                isValidVar = false;
                                return false;
                            }
                            break;
                        // Omitted rest of the cases...
                    }
                    break;
                // Omitted other decision cases...
            }
        }
    });
    return isValidVar;
}
```

Listing 9.9: Snippet of JavaScript utility function `isValid()` in class `Policy`.

9.2 Deploying and Configuring Off-Chain Runtime Components from SymboleoAC

Building on the architectural framework presented in Chapter 5 and the SymboleoAC API (7) deployment details discussed in Chapter 6, this section clarifies the boundary between i) automated SymboleoAC specification-driven runtime behavior and ii) one-time manual configuration tasks in the framework's off-chain components.

The workflow illustrated in Figure 9.2 highlights how off-chain runtime components are either derived automatically from a SymboleoAC specification and code generation (SymboleoAC2SC) or manually configured once to enhance the execution environment. This distinction is essential for understanding the level of automation achieved by SymboleoAC2SC and the minimal operational effort required to support CPSC execution.

The execution flow illustrated in Figure 9.2 can be summarized through the following operational steps:

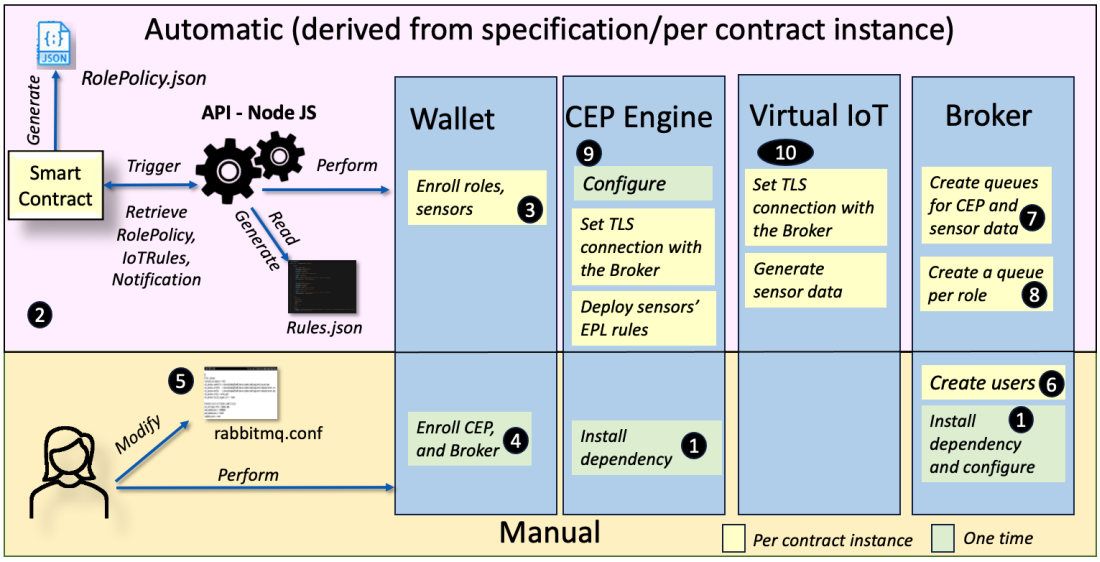


Figure 9.2: Boundary between automated (derived from the SymboleoAC specification per contract instance) and manual configuration tasks in the off-chain runtime components. The upper section illustrates automatically generated and executed steps (2–3, 7–10), while the lower section highlights one-time manual setup tasks (1, 4–6).

(a) Message Broker and CEP Installation (Manual – Done Once Only)

- ❶ Install Esper (CEP engine), RabbitMQ, and all required dependencies.

(b) Retrieve `RolePolicy` and `IoTRules` (Automatic – Per Contract Instance)

- ❷ After deployment of the generated smart contract on the Hyperledger platform, the regulator invokes the transaction `getRolePolicy()` to retrieve the role policy, and then invokes `getIoTCondition()` to retrieve the IoT rules generated by SymboleoAC2SC.
- ❸ Enroll roles in the wallet based on the generated `RolePolicy.json`, and enroll IoT sensors based on the generated `rules.json`.

(c) Enroll Components in the Wallet (Manual – Done Once Only)

- ❹ Enroll the message broker and CEP identities in the wallet.

(d) RabbitMQ Configuration (Manual – Once + Per Contract Instance)

- ❺ Modify `rabbitmq.conf` to configure authentication mechanisms, messaging protocol (e.g., AMQP over TLS), and secure port numbers as shown in Figure 6.1.
- ❻ Create users and define their permissions (read/write access to queues). For role-based queues, permissions are derived from the SymboleoAC policy.

(e) RabbitMQ Configuration (Automatic - Per Contract Instance)

- ❼ Create queues for CEP and sensor data exchange.

- ❽ Create one queue per role, as specified in `rules.json`.

(f) CEP Configuration (Automatic – Per Contract Instance)

- ❾ Configure the CEP engine to connect to the message broker using TLS authentication and deploy the corresponding EPL rules derived from `rules.json`.

(g) Virtual Sensor Configuration (Automatic – Per Contract Instance)

- ❿ Configure TLS communication with the message broker, assign the appropriate username and password credentials, and generate virtual sensors.

9.3 Implementation of Access Control on Hyperledger

Hyperledger Fabric is an enterprise-level and permissioned distributed ledger platform, used to run smart contracts [13]. An existing tool, SymboleO2SC [101], already generates smart contracts from SymboleO specifications, but without concern for access control. Our proposed access control model, implemented in SymboleOAC2SC, includes two layers of security for SymboleOAC smart contracts: authentication and authorization (see ❺ in Figure 5.1).

9.3.1 Authentication

In addition to the role-based access control provided by SymboleOAC, we incorporate an additional layer of security to authenticate and authorize contract parties within the Hyperledger Fabric environment. This is achieved by leveraging the `ClientIdentity` class[6] from the `fabric-chaincode-node` package. Each party involved in a SymboleOAC contract is assigned a *digital certificate*, which is generated by the network administrator or SymboleOAC contract regulator. Each certificate contains a pair of private and public keys, which is securely stored in the party's digital wallet. A private key is associated with each party and is used to sign transactions, ensuring authenticity, while a public key allows others to verify the legitimacy of the party's actions. The steps of this process are:

1. We extract the roles from the SymboleOAC specification using the `storeRolesPolicy()` transaction and we store the on-chain roles list in the ledger as `ACPolicyRecord` with a signed hash using the `crypto` class[7], and emit a tamper-proof event. It can only be called by the SymboleOAC regulator or the blockchain admin from the SymboleOAC API. Listing 9.10 shows a snippet demonstrating how we build the list of role objects from the SymboleOAC specification.

```
async storeRolesPolicy(ctx, contractId) {
    let roleObj;
    const contractState = await ctx.stub.getState(contractId)
```

[6] https://hyperledger.github.io/fabric-chaincode-node/release-2.2/api/fabric-shim.ClientIdentity.html

[7] https://nodejs.org/api/crypto.html

```
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())
    this.initialize(contract)


    const cid = new ClientIdentity(ctx.stub);
    const userId = cid.getID();
    const role = cid.getAttributeValue('HF.role');

    console.log("Attr name")
    console.log(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
        cid.getAttributeValue('organization'), cid.getAttributeValue('department'))

    try{
      if (role !== 'Admin' && role !== 'Regulator') {

      throw new Error('Only Admin or Regulator can trigger roles policy storage');
     }else{
        roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
        cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

             if(roleObj === null ){
              throw new Error('Unauthorized: Unknown access');

        }
     }// else
     }catch(err){
        console.log('access control error: ', err)
        return { successful: false, message: err.message }
       }// End of first layer

    // Build roles policy from  list in contract spec
    const policy = {
      roles: contract._roles.map(role => ({
        name: role._name,
        type: role._type,
        dept: role.dept._value,
        org: role.org._value
      })),
      metadata: {
        storedBy: cid.getID(),
        timestamp: new Date().toISOString()
      }
    };

    const policyStr = JSON.stringify(policy);
    const policyHash = crypto.createHash('sha256').update(policyStr).digest();

    // ... Omitted code

    await ctx.stub.putState('ACPolicyRecord', Buffer.from(JSON.stringify(record)));

    // ... Omitted code

  }
```

Listing 9.10: Hyperledger Fabric transaction `storeRolesPolicy()` in JavaScript.

2. We get a list of the contract's roles specified in the SymboleoAC specification of the contract that was hashed and stored previously in the Hyperledger Fabric ledger. When the smart contract is instantiated, it retrieves the list of users from the contract through the `getRolePolicy()` transaction shown in Listing 9.11, but only if the caller is authenticated and authorized using Fabric CA and the SymboleoAC API, which, in our case, includes only the admin and regulator roles. It also returns the stored SHA-256 hash, which the caller from the SymboleoAC API (i.e., admin or regulator) can compare to a locally computed hash of the contract's current role list. This supports integrity by ensuring that the roles specified in the SymboleoAC specification are consistent with the list stored on-chain and have not been tampered.

```
async getRolePolicy(ctx, contractId) {
    // ... Omitted code
    // Start of first security layer
    const cid = new ClientIdentity(ctx.stub);
    const userId = cid.getID();
```

```
    const role = cid.getAttributeValue('HF.role');

    try{
      if (role !== 'Admin' && role !== 'Regulator') {

      throw new Error('Only Admin or Regulator can trigger roles policy storage');
     }else{
        roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
        cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

             if(roleObj === null ){
              throw new Error('Unauthorized: Unknown access');

        }
     }// else
     }catch(err){
        console.log('access control error: ', err)
        return { successful: false, message: err.message }
       }// End of first layer

    const policyBytes = await ctx.stub.getState('ACPolicyRecord');
    if (!policyBytes || policyBytes.length === 0) {
      return { successful: false, message: 'ACPolicyRecord not found' };
    }

    const policy = JSON.parse(policyBytes.toString());
    // ... Omitted code
  }
```

Listing 9.11: Hyperledger Fabric transaction `getRolePolicy()` in JavaScript.

3. We assign a unique identity and certificate to each user in the list retrieved from the previous step, enabling secure interaction with the blockchain. These certificates are issued by Hyperledger Fabric's Certificate Authority (CA), and then stored in the *wallet* and used for authentication and role-based access control for SymboleoAC. The smart contract automatically calls an external API, interacting with this CA, to issue X.509 certificates for them using the `registerAndEnrollUser()` utility function from our SymboleoAC API, shown in Listing 9.12. This function plays a crucial role in managing user identities within a Hyperledger Fabric network by handling the registration and enrollment of new users with the CA.

```
exports.registerAndEnrollUser = async (caClient, wallet, orgMspId, userId, affiliation, attributeValue) => {
  try {
    // Check to see if we've already enrolled the user
    const userIdentity = await wallet.get(userId);
    if (userIdentity) {
      // Omitted code
      return;
    }

    // Must use an admin to register a new user
    const adminIdentity = await wallet.get(adminUserId);
    if (!adminIdentity) {
      console.log('An identity for the admin user does not exist in the wallet');
      console.log('Enroll the admin user before retrying');
      return;
    }

    // build a user object for authenticating with the CA
    const provider = wallet.getProviderRegistry().getProvider(adminIdentity.type);
    const adminUser = await provider.getUserContext(adminIdentity, adminUserId);

    // Register the user, enroll the user, and import the new identity into the wallet.
    // if affiliation is specified by client, the affiliation value must be configured in CA
      // Omitted code
    ];

    const secret = await caClient.register({
      affiliation: affiliation,
      enrollmentID: userId,
      attrs: attributes,
      role: 'client',
      caname: 'ca-org1'
    }, adminUser);

    const enrollment = await caClient.enroll({
      enrollmentID: userId,
```

```
37        enrollmentSecret: secret
38      });
39
40      const x509Identity = {
41        credentials: {
42          certificate: enrollment.certificate,
43          privateKey: enrollment.key.toBytes(),
44        },
45        mspId: orgMspId,
46        type: 'X.509',
47      };
48
49      await wallet.put(userId, x509Identity);
50      console.log('\nRegistered user attributes:', attributes);
51          // Omitted code
52
53    } catch (error) {
54      // Omitted code
55    }
56  };
```

Listing 9.12: `registerAndEnrollUser()` utility function used by the SymboleoAC API to register and enroll SymboleoAC users via the Hyperledger Fabric CA API.

This function also ensures that each user is properly authenticated and authorized to interact with the blockchain going through the following steps:

(a) When invoked, the function first checks whether the user's identity already exists in the wallet. If the identity is found, it logs a message and exits, preventing redundant registrations (lines 3-8). If the identity is not present, the function verifies the existence of an admin identity in the wallet, as only an admin can register new users. If the admin identity is missing, it warns that the admin must first enroll before proceeding (lines 10-16).

(b) Once the admin identity is verified, the function constructs a user object for authentication with the CA. It then registers the new user by assigning an enrollment identifier, an affiliation (which must be pre-configured in the CA), and a set of attributes that help define the user's permissions and access control within the network (lines 22-33).

(c) After registration, the function enrolls the user by obtaining an enrollment secret and requesting an X.509 certificate from the CA. The generated certificate, along with its private key, is structured into an `x509Identity` object, which includes the organization's Membership Service Provider (MSP) identifier and the certificate type. Finally, the function stores the newly created identity in the wallet, making it available for blockchain interactions (lines 35-51).

4. The assigned identity and the certificate of the user invoking the transaction are verified using the `ClientIdentity` class in Hyperledger Fabric. Listing 9.13 shows the `authenticate()` function that is called from each transaction, hence applying the first level of security to automatically verify the identity of the caller of the transaction. When a user submits a transaction, e.g., `trigger_delivered` by the seller in our meat sale running example, the function extracts the user's identity `userId` from the certificate provided during the transaction invocation. Specifically, it parses the enrollment identifier from the CN (common name) field of the certificate, which corresponds to the user registered and enrolled through the `registerAndEnrollUser()` utility function from Listing 9.12. To ensure that the user is authorized, a set

of attributes are used (e.g., organization and department) to check if the extracted enrollment identifier from the certificate matches the expected attributes from the SymboleoAC specification. If the identity does not conform to this expected attribute, access to the transaction is denied.

```
async trigger_delivered(ctx , args)  {
// start of first security layer
    const cid = new ClientIdentity(ctx.stub);
    let roleObj;

    const inputs = JSON.parse(args);
    const contractId = inputs.contractId;
    const event = inputs.event;
    const contractState = await ctx.stub.getState(contractId)
    if (contractState == null) {
      return {successful: false}
    }
    const contract = deserialize(contractState.toString())

    this.initialize(contract)
    if (contract.isInEffect()  ){

    try{

       roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
       cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

            if(roleObj === null ){ // this means the roleObj (role who calls the transaction) exists in our
     contract
             throw new Error('Unauthorized: Unknown access');
        }

    }catch(err){
        console.log('access control error: ', err)
        return { successful: false, message: err.message }
    }// end of first layer
// ... Omitted code
}
```

Listing 9.13: Calling the `authenticate()` function to apply the security first layer for, in JavaScript.

9.3.2 Authorization

Authorization is the second layer of our security model, where we check whether the caller has permission to invoke a transaction and access the contract content. We use the utility function `hasPermission()` from Listing 9.6 to check whether the user is pre-authorized or if there are additional rules granting them permission at run-time. Additionally, we use `isValid()` from Listing 9.9 to verify if there are any constraints that override pre-authorization rules or rules explicitly specified at design time. If the caller has permission for the resource without restrictions and possesses a valid key, they can execute the transaction and access the resource.

Listing 9.14 illustrates the use of this second security layer with the generated transaction `trigger_delivered()`.

```
async trigger_delivered(ctx, args) {
   // Omitted the non-access control related code for clarity

   if (contract.isInEffect()  ){

   try{
    // Layer 1 of security (Authentication)

    // Layer 2 of security (Authorization)


```

```
12          let controllers = contract.paid._controller
13        if(!contract.accessPolicy.hasPermesstion('grant','read', contract.paid, roleObj, contract.paid.getController(
          controllers.length - 1)) ||
14              !contract.accessPolicy.isValid(new Rule('grant','read', contract.paid, roleObj, contract.paid.
          getController(controllers.length - 1))) ){
15                throw new Error('access denied...')
16              }
17        contract.paid.happen(event)
18
19    // Omitted the non-access control related code for clarity
20  }
```

Listing 9.14: Delivered event transaction with second security layer, in JavaScript.

9.4 SymboleoAC2SC Code Generator

An existing code generator, named Symboleo2SC and written in Xtend [20], generates executable smart contracts for the HyperLedger Fabric platform from Symboleo specifications [101]. We have extended this tool to support the new access control concepts found in SymboleoAC, resulting in a new SymboleoAC2SC code generator. This new tool generates code that exploits the extended SymboleoACJS library developed in Section 9.1, and that interacts with the SymboleoAC API.

There are two important considerations for the conversion here: default security settings (i.e., controller and pre-authorization rules, discussed in Section 7.6) and the design-time access control rules and policies defined explicitly in a SymboleoAC specification.

This section presents the procedures used to generate *secure* smart contract code for each of the main ontological concepts supported by the SymboleoAC language (Figure 7.1), mainly as extensions of Rasti's work [101], with code transformations detailed using Xtend.

9.4.1 Domain Model Classes

The code generation for these classes has been updated to include the default controller as a parameter, which is propagated through the entire class hierarchy to the `Resource` core class. In this core class, the `hasPermission()` function is utilized to determine whether to add the passed controller to the controller list of the created resource. Also, all related domain attributes are initialized as `Attribute` instances, with their name, value, and the default controller that is passed to the `Resource` class to be added to their controller list. Listing 9.15 shows the Xtend source code generation for all domain *events* using the `generateEvent()` function. As explained in Section 7.6, the default controller for any domain event is its performer. In line 9, all domain attributes are passed as parameters to the domain event class. Notice that the *name* and *performer* of the event are mandatory.

In line 10, only the performer is passed to the super class to be assigned as the controller. The name is assigned to the *_name* variable, while the domain attributes are initialized as `Attributes` with three parameters: name, value, and controller (lines 12-15). This setup

ensures that each attribute has its own controller, while the controller of the containing class is its default controller.

```
def void generateEvent(IFileSystemAccess2 fsa, Model model, RegularType event) {
    val isBase = event.ontologyType !== null

    if (isBase === true) {
      val code = '''
        const { Event } = require(<<EVENT_CLASS_IMPORT_PATH>>);
        const { Attribute } = require(<<ATTRIBUTE_CLASS_IMPORT_PATH>>);
        class <<event.name>> extends Event {
          constructor(_name,performer,<<event.attributes.filter[Attribute a | a.name != "performer" && a.name != "
     controller"].map[Attribute a | a.name].join(', ')>> ) {
            super(performer)
            this._name = _name
            <<FOR attribute : event.attributes>>
             <<IF (attribute.name !='performer' && attribute.name !='controller')>>
            this.<<attribute.name>> = new Attribute("<<attribute.name>>",<<attribute.name>>, performer)
            <<ENDIF>>
            <<ENDFOR>>
          }
        }

        module.exports.<<event.name>> = <<event.name>>
      '''
      fsa.generateFile("./" + model.contractName + "/domain/events/" + event.name + ".js", code)
    } else ...// The rest of the code generates a class hierarchy, including domain events that are subtypes of other
     domain events with the same updates we made on the above class.
  }
```

Listing 9.15: Xtend source code of part of the improved `generateEvent()` function.

9.4.2 Contract Class

As the contract class is the first class called to initiate the generated smart contract, most of the default settings are initiated from there. All the declared attributes and variables (roles, assets, events, and their attributes) are instantiated using their suitable core type according to their *declarations* in the SymboleoAC specification. The default and specified controllers will be parsed and passed through the instance creation to the defined classes or added using the `addController()` function, as shown in Listing 9.16, lines (7-22). The contract controllers are the contractor's parties and are assigned in lines (25-28).

```
// Omitted code -- importing the necessary libraries
class <<model.contractName>> extends SymboleoContract {
  constructor(<<model.parameters.map[Parameter p | p.name].join(',')>>) {
    super("<<model.contractName>>")

    // assign variables of the contract
    <<FOR variable : model.variables>>
      <<IF variable.type instanceof RegularType>>
        this.<<variable.name>> = new <<variable.type.name>>("<<variable.name>>")

        <<FOR assignment : variable.attributes>>
          <<IF assignment instanceof AssignExpression>>
            <<IF assignment.name != 'controller' && assignment.name != 'performer'>>
              this.<<variable.name>>.<<assignment.name>>._value =
                <<generateExpressionString(assignment.value, 'this')>>
            <<ELSE>>
              <<IF assignment.name == 'controller'>>
                this.<<variable.name>>.addConroller(
                  <<generateExpressionString(assignment.value, 'this')>>
                )
              <<ENDIF>>
            <<ENDIF>>
          <<ENDIF>>
        <<ENDFOR>>

      <<ENDIF>>
    <<ENDFOR>>

```

```
29      this.aCPolicy = new ACPolicy(<<getDefaultControllerACPolicy(model)>>)
30
31      <<FOR variable : model.variables>>
32        <<IF findRole(variable.type.name)>>
33          this.addController(this.<<variable.name>>)
34        <<ENDIF>>
35      <<ENDFOR>>
36
37      // create instance of triggered obligations
38      <<FOR obligation : triggeredObligations>>
39        this.<<obligation.name>>Situation = new LegalSituation()
40
41        <<IF !(obligation.consequent instanceof PAtomPredicateTrueLiteral)>>
42          <<generateLegalpositionCondition(
43              obligation.consequent,
44              "this." + obligation.name + "Situation.addConsequentOf("
45          )>>
46        <<ENDIF>>
47
48        <<IF !(obligation.antecedent instanceof PAtomPredicateTrueLiteral)>>
49          <<generateLegalpositionCondition(
50              obligation.antecedent,
51              "this." + obligation.name + "Situation.addAntecedentOf("
52          )>>
53        <<ENDIF>>
54
55        this.obligations.<<obligation.name>> =
56          new Obligation(
57            '<<obligation.name>>',
58            <<generateDotExpressionString(obligation.creditor, 'this')>>,
59            <<generateDotExpressionString(obligation.debtor, 'this')>>,
60            this,
61            this.<<obligation.name>>Situation
62          )
63
64        <<getSpecifiedControllerObligation(obligation)>>
65
66      <<ENDFOR>>
67
68      // omitted code ...
69
70      <<FOR rule : model.rules>>
71        <<getSpecifiedRulesUnCond(rule)>>
72      <<ENDFOR>>
73    }
74  }
```

Listing 9.16: Xtend source code of part of the improved `compileContract()` function.

9.4.3 LegalPosition Class

According to the pre-authorization rules (Section 7.6) for powers, the default controller is the *creditor*, while for obligations, the default controller is the *debtor*. This configuration is incorporated into the obligation and power classes, where the obligation class sends its debtor to the super class `LegalPosition`, and the power class sends its creditor to the super class as well. These controllers are then passed to the `Resource` class and assigned as controllers. Lines from 30 to 33 in Listing 9.16 show how the instances of unconditional obligations are created and their parameters are passed. However, the function `getSpecifiedControllerObligation(obligation)` (line 64) is utilized to add the controller specified by SymboleoAC as shown in Listing 7.1, line 22.

9.4.4 Access Control Policy (ACPolicy) Class

Like for resources, the initialization controllers (contract parties) are passed to the `Resource` class where they are added to its controller using the method `getDefaultControlerACPolicy()`, as shown in Listing 9.16, line 29.

9.4.5 Access Control Rules

The rules defined in a SymboleoAC specification (e.g., Listing 7.1, lines 31–38) are extracted, and corresponding instances of the `Rule` class are created using the method `getSpecifiedRulesUncond(rule)`, as shown in Listing 9.17 (lines 3–5). This method (Listing 9.18) extracts the role, the provided permission, and the accessed resource from the rule specification, and then passes them to the `addRulee()` method. This method also ensures the validity of the rule before adding it to the policy's rules list. It checks whether the role already has this permission on the specified resource (i.e., has pre-authorization rights or the rule already exists). If the rule is new, it is created and added to the policy. As shown in Listing 9.18, the method checks the type of the accessed resource and sends the corresponding object type to the `addRulee()` function. This method covers all the declared elements and the *unconditional* legal positions.

```
1  class <<model.contractName>> extends SymboleoContract {
2      // ... omitted code
3      <<FOR rule : model.rules>>
4          <<getSpecifiedRulesUnCond(rule)>>
5      <<ENDFOR>>
```

Listing 9.17: Xtend source code of part of the improved `compileContract()` function.

```
1  def String getSpecifiedRulesUnCond(Rule rule){
2      var String addRule=""
3      var Permission contr = rule.getPermission()
4      var Ref rl=rule.getAccessedRole()
5      var String rName=""
6      if (rule.accessedResource instanceof ResourceObligation) {
7          val obr = rule.accessedResource as ResourceObligation
8          val obl = obr.resourceOp as Obligation
9          rName = "obligations."+obl.name
10         if (obl.trigger === null){
11             addRule=addRule+"this.aCPolicy.addRulee("+contr.name+", "+rName+", this."
12                     +(rl as VariableRef).variable+")\n"
13         }
14     }
15     if (rule.accessedResource instanceof ResourceDot) {
16         val resourceDot = rule.accessedResource as ResourceDot
17         val  r = resourceDot.resourceDot
18         rName =generateDotExpressionString(r, "this")
19         addRule=addRule+"this.aCPolicy.addRulee("+contr.name+", "+rName+", this."
20                 +(rl as VariableRef).variable+")\n"
21     }
22     return addRule
23 }
```

Listing 9.18: New Xtend function `getSpecifiedRulesUnCond()`.

9.4.6 Conditional LegalPosition

The *conditional* legal positions (obligations and powers), which are initiated whenever the associated logical expression evaluates to true, are created within the listener functions rather than in the constructor of the domain contract. Consequently, the related controllers and rules are added after their creation using the `getSpecifiedRulesCondObligation()` and `getSpecifiedRulesCondPower()` functions, described respectively in Listings 9.19 and 9.20.

```
1  def String getSpecifiedRulesCondObligation(Obligation oblC, Model model) {
2      var String addRule = ""
3
4      for (Rule rule : model.rules) {
```

```
        var Permission contr = rule.getPermission()
        var Ref rl = rule.getAccessedRole()
        var String rName = ""

        if (rule.accessedResource instanceof ResourceObligation) {
            val obr = rule.accessedResource as ResourceObligation
            val obl = obr.resourceOp as Obligation
            rName = "obligations." + obl.name

            if ((obl.trigger !== null) && (oblC.name === obl.name)) {
                addRule = addRule + "this.aCPolicy.addRulee(" + contr.name +
                          ", contract." + rName +
                          ", contract." + (rl as VariableRef).variable + ")\n"
            }
        }
    }
    return addRule
}
```

Listing 9.19: New Xtend function `getSpecifiedRulesCondObligation()`.

```
def String getSpecifiedRulesCondPower(Power powerC, Model model) {
    var String addRule = ""

    for (Rule rule : model.rules) {
        var Permission contr = rule.getPermission()
        var Ref rl = rule.getAccessedRole()
        var String rName = ""

        if (rule.accessedResource instanceof ResourcePower) {
            val ResourcePower p = rule.accessedResource as ResourcePower
            val Power power = p.resourcePo as Power
            rName = "powers." + power.name

            if ((power.trigger !== null) && (powerC.name == power.name)) {
                addRule = addRule + "this.aCPolicy.addRulee(" + contr.name +
                          ", contract." + rName +
                          ", contract." + (rl as VariableRef).variable + ")\n"
            }
        }
    }
    return addRule
}
```

Listing 9.20: New Xtend function `getSpecifiedRulesCondPower()`.

9.4.7 New Domain Element – DataTransfer

To process IoT data effectively, a new domain element, DataTransfer, was added to SymboleoAC to support contracts that require IoT integration. This element extends the SymboleoAC ontology class DataTransfer.

Each DataTransfer instance is incorporated into the event model as part of the contract's event hierarchy. The `generateEvent()` method is extended so that it can be invoked for every data transfer specified. For each such specification, a corresponding event class is generated. These generated classes extend the AbstractEvent ontology concept, thereby ensuring semantic consistency with the existing event framework and enabling uniform treatment during runtime execution.

Listing 9.21 presents the Xtend implementation of the improved generateEvent() function.

```
 def void generateEvent(IFileSystemAccess2 fsa, Model model, RegularType event) {
    val isBase = event.ontologyType !== null
    if (isBase === true) {
```

```
      val code = '''
        const { «event.ontologyType.name» } = require(«EVENT_CLASS_IMPORT_PATH»);
        const { Attribute } = require(«ATTRIBUTE_CLASS_IMPORT_PATH»);
        class «event.name» extends «event.ontologyType.name» {
          constructor(_name,performer,«event.attributes.filter[Attribute a | a.name != "performer" && a.name != "
    controller"].map[Attribute a | a.name].join(', ')» ) {
            super(performer)
            this._name = _name
            this._type = "«event.name»"
            «IF((event.ontologyType.name)=="DataTransfer")»
            this.sensorId=_name+"_sensor"+_name + "Rule"
            «ENDIF»
            «FOR attribute : event.attributes»
             «IF (attribute.name !='performer' && attribute.name !='controller')»
             this.«attribute.name» = new Attribute("«attribute.name»",«attribute.name»)
            «ENDIF»
            «ENDFOR»
          }
        }

        module.exports.«event.name» = «event.name»
      '''
      fsa.generateFile("./" + model.contractName + "/domain/events/"+((event.ontologyType.name)=="DataTransfer"?"
    datatransfer/":"")+ event.name + ".js", code)

}
```

Listing 9.21: Xtend source code of the improved `generateEvent()` function handling DataTransfer objects

9.4.8 Assignment Expression

The `generateOAssignObjectString()` function is tasked with creating JavaScript expressions that map to assignment expressions specified using the SymboleoAC language. The `AssignVar` array holds variables whose values are updated dynamically during runtime using the newly implemented `Assign()` and `HappensAssign()` functions. The function's for-loop iterates through these variables to ensure that the most current values are retrieved after the contract has been initiated.

```
def String generateOAssignObjectString(List<OAssignment> a) {
   var s = ""
   var eName=""
   var found=false
   for(e: a){
     found=false
      if (e instanceof OAssignExpression){
        eName=generateDotExpressionString(e.name2,"" )
        if (!(AssignVar.contains(eName) )) {
         for( p : parameters){

                  if (p.name.toString()==eName.toString()){found=true}
                }
              if (!(found)){AssignVar.add(eName) }

        }
          s= s+generateDotExpressionString(e.name2,"contract" )+ " = " + generateExpressionString(e.value, 'contract')
         s=s+" \n"
         }
       }
   return s
 }
```

Listing 9.22: New Xtend function `generateOAssignObjectString()`.

9.4.9 LegalSituation – Antecedent and Consequent

As outlined in Section 7.6.2, the debtor of an obligation, namely the *performer* and the *liable* party, is pre-authorized to access all informational parts of that obligation, including its antecedent, consequent, and any events that determine its status. Similarly, for a power, the creditor, represented by the *performer* and the *rightHolder*, is pre-authorized to access all informational parts of that power.

To ensure that each role accesses only the informational parts permitted by the access control policy, we generate the antecedent and consequent of each LegalSituation instantiated at contract creation time. This ensures that only authorized roles can view or interact with the informational parts, which may be one of three types: *state condition*, *condition*, or *event condition*.

Listing 9.23 presents the improved `compileContract` function, showing the instantiation of `LegalSituation` and the addition of `addAntecedentOf()` and `addConsequentOf()` for these three categories of informational parts (see, e.g., lines 6, 8, and 25).

The antecedent and consequent of a conditional legal position are illustrated in Appendix A.

```
1  def void compileContract(IFileSystemAccess2 fsa, Model model) {
2  // ... omitted code
3
4    // create instance of triggered obligations
5           «FOR obligation : triggeredObligations»
6                 this.«obligation.name»Situation = new LegalSituation();
7               «IF !(obligation.consequent instanceof PAtomPredicateTrueLiteral)»
8                «generateLegalpositionCondition(obligation.consequent,"this."+obligation.name+"Situation.addConsequentOf(
        ")»
9               «ENDIF»
10              «IF !(obligation.antecedent instanceof PAtomPredicateTrueLiteral)»
11               «generateLegalpositionCondition(obligation.antecedent,"this."+obligation.name+"Situation.addAntecedentOf(
        ")»
12              «ENDIF»
13                this.obligations.«obligation.name» = new Obligation('«obligation.name»', «generateDotExpressionString(
        obligation.creditor, 'this')», «generateDotExpressionString(obligation.debtor, 'this')», this, this.«obligation.
        name»Situation)
14                «getSpecifiedControllerObligation(obligation)»
15          «ENDFOR»
16
17          «FOR obligation : triggeredSurvivingObligations»
18                this.survivingObligations.«obligation.name» = new Obligation('«obligation.name»', «
        generateDotExpressionString(obligation.creditor, 'this')», «generateDotExpressionString(obligation.debtor, 'this')»
        , this, true,this.«obligation.name»Situation)
19                «getSpecifiedControllerObligation(obligation)»
20          «ENDFOR»
21
22          «FOR power : triggeredPowers»
23              this.«power.name»Situation = new LegalSituation();
24           «IF !(power.antecedent instanceof PAtomPredicateTrueLiteral)»
25               «generateLegalpositionCondition(power.antecedent,"this."+power.name+"Situation.addAntecedentOf(")»
26           «ENDIF»
27              this.«power.name»Situation.addConsequentOf({_type: 'stateCondition',«compilePowerCondition(power.
        consequent)»})
28              this.powers.«power.name» = new Power('«power.name»', «generateDotExpressionString(power.creditor, 'this')»
        , «generateDotExpressionString(power.debtor, 'this')», this,this.«power.name»Situation)
29              «getSpecifiedControllerPower(power)»
30          «ENDFOR»
31
32 // ... omitted code
33
34 }
```

Listing 9.23: Xtend source code of part of the improved `compileContract()` function

9.4.10 Serializer and Deserializer

The state of the contract instance, along with all related objects (including legal positions, events, roles, and assets), is stored on the ledger as a serialized object. Since smart contracts are stateless by design, this state must be retrieved each time a transaction is invoked, as the contract's code is only executed during transaction calls. The methods `stub.putState(key , value)` and `stub.getState(key)` handle these state updates and queries. The key serves as the unique identifier for the JavaScript contract instance, while the value is a JSON string that encodes the complete state of the contract instance. Applying SymboleoAC's security model leads to revise the two methods for serializing and deserializing contract objects to and from JSON strings. The serializer converts a contract object into a JSON format, while the deserializer reconstructs a full contract object from the JSON string – restoring all related legal positions, events, assets, and roles with their latest updates.

However, extending these objects with the SymboleoAC security model introduces a *circular reference* issue during serialization and deserialization, particularly for roles assigned as controllers of their respective objects and when an object references another object that references the first object. To address this issue, the Flatten library[8] was selected.

While the Flatten library resolves circular dependencies, it introduces a new issue, namely *reference mismatches* for identical objects. In our meat sale running example, if the seller role is i) the controller of the delivered event, ii) the performer of the delivered obligation, iii) the liable party for the payment obligation, iv) assigned new permissions in the access policy rule, and v) the performer of the terminateContract power, the Flatten library creates separate seller objects with identical attributes and values for each reference. As a result, changes made to one instance of the seller object are not reflected in the others. For example, if the seller updates their contact information, this change will not automatically propagate to the other seller instances. Consequently, if the buyer acts as the performer of a notification event that references the seller as the recipient, the buyer may not receive the updated contact details.

This reference mismatch issue highlights the challenge of maintaining consistent state updates when using the Flatten library to handle circular references.

Addressing Reference Mismatches in the Deserializer

To resolve the reference mismatch issue caused by the Flatten library, the deserializer[9] was enhanced with new functionality designed to unify object references across all retrieved elements while providing a generic solution that handles all types of contract objects, minimizing dependency on domain-specific logic.

1. Initialize the Contract Object: Parse the contract object retrieved from the ledger (referred to as the parsed object). Initialize a new contract instance with default

[8] https://www.npmjs.com/package/flatted

[9] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/MeatSale/serializer.js

values based on the design-time specification. This step ensures all core objects (e.g., obligations, powers, roles, events) are properly initialized.

2. Restore Object Attributes: For each attribute in the contract object (excluding legal positions):
   (a) If the attribute is not an object, assign the value from the parsed object to the contract object.
   (b) If the attribute is an object and not a controller, assign the reference from the contract object that matches the identity of the object in the parsed data.
   (c) If the attribute does not exist in the contract, assign the corresponding object from the parsed object to the contract attribute.
3. Restore Obligations and Powers: For each obligation and power in the contract object, rebuild attributes such as state, consequent, antecedent, and event details.
4. Unify List-Based References: Unify the object references and ensure that any run-time updates to these lists are correctly applied.
   (a) For lists of objects such as controller, performer, liable, rules, and rightholders.
      - Remove outdated lists and add the appropriate reference by matching object identities.
      - Iterate through each item in the list.
      - Assign the contract object that matches the identity of the corresponding object in the parsed data.
   (b) For each access policy rule, restore references to the accessed resource (e.g., obligation, power, event, role, etc.)., the accessed role, and the byRole property.

Following these steps, we deeply unify all references in the contract while updating them with the latest changes. Any changes made to referenced objects while running the initiated contract will now propagate across all dependent entities.

9.5 Generation of Smart Contract Transactions – Run-Time

Each SymboleO and SymboleOAC contract specification results in the creation of a corresponding smart contract, with all necessary transactions automatically defined within the `index.js` file [101]. The generated contract builds upon and extends the interface offered by the `fabric-chaincode-node` package[10] for Hyperledger Fabric Node.js.

[10] https://hyperledger.github.io/fabric-chaincode-node/release-2.2/api/

To enforce access control, we integrate permission checks into the generated smart contract. The implementation ensures that only authorized entities can trigger transactions, retrieve contract states, or receive notification.

Some previously generated transactions (e.g., triggering an event) [101] have been extended to include access control, ensuring that only authorized roles can invoke them. In addition, new transactions have been introduced, such as generating notifications. Thus, there are nine additional types of transactions that must be generated:

1. Transactions for triggering a **DataTransfer**, which indicate that a domain data transfer has occurred. These transactions are invoked by the environment through an IoT sensor (Section 9.5.1).

2. Transactions for triggering notifications, which generate notification events whenever a state change occurs due to different types of transactions, including: (1) triggering an event; (2) triggering a data transfer (3) violating an obligation; (4) expiring an obligation; (5) exerting a power; (6) expiring a power; (7) adding a roles policy; (8) retrieving a role policy; and (9) retrieving IoT rules and conditions for export to external applications, where a CEP engine can evaluate incoming IoT data from the message broker (Section 9.5.2) and the remaining transactions (Section B.1).

3. Embedding a two-layer security mechanism into every smart contract transaction, where the first layer performs authentication and the second layer enforces authorization (Section 9.5.3) and the remaining transactions (Section B.2).

4. Transactions for getting IoT sensor rules and conditions ❷, which are used to retrieve data transfer conditions specified in the contract. For all **DataTransfer** declared in the SymboleoAC contract, a transaction is generated to retrieve the associated IoT conditions. This transaction is intended to be invoked by the environment through a trusted third party, namely the SymboleoAC regulator (Section 9.5.4).

5. A transaction that checks whether a specific event has occurred or not, such as whether the event state has "happened" or not. Such transaction will also be governed by access control to ensure that only authorized roles are allowed to query the event state, preventing unauthorized users from accessing this critical information (Section 9.5.5).

6. A transaction that checks the state of a legal position, ensuring that only authorized roles or roles with privileges (such as controller, performer, liable, and rightholder) can query the status of obligations and powers. This transaction will also enforce access control by restricting access based on defined roles (Section 9.5.6).

7. A transaction that evaluates the informational parts of legal positions, including the legal situations (conditions) that encompass Event, DataTransfer, Attribute, and StateTransition. Such transaction ensures that the controller or performer of the corresponding legal position can access only the outcome and its timestamp, rather than the underlying resources themselves, enhancing security and privacy (Section 9.5.7).

8. A transaction for extracting roles from the SymboleoAC contract and storing them in the ledger ❷. Such transaction is invoked by the environment through a trusted third party, such as the regulator or a blockchain administrator (Section 9.5.8).

9. A transaction for retrieving roles from the ledger ❷, which is invoked by the environment through a trusted third party, namely, a regulator or a blockchain administrator, to generate certificates for the list of roles defined in the SymboleoAC contract (Section 9.5.9).

Additionally, five existing transactions are updated to incorporate access control and to generate notifications, namely: 1) triggering an event, 2) violating an obligation, 3) expiring an obligation, 4) exerting a power, and 5) expiring a power (Sections 9.5.3 and 9.5.2).

Note that to improve the efficiency of the generated code, all the added transactions are designed to provide the intended services for all resources of the same type. This optimization improves the scalability of the smart contracts generated and reduces the code size [90, 101].

9.5.1 Transactions for Triggering a Data Transfer

These transactions are invoked when an IoT sensor event happens, resulting to a message received from the environment. A transaction associated with an IoT sensor is used to update the corresponding contract state in the ledger. For each **DataTransfer** declared in a SymboleoAC contract, a corresponding transaction is generated.

For example, in the `MeatSale` case study (see Listing 10.1), the declared data transferred include temperature and humidity. As a result, the names of the generated transactions are derived from the transfer names using the pattern `trigger_{variableName}`, such as `trigger_temperature` and `trigger_humidity`. The Xtend transformation used to generate these data transfer transactions is shown in Listing 9.24.

```
async trigger_«variable.name»(ctx, args) {
          const cid = new ClientIdentity(ctx.stub);
          let roleObj;
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
            const contractState = await ctx.stub.getState(contractId)
            if (contractState == null) {
         return {successful: false}
          }
          const contract = deserialize(contractState.toString())

          const oldMessagesList = []
          oldMessagesList.push(contract.notified.message.slice())
          this.initialize(contract)
          if (contract.isInEffect() «survivEvent(variable.name)» ){
            // First security layer
            try{
                   roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
                   cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

                        if(roleObj === null ){
                         throw new Error('Unauthorized: Unknown access');
                    }

                }catch(err){
                    console.log('access control error: ', err)
                    return { successful: false, message: err.message }
                }// End of first layer
                // Second layer
```

```
            let controllers = contract.«variable.name»._controller
        if(!contract.accessPolicy.hasPermesstion('grant','read', contract.«variable.name»,roleObj, contract.«variable.name».getController(controllers.length - 1)) ||
            !contract.accessPolicy.isValid(new Rule('grant','read', contract.«variable.name», roleObj, contract.«variable.name».getController(controllers.length - 1)) ){
              throw new Error('access denied...')
            } // End of second layer
        contract.«variable.name».happen(event)
        Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent, InternalEventType.contractEvent.Happened, contract.«variable.name»))
         // Notification
            for (const message of contract.notified.message) {
            if (!oldMessagesList[0].includes(message)) {
              this.trigger_notification(ctx, message)
            }
          }

        await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
        return {successful: true}
      } else {
        return {successful: false}
      }
    }
```

Listing 9.24: Xtend source code for generating transactions to trigger DataTransfers.

9.5.2 Transactions for Generating Notifications

A notification transaction is generated for each transaction type in a SymboleoAC smart contract, namely: 1) triggering an event; 2) triggering a data transfer; 3) violating an obligation; 4) expiring an obligation; 5) exerting a power; 6) expiring a power; 7) adding a rule/policy; 8) retrieving a rule/policy; and 9) retrieving IoT rules and conditions.

A generated notification transaction produces a notification event to inform authorized roles about the updated contract state in the ledger. The notification event payload includes information such as the previous state, the current state, timestamps, and the intended recipient roles. Listing 9.25 shows the Xtend source code used to generate notification transactions. Each of the transactions mentioned earlier calls `trigger_notification()`. For example, when an event is triggered, the corresponding transaction invokes it. The logic of `trigger_notification()` for other transactions, such as `violateObligation()`, follows the same structure, except for 5) exerting a power, which is explained below. Listing 9.26 shows the transformation code for one such notification transaction (`violateObligation()`).

```
async trigger_notification(ctx, event) {

        await ctx.stub.setEvent(event.name, Buffer.from(JSON.stringify({
            event: event
          })));

        return {successful: true}
      }
```

Listing 9.25: Xtend source code for generating transactions that produce notifications.

```
async violateObligation_«obligation.name»(ctx, contractId) {
// ... Omitted code

let controllers = contract.obligations.«obligation.name»._controller

if (!contract.accessPolicy.hasPermesstion('grant','read', contract.obligations.«obligation.name», roleObj, contract.«obligation.name».getController(controllers.length - 1)) ||
    !contract.accessPolicy.isValid(new Rule('grant','read', contract.obligations.«obligation.name», roleObj, contract.«obligation.name».getController(controllers.length - 1)))) {
```

```
8       throw new Error('access denied...')
9     }
10
11    let transitionState = contract.«isSurvivingObligation(obligation.name) ? "survivingObligations" : "obligations"».«
        obligation.name».state;
12    if (contract.obligations.«obligation.name».violated()) {
13      await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
14      «val obName="contract."+(isSurvivingObligation(obligation.name) ? "survivingObligations" : "obligations"+"."+
          obligation.name)»
15      // Notify
16      let MSG = transitionState+" Changed to "+«obName».state+","+«obName».name+", " + «obName».contract.id;
17      contract.notified.message.push({name: '«obName»', message: MSG, roles:contract.accessPolicy.permissionValid(«obName»
          ,[«obName».creditor,«obName».debtor],«obName».getController(controllers.length - 1), contract), time: new Date().
          toISOString()})
18      // Notification
19      for (const message of contract.notified.message) {
20        this.trigger_notification(ctx, message)
21      }
22      return {successful: true}
23    } else {
24      return {successful: false}
25    }
26  }
```

Listing 9.26: Improved Xtend source code for an obligation violation transaction that generates notifications.

For all other transaction types, the notification generation process remains the same. However, in the case of contract termination, we iterate through all obligations and powers, updates their states, and generates a corresponding notification for each. The implementation is presented in Appendix B.

These generated notification events are listened to by authenticated external API listeners and published to a message broker, where authorized roles can subscribe to them. For roles that are not pre-authorized through pre-authorization rules (Section 7.6), but are granted permission by SymboleoAC rules or are associated with a resource that is part of a legal position, notifications are limited to querying the state through the Hyperledger Fabric API. To support this, we provide three read only transactions that generate notifications and return the current state of events, legal positions, and legal position parts, as explained in Sections 9.5.5, 9.5.6, and 9.5.7.

9.5.3 Embedding a Two-Layer Security Mechanism in Transactions

As explained earlier in Section 9.3, two security layers are added to every transaction (e.g.,`trigger_delivered`) in the generated smart contracts. There are ten types of transactions that must incorporate these two security layers: 1) Triggering a data transfer; 2) Triggering an event; 3) Violating an obligation; 4) Expiring an obligation; 5) Exerting a power; 6) Expiring a power; 7) Triggering a notification; 8) Adding roles; 9) Getting roles; and 10) Getting IoT sensor conditions.

The only exception is the `init` transaction, which instantiates the smart contract. For this transaction, only the first security layer is required, as it verifies that the entity initiating the contract is the authorized role, namely, the SymboleoAC regulator, and possesses a valid certificate.

Two of these transactions will be used as examples for generating the two security layers using Xtend as shown in Appendix B. The implementation of the remaining transactions is available online[11], as they follow the same pattern. Only the security layers are added and therefore their code is repetitive across transaction types.

Generating the First Security Layer for the `init` Transaction For this transaction, only the first security layer is required, as it is responsible for instantiating the smart contract and assigning its parameters. The SymboleoAC regulator is the only entity authorized to instantiate the smart contract and assign these parameters. Listing 9.27 shows the first security layer for the `init` transaction.

```
async init(ctx, args) {
       const cid = new ClientIdentity(ctx.stub);
       let roleObj;
       const inputs = JSON.parse(args);
       const contractInstance = new «model.contractName» («model.parameters.map[Parameter p | "inputs." + p.name].join(', ')»)
       this.initialize(contractInstance)
       if (contractInstance.activated()) {
         // call trigger transitions for legal positions
         «FOR obligation : triggeredObligations»
           «IF obligation.antecedent instanceof PAtomPredicateTrueLiteral»
             contractInstance.obligations.«obligation.name».trigerredUnconditional()
           «ELSE»
             contractInstance.obligations.«obligation.name».trigerredConditional()
           «ENDIF»
         «ENDFOR»
         «FOR obligation : triggeredSurvivingObligations»
           «IF obligation.antecedent instanceof PAtomPredicateTrueLiteral»
             contractInstance.survivingObligations.«obligation.name».trigerredUnconditional()
           «ELSE»
             contractInstance.survivingObligations.«obligation.name».trigerredConditional()
           «ENDIF»
         «ENDFOR»
         «FOR power : triggeredPowers»
           «IF power.antecedent instanceof PAtomPredicateTrueLiteral»
             contractInstance.powers.«power.name».trigerredUnconditional()
           «ELSE»
             contractInstance.powers.«power.name».trigerredConditional()
           «ENDIF»
         «ENDFOR»
     // First security layer
                 try{
                        roleObj = contractInstance.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
                        cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contractInstance)

                              if(roleObj === null ){
                               throw new Error('Unauthorized: Unknown access');
                         }

                      }catch(err){
                          console.log('access control error: ', err)
                          return { successful: false, message: err.message }
                      }// end of first layer
         await ctx.stub.putState(contractInstance.id, Buffer.from(serialize(contractInstance)))

         return {successful: true, contractId: contractInstance.id}
       } else {
         return {successful: false}
       }
     }
```

Listing 9.27: Xtend source code for a transaction to trigger `init`, with first security layer.

[11] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-IDE/blob/main/ca.uottawa.csmlab.symboleo/src/ca/uottawa/csmlab/symboleo/generator/Symboleo2SC.xtend

9.5.4 Transactions for Getting IoT Sensor Rules and Conditions

Transactions for getting IoT sensor rules and conditions are used to retrieve the DataTransfer condition details defined in a contract specification. For each data transfer declared in a SymboleoAC contract, a corresponding transaction is generated to retrieve the associated IoT conditions, including the sensor id, condition expression, window size, count threshold, and so on. These conditions are exported to external applications, such as a CEP engine ❾ to configure EPL rules, where incoming IoT data from the message broker can be evaluated. This generated information is also used to enroll sensors ❸, generate them ❿, create one queue per role in the message broker ❼, and create users in the broker ❻.This transaction is intended to be invoked by the environment through a trusted third party, namely the SymboleoAC regulator. Listing 9.28 shows the Xtend source code used to generate transactions that retrieve not only the condition specifications, but also the `contractId`, `chaincodeName`, and the intended `chaincodeFunction` that update the contract state. This allows supporting multiple instances of a SymboleoAC contract and enables dynamic invocation of the smart contract transactions affected by incoming IoT sensor data.

```
async getIoTCondition(ctx, contractId) {

                        let roleObj;
                        let contractState = await ctx.stub.getState(contractId)
                        if (contractState == null) {
                          return {successful: false}
                        }
                        const contract = deserialize(contractState.toString())
                        this.initialize(contract)


                        const cid = new ClientIdentity(ctx.stub);
                        const userId = cid.getID();
                        const role = cid.getAttributeValue('HF.role');
                        // First security layer
                        try{
                          if (role !== 'Admin' && role !== 'Regulator') {

                          throw new Error('Only Admin or Regulator can trigger getIoTCondition');
                         }else{
                            roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.
    name'),
                            cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

                                 if(roleObj === null ){
                                  throw new Error('Unauthorized: Unknown access');

                            }
                         }//else
                         }catch(err){
                            console.log('access control error: ', err)
                            return { successful: false, message: err.message }
                          }// End of first layer

                contractState = await ctx.stub.getState(contractId)
                let rules = { rules: [], roles: [] };

                const eventList = [
                  «variables.filter[e | Helpers.getBaseType(e.type).ontologyType.name=='DataTransfer']
                        .map[v | "'" + v.name + "'" ]
                        .join(", ")
                  »
                ];

               for (const key of eventList) {

                  // skip undefined contract entries
                  if (contract[key] === undefined) continue;
                  if (!contract.hasOwnProperty(key)) continue;

                  const dObj = contract[key];

                  rules.rules.push({
                    id: dObj._name + "Rule",
                    contractId: contractId,
                    chaincodeName: "«model.contractName.toLowerCase»",
```

```
                    eventType: "SensorEvent",
                    sensorType: dObj._name,
                    sensorId: dObj.sensorId,
                    condition:(dObj.condition._value.trim() === "")? "": dObj.condition._value,
                    window: (dObj.window._value.trim() === "")? "": "time("+dObj.window._value+ " min)",
                    having: (dObj.count._value.trim() === "")? "":"count(*) > " +dObj.count._value,
                    select: "sensorId, sensorTimestamp"+((dObj.count._value.trim() === "")? ", value " : ", count(*) as
    cnt, avg(value)")+" as avgValue",
                    chaincodeFunction: "trigger_" + dObj._name
                  });
                }
                 // -------------------------------
                 // Build roles list from contract
                 // -------------------------------
                 rules.roles = contract._roles.map(role => role.name._value);

                 // metadata block
                 rules.metadata = {
                   storedBy: cid.getID(),
                   timestamp: new Date().toISOString()
                 };

                              const ruleStr = JSON.stringify(rules);
                              const ruleHash = crypto.createHash('sha256').update(ruleStr).digest();
                              const record = {
                                hash: ruleHash.toString('hex'),
                                rules,
                                verified: true,
                                signer: userId
                              };

              return {
                successful: true,
                message: 'Retrieved successfully',
                record: record
              };

}
```

Listing 9.28: Xtend source code for a transaction to generate IoT sensor rules

9.5.5 Transaction for Retrieving Event Triggered State

For all roles that are granted permission through SymboleoAC rules, we provide a transaction that allows them to query the state of an event in the ledger. Listing 9.29 describes the `getEventDateAndTime()` transaction, which checks whether an event (e.g., the delivered event in the meat sale running example) has occurred and returns the event's timestamp and status. Access to this information is restricted to authorized roles only.

```
// Get date and time of any event
async getEventDateAndTime(ctx, args) {
  const cid = new ClientIdentity(ctx.stub);
  let roleObj;
  const inputs = JSON.parse(args);
  const contractId = inputs.contractId;
  const requiredResource = inputs.event
  let output = {}
  const contractState = await ctx.stub.getState(contractId)
  if (contractState == null) {
    return {successful: false}
  }
  const contract = deserialize(contractState.toString())
  this.initialize(contract)
  try { // First layer
    roleObj = contract.authenticate(
      cid.getAttributeValue('HF.role'),
      cid.getAttributeValue('HF.name'),
      cid.getAttributeValue('organization'),
      cid.getAttributeValue('department'),
      contract
    )
    if (roleObj === null) {
      throw new Error('Unauthorized: Unknown access');
    }
  } catch (err) {
    console.log('access control error: ', err)
    return { successful: false, message: err.message }
```

```
  } // End of first layer
    // Second layer
  let eventObj = contract.findObject(requiredResource.event, requiredResource._type, contract)
  if (eventObj != null) {
    let controllers = eventObj._controller
    if (
      contract.accessPolicy.hasPermesstion(
        'grant','read', eventObj, roleObj, eventObj.getController(controllers.length - 1)
      ) ||
      contract.accessPolicy.hasPermesstionOnLegalPosition(
        'grant','read', eventObj, roleObj, eventObj.getController(controllers.length - 1)
      )
    ) {
      output = {time: eventObj.getHappenedTime(), state: eventObj.hasHappened() ? "Happened" : "Not Happened"}
    } else {
      throw new Error('access denied...')
    }
    return output
  } else {
    throw new Error('The event does not exist...')
  }
}
```

Listing 9.29: Xtend source code of a transaction to get the state and timestamp of an event, with access control.

9.5.6 Transaction for Retrieving Legal Position State

Listing 9.30 shows the `getLegalPositionStateAndTime()` transaction that checks the state and timestamp of a specific legal position within a contract, based on the type of resource (obligation or power) and the state of that resource. The function first checks the superstate of the legal position resource, such as suspension or inEffect. If the resource is suspended, the function returns the suspension time. If the resource is in effect, it checks further states such as active, discharge, create, or violation to return the corresponding time. If the superstate is none of the above, the function defaults to throwing an error for invalid states.

```
async getLegalPositionStateAndTime(ctx, args) {
          const cid = new ClientIdentity(ctx.stub);
                              let roleObj;
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const quiredState = inputs.quiredState.state
          const requiredResource = inputs.quiredState.resource
          const requiredResourceType = inputs.quiredState.resourceType

          let output = {}
          const contractState = await ctx.stub.getState(contractId)
          if (contractState == null) {
            return {successful: false}
          }
          const contract = deserialize(contractState.toString())
          this.initialize(contract)
          try{ // First layer
             roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
             cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

                   if(roleObj === null ){
                    throw new Error('Unauthorized: Unknown access');
                }

          }catch(err){
              console.log('access control error: ', err)
              return { successful: false, message: err.message }
          }// End of first layer
          // Second layer
          const aResource = contract.findLegalPosition(requiredResource, requiredResourceType, contract)
          let controllers = aResource._controller
          if(aResource !== null){
            switch(requiredResourceType.toLowerCase()){
              case 'obligation':
                if(contract.accessPolicy.hasPermesstion('grant','read', aResource, roleObj, aResource.getController(
    controllers.length - 1))) {
                    output= contract.findStateTimeLegalPosition(aResource)
          }else{
```

```
38              throw new Error('access denied...')
39            }
40            break
41            case 'power':
42              if(contract.accessPolicy.hasPermesstion('grant','read', aResource, roleObj, aResource.getController(
      controllers.length - 1))) {
43                  output=contract.findStateTimeLegalPosition(aResource)
44            }else{
45              throw new Error('access denied...')
46            }
47          }
48          } else{throw new Error('Resource does not exist...')}
49          return output
50        }
```

Listing 9.30: Xtend source code of a transaction to get the state and timestamp of a legal position, with access control.

9.5.7 Transaction for Retrieving Legal Position Informational Parts State

As specified in Section 7.6, the debtor of an obligation (i.e., performer and liable) is pre-authorized and has access to the obligation's operations and informational parts (i.e., antecedent, consequent, and events that determine the status of the trigger). Similarly, the creditor (i.e., performer and rightHolder) is pre-authorized and can access the power's operations and all its informational parts. Listing 9.31 shows the `getStateTimeOfParts()` transaction, which retrieves the state and time of informational parts of the legal position, ensuring that only authorized roles can access the data. It handles three types of informational parts: state condition, condition, and event condition.

- **State condition**: Checks whether the informational part is of type `stateCondition`. It verifies whether the user has permission to read the state of the relevant legal position (an obligation or a power). If the resource is in a specific state like suspension or inEffect , the function retrieves the corresponding time (e.g., suspended time or activation time, respectively). If the state is violation or create, the function returns the appropriate time (violation time, creation time, etc.). Access to such data is granted only if the user has the appropriate permissions for the relevant legal position.
- **Condition**: If the informational parts is of type `condition`, the function evaluates a condition expression (using values like `leftSide`, `op`, and `rightSide` from the input). It calculates the value of the condition and returns it. Again, access is allowed only if the user has the necessary permissions on the legal position.
- **Event condition**: For an `eventCondition`, the function checks if the user is authorized to access the event-related data (e.g., delivered or other event states). It verifies whether the event has happened and returns the corresponding time and state (Happened or Not Happened). The access control policy ensures that only authorized users can retrieve the event state.

```
async getStateTimeOfParts(ctx, args){
          const cid = new ClientIdentity(ctx.stub);
                let roleObj;
        const inputs = JSON.parse(args);
        const contractId = inputs.contractId;
        const requiredResource = inputs.condition

        let output = {}

        const contractState = await ctx.stub.getState(contractId)
        if (contractState == null) {
          return {successful: false}
        }
       // First security layer
                try{
                      roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
                      cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

                            if(roleObj === null ){
                             throw new Error('Unauthorized: Unknown access');
                       }

                    }catch(err){
                        console.log('access control error: ', err)
                        return { successful: false, message: err.message }
                    }// End of first layer
                    // Second layer


        const contract = deserialize(contractState.toString())
        this.initialize(contract)
        const aLegalPositionIncodition = contract.findLegalPosition(requiredResource.resource, requiredResource.resourceType, contract)
        let controllers = aLegalPositionIncodition._controller
        if(aLegalPositionIncodition !==null){
           switch(requiredResource._type.toLowerCase()){
             case 'statecondition':
               if(contract.accessPolicy.hasPermesstionOnLegalPosition('grant','read', requiredResource, roleObj, aLegalPositionIncodition.getController(controllers.length - 1),contract)){
                output=contract.findStateTimeLegalPosition(aLegalPositionIncodition)
                 if(output.State !== null && output.State !== undefined ){
                   if (output.State.toLowerCase() !== requiredResource.state.toLowerCase() ) {
                        output = {state: requiredResource.state.toLowerCase()+' is Not Happened', time: null}
                     }
                   }
               } else{
                 throw new Error('access denied...')
               }
             break

             case 'condition':
               if(contract.accessPolicy.hasPermesstionOnLegalPosition('grant','read', requiredResource, roleObj, aLegalPositionIncodition.getController(controllers.length - 1),contract)){
                 let conditionValue = eval('contract.'+requiredResource.leftSide + " " + requiredResource.op + " " + requiredResource.rightSide)
                output = {state: conditionValue, time: null}
               }else{
                     throw new Error('access denied...')
               }
             break

             case 'eventcondition':
               if(contract.accessPolicy.hasPermesstionOnLegalPosition('grant','read', requiredResource, roleObj, aLegalPositionIncodition.getController(controllers.length - 1),contract)){
                 let eventObj = contract.findObject(requiredResource.partResource, requiredResource.partResourceType, contract)
                 output = {time: eventObj.getHappenedTime(), state: eventObj.hasHappened()  ? "Happened" : "Not Happened"}
               }else{
                    throw new Error('access denied...')
               }
             break

             default: throw new Error('This is not a valid part of legal situation...')
           }

        }else {throw new Error('Resource does not exist...')}

        return output

      }
```

Listing 9.31: Xtend source code for a transaction to get LegalPosition Informational Parts state and timestamp, with access control.

Each informational part is handled separately, ensuring that callers can only access the information they are permitted to based on the contract's access control policy. Unauthorized access is prevented by enforcing these permissions at each step.

9.5.8 Transaction for Extracting Roles

A transaction is generated to extract the roles defined in the SymboleoAC specification and store the resulting on-chain role list in the ledger as an `ACPolicyRecord`. The record includes a signed hash computed using the `crypto` class[12], and a tamper-proof event is emitted upon successful storage. This transaction is intended to be invoked by the environment through a trusted third party, such as the SymboleoAC regulator or a blockchain administrator. Listing 9.32 presents a Xtend source code defining how the list of role objects is generated from the SymboleoAC specification.

```
async storeRolesPolicy(ctx, contractId) {
           let roleObj;
           const contractState = await ctx.stub.getState(contractId)
           if (contractState == null) {
             return {successful: false}
           }
           const contract = deserialize(contractState.toString())
           this.initialize(contract)

           //
           const cid = new ClientIdentity(ctx.stub);
           const userId = cid.getID();
           const role = cid.getAttributeValue('HF.role');

           console.log("Attr name")
           console.log(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
               cid.getAttributeValue('organization'), cid.getAttributeValue('department'))

           try{
             if (role !== 'Admin' && role !== 'Regulator') {

             throw new Error('Only Admin or Regulator can trigger roles policy storage');
            }else{
               roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
               cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

                    if(roleObj === null ){
                     throw new Error('Unauthorized: Unknown access');

               }
            } // else
            }catch(err){
               console.log('access control error: ', err)
               return { successful: false, message: err.message }
             } // End of first layer

           // Build roles policy from contract spec
           const policy = {
             roles: contract._roles.map(role => ({
               name: role._name,
               type: role._type,
               dept: role.dept._value,
               org: role.org._value
             })),
             metadata: {
               storedBy: cid.getID(),
               timestamp: new Date().toISOString()
             }
           };

           const policyStr = JSON.stringify(policy);
           const policyHash = crypto.createHash('sha256').update(policyStr).digest();

           const record = {
             hash: policyHash.toString('hex'),
             policy,
             verified: true,
             signer: userId
           };
```

[12] https://nodejs.org/api/crypto.html

```
            await ctx.stub.putState('ACPolicyRecord', Buffer.from(JSON.stringify(record)));

            // Emit tamper-proof event
            await ctx.stub.setEvent('ACPolicyStored', Buffer.from(JSON.stringify({
              accessor: userId,
              role,
              hash: policyHash.toString('hex'),
              time: new Date().toISOString()
            })));

            return {
              successful: true,
              hash: policyHash.toString('hex'),
              message: 'ACPolicy stored successfully with verified signature'
            };
          }
```

Listing 9.32: Xtend source code for a transaction to extract and store roles.

9.5.9 Transaction for Retrieving Roles

A transaction is generated to retrieve the list of roles ❷ defined in the SymboleoAC specification of a contract that was previously hashed and stored in the Hyperledger Fabric ledger. When the smart contract is instantiated, a transaction is called by the environment to access this role list, provided that the caller is authenticated and authorized through Fabric CA and SymboleoAC. In our setting, this access is restricted to the admin and regulator roles only. Listing 9.33 shows the corresponding Xtend source code.

```
async getRolePolicy(ctx, contractId) {

            let roleObj;
            const contractState = await ctx.stub.getState(contractId)
            if (contractState == null) {
              return {successful: false}
            }
            const contract = deserialize(contractState.toString())
            this.initialize(contract)

            // First security layer
            const cid = new ClientIdentity(ctx.stub);
            const userId = cid.getID();
            const role = cid.getAttributeValue('HF.role');

            console.log("Attr name in getPolicy")
            console.log(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
                cid.getAttributeValue('organization'), cid.getAttributeValue('department'))

            try{
              if (role !== 'Admin' && role !== 'Regulator') {

               throw new Error('Only Admin or Regulator can trigger roles policy storage');
             }else{ // Second layer
                roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
                cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

                     if(roleObj === null ){
                      throw new Error('Unauthorized: Unknown access');

                } // End of second layer
             } // else
             }catch(err){
                console.log('access control error: ', err)
                return { successful: false, message: err.message }
               }// End of first layer

            const policyBytes = await ctx.stub.getState('ACPolicyRecord');
            if (!policyBytes || policyBytes.length === 0) {
              return { successful: false, message: 'ACPolicyRecord not found' };
            }

            const policy = JSON.parse(policyBytes.toString());

            // Emit access event for auditing
            await ctx.stub.setEvent('ACPolicyAccessed', Buffer.from(JSON.stringify({
              accessor: userId,
```

```
48              role,
49              time: new Date().toISOString()
50            })));
51
52            return {
53              successful: true,
54              message: 'ACPolicy retrieved successfully',
55              policyRecord: policy
56            };
57          }
```

Listing 9.33: Xtend source code for a transaction to retrieve roles.

9.6 Conclusion

In this chapter, we presented an enriched SymboleoACJS ontology library with utility functions for enforcing access rules, improved code generation capabilities in SymboleoAC2SC, and provided procedures (in Xtend) to create run-time transactions that dynamically apply access control policies. Furthermore, we introduced a two-level security mechanism combining authentication and authorization checks, and optimized serialization techniques to enhance performance. We also introduced a tool (SymboleoAC2SC) that is capable of generating JavaScript-based smart contract code for Hyperledger Fabric from SymboleoAC specifications. This code, which exploits a JavaScript implementation of the extended ontology (SymboleoACJS) is compliant with the implicit and specified access rules, ensuring robust security and seamless integration. In the next chapter, we demonstrate the effectiveness of these tools by evaluating their application to two case studies.

Chapter 10

Evaluation

To evaluate the feasibility of using SymboleoAC2SC in practice, as well as the correctness of the tool's output, this chapter presents different variants of two case studies based on realistic legal contracts. These case studies involve the generation, deployment, and testing of smart contracts within a cyber-physical environment that conforms to the CPSC architecture described in Chapter 6.

Section 10.1 first discusses the complementarity and the coverage of the selected case studies, as well as the complementary nature of the testing and evaluation strategies. Section 10.2 continues with an evaluation of the Meat Sale contract, followed by a COVID-19 Vaccine Procurement contract in Section 10.3 that is used as a second case study targeting generalization to another contractual domain. Then, Section 10.4 presents the results related to the correct enrollment of the major components of our SymboleoAC architecture (Figure 5.1), namely users, sensors, the CEP engine, and the message broker, as well as the results of notification delivery from the smart contract and the full-cycle execution of the CPSC architecture. Section 10.5 presents an evaluation of additional correctness concerns by running multiple instances of the same contract, as well as multiple instances of multiple different contracts, with shared parties that have different access rights at design time across different instances. Lastly, Section 10.6 concludes the evaluation.

10.1 Coverage and Complementarity of Case Studies

The two case studies were intentionally selected to provide complementary coverage of SymboleoAC framework features and execution scenarios. The Meat Sale contract represents a comparatively simpler cyber-physical smart contract scenario, with a smaller number of events, legal positions, legal situations, run-time interactions, and IoT devices compared to the Vaccine Procurement contract. On the other hand, the Meat Sale case study introduces more complex data structures (e.g., lines 15 and 16 in Listing 10.1) compared to the Vaccine Procurement contract.

In contrast, the Vaccine Procurement contract introduces more complex contractual behavior and richer data interactions. It includes surviving obligations, assignment expres-

sions, multiple concurrent IoT monitoring streams, multiple instances of legal positions, and more advanced execution workflows and state transitions.

Together, the two case studies provide complementary coverage of the SymboleoAC language, code generation process, access control mechanisms, IoT integration, and event-driven execution behavior in CPSCs. Table 10.1 summarizes the main coverage differences between the two case studies.

Table 10.1: Coverage comparison between the Meat Sale and Vaccine Procurement case studies

Feature	**Meat Sale**	**Vaccine Procurement**
Basic legal positions	✓	✓
Multiple instances of legal positions	×	✓
Access control rules	✓	✓
Controller rules	✓	✓
Preauthorization rules	✓	✓
DataTransfer	✓	✓
EPL rule variations (CEP)	Moderate	High
Multiple sensor types	Limited	Extensive
Assignment expressions	×	✓
Surviving obligations	×	✓
Environmental variables	×	✓
Complex data structures	✓	×
Notification generation	✓	✓
Multiple third parties	✓	✓
Event dependencies	Moderate	High

This thesis distinguishes between *testing* and *evaluation* of its contributed artifacts.

- Testing primarily focused on checking the correctness of the generated code, runtime behavior, and access control enforcement at the implementation level. For that purpose, a comprehensive set of unit tests was developed and checked. In total, 57 tests were performed for the Meat Sale case study and 44 more for the Vaccine Procurement case study, covering authorized access, unauthorized access, and exception scenarios. These tests were designed to validate both functional correctness and security enforcement, particularly ensuring that access control policies are correctly enforced under different conditions. In addition, 80 unit tests were conducted on the SymboleoACJS library, generated smart contract transactions, and integration components to validate expected execution behavior under different scenarios.

- In contrast, the evaluation focused on assessing the technical feasibility and applicability of the overall SymboleoAC framework within cyber-physical smart contract environments. This evaluation was conducted through two complementary case studies and multiple instances of the two case studies with different parameters and shared parties, involving end-to-end execution, IoT integration, event-driven interactions, access control enforcement, CEP processing, notification delivery, and deployment across multiple architectural components.

10.2 Meat Sale Contract Case Study

This section introduces the first case study, based on a real contract first explored by Sharifi [110], and then enhanced by Rasti [99]. The natural-language clauses of the contract have already been presented in Table 2.1. We wrote an extended SymboleoAC version of that specification (see Listing 10.1) in the SymboleoAC IDE. This new version goes beyond the Symboleo version from Listing 2.1 by supporting access control concepts and data transfers coming from IoT devices in the cyber-physical environment.

```
Domain meatSaleDomain
  //controller by default is the role itself
  Seller isA Role with returnAddress: String, name: String, org: String, dept: String;
  Buyer isA Role with name: String, warehouse: String, org: String, dept: String;
  //thirdParty is added to differentiate thirdParty role from contracting parties
  TransportCo isA Role thirdParty with name:String, org: String, dept: String;
  Assessor isA Role thirdParty with name: String, org: String, dept: String;
  Regulator isA Role thirdParty with name: String, org: String, dept: String;
  Storage isA Role thirdParty with name: String, address: String, org: String, dept: String;
  Shipper isA Role thirdParty with name: String, org: String, dept: String;
  Admin isA Role thirdParty with name: String, org: String, dept: String;
  Currency isAn Enumeration(CAD, USD, EUR);
  MeatQuality isAn Enumeration(PRIME, AAA, AA, A);
  //by default controller of an asset is its owner
  PerishableGood isAn Asset with quantity: Number, quality: MeatQuality, barcode:String, owner: Seller;
  Meat isA PerishableGood;
  //TransportCo assigns as the performer (instead of the default value)
  Delivered isAn Event with deliveryAddress: String, delDueDate: Date, performer: TransportCo, controller: Seller;
  Paid isAn Event with amount: Number, currency: Currency, from: Buyer, to: Seller, payDueDate: Date, performer: Buyer
      ;
  PaidLate isAn Event with amount: Number, currency: Currency, from: Buyer, to: Seller, performer: Buyer;
  InspectedQuality isAn Event with Env quantityFound: Number, Env qualityFound:MeatQuality, Env barFound: String,
     performer:Assessor;
  Alert isA DataTransfer with Env value: Number, Env sensorTimestamp:String, condition: String, window: String, count:
       String, controller:Seller, performer:Regulator;
  PasswordNotification isAn Event with Env pin:String, performer:TransportCo;
  UnLoaded isAn Event with performer:Assessor;
endDomain

Contract MeatSale (buyerP : Buyer, sellerP : Seller, transportCoP : TransportCo, assessorP : Assessor, regulatorP:
     Regulator, storageP: Storage, shipperP: Shipper, adminP: Admin, barcodeP : String, qnt : Number, qlt :
     MeatQuality, amt : Number, curr : Currency, payDueDate: Date,
  delAdd : String, effDate : Date, delDueDateDays : Number, interestRate: Number
)

Declarations
  seller: Seller with name:= sellerP.name, returnAddress := sellerP.returnAddress, org:= sellerP.org, dept:= sellerP.
     dept;
  buyer: Buyer with name:=buyerP.name, warehouse:= buyerP.warehouse, org:= buyerP.org, dept:= buyerP.dept;
  transportCo: TransportCo with name:= transportCoP.name, org:= transportCoP.org, dept:= transportCoP.dept;
  assessor: Assessor with name:= assessorP.name, org:= assessorP.org, dept:= assessorP.dept;
  regulator: Regulator with name:= regulatorP.name, org:= regulatorP.org, dept:= regulatorP.dept;
  storage: Storage with name:= storageP.name, address:= storageP.address, org:= storageP.org, dept:= storageP.dept;
  shipper: Shipper with name:= shipperP.name, org:= shipperP.org, dept:= shipperP.dept;
  admin: Admin with name:= adminP.name, org:= adminP.org, dept:= adminP.dept;
  goods: Meat with quantity := qnt, quality := qlt, barcode:=barcodeP, owner:= seller;
  delivered: Delivered with deliveryAddress := delAdd, delDueDate := Date.add(effDate, delDueDateDays, days),
     performer:= transportCo, controller:= seller;
  paidLate: PaidLate with amount := (1 + interestRate / 100) * amt, currency := curr, from := buyer, to := seller,
     performer:= buyer;
  paid: Paid with amount := amt, currency := curr, from := buyer, to := seller, payDueDate := payDueDate, performer:=
     buyer;
  temperature: Alert with condition:= "value > 2", window:="10", count:="1", controller:=seller, performer:=regulator;
  humidity: Alert with condition:= "value < 85 OR value > 90", window:="10", count:="1", controller:=seller, performer
     :=regulator;
  passwordNotification: PasswordNotification  with performer:=transportCo;
  inspectedQuality:InspectedQuality with performer:=assessor;
  unLoaded:UnLoaded with performer:=assessor;

Obligations
  // controller by default is the debtor (seller) of obligation delivery
  delivery: Obligation(seller, buyer, true, WhappensBefore(delivered, delivered.delDueDate) and  delivered.
     deliveryAddress == buyer.warehouse and not Happens(temperature) and not Happens(humidity) );
  inspectMeat: Happens(delivered) -> Obligation(assessor, buyer, Happens(passwordNotification), Happens(
     inspectedQuality) and inspectedQuality.barFound == goods.barcode and inspectedQuality.qualityFound == goods.
     quality  and inspectedQuality.quantityFound == goods.quantity);
  payment: O(buyer, seller , Happens(unLoaded), WhappensBefore(paid, paid.payDueDate));
  latePayment: Happens(Violated(obligations.payment)) -> O(buyer, seller, true, Happens(paidLate));

Powers
  // controller by default is the creditor of power suspendDelivery i.e,seller
  suspendDelivery : Happens(Violated(obligations.payment)) -> Power(seller, buyer, true, Suspended(obligations.
     delivery)) with Controller seller;
  resumeDelivery: HappensWithin(paidLate, Suspension(obligations.delivery)) -> P(buyer, seller, true, Resumed(
```

```
      obligations.delivery));
61   terminateContract: Happens(Violated(obligations.delivery)) -> P(buyer, seller, true, Terminated(self));
62
63 //access policy that contain list of rules to Grant/Revoke permission
64 ACPolicy with Controller seller //controller of policy are the regulator who can override rules and pre-authrization
       rules
65  Rule1: Grant read To buyer On goods.quantity by seller; //access to specific asset attribute
66  Rule2: Grant read To assessor  On obligations.delivery by seller; //access to an obligation
67  Rule3: Grant read To transportCo On inspectedQuality by assessor;
68  Rule4: Grant read To seller On inspectedQuality by assessor;
69  Rule5: Grant write To assessor On inspectedQuality by seller;
70  Rule6: Grant write To transportCo On powers.suspendDelivery by seller;
71  Rule7: Grant write To transportCo On powers.resumeDelivery by seller;
72  Rule8: Revoke read To buyer On goods.quality by seller;
73  Rule9: Grant read To buyer On temperature.value by seller;
74
75  Constraints
76   not(IsEqual(buyer, seller));
77
78 endContract
```

Listing 10.1: Meat Sale contract specification in SymboleoAC, adapted from [89, 110]

This meat sale contract specifies the roles of a seller, a buyer, and several third parties, including a transport company, an assessor, and a regulator. The seller is responsible for delivering the goods, which are perishable items such as meat. The contract outlines obligations, such as the seller's duty to deliver the goods under appropriate temperature and humidity conditions and the buyer's obligation to pay in a timely way. Delivery is performed by the transport company and is monitored by the seller. If the buyer violates the payment obligation, the seller has the power to suspend delivery. The contract also includes provisions for late payment and resumption of delivery. Additionally, the assessor inspects the quality of the meat, and both the transport company and assessor handle events related to the transport and quality inspection processes. The perishable goods are transported under strict conditions monitored by IoT devices. These devices continuously track critical data throughout the transportation process.

For our case study, the environmental values specified in the SymboleoAC specification for cold-chain logistics and other types of food logistics are defined in accordance with the guidelines and food safety regulations issued by the Canadian Food Inspection Agency (CFIA) and Health Canada[1]:

- Temperature sensors monitor that the goods remain within acceptable temperature limits. For dried meat, the temperature must remain at or below 2 °C. An alert is triggered if this threshold is exceeded.
- Humidity sensors monitor that the humidity level remains between 85% and 90% in the case of transporting dried meat. An alert is triggered if the humidity falls below 85% or exceeds 90%.

We evaluated our SymboleoAC approach and tools using different variants of the Meat Sale contract in order to assess features that are not present in the original version. The main changes are summarized as follows:

[1] https://inspection.canada.ca/en/food-guidance-commodity/meat-products-and-food-animals, https://inspection.canada.ca/en/food-safety-industry/food-safety-standards-guidelines, https://www.canada.ca/en/health-canada/services/food-safety.html

- **Mandatory administrative and role attributes.** An Admin role is introduced as a mandatory role in the specification. In addition, for each role, the attributes dept, org, and name are mandatory. These attributes support the first layer of security (see Section 9.3), as they are also required and compared at the SymboleoAC API level.
- **Introduction of third-party roles.** A thirdParty attribute is added to distinguish external actors from the main contracting parties.
- **Extended set of roles.** Additional third-party roles, such as TransportCo, Storage, and Shipper, are introduced to test the expressiveness of the contract and the access control model. These roles allow different performers to be assigned to different resources, beyond the original Seller and Buyer, enabling a more realistic evaluation of fine-grained access control.
- **Roles treated as first class objects.** Unlike Rasti's Symboleo2SC compiler [101], SymboleoACJS now treats roles from the **Declarations** section as explicit objects.
- **Extended legal positions.** In addition to conditional powers, we introduce *unconditional* and *triggered* powers. The original contract included only conditional powers with trigger conditions. This extension allows for different types of antecedents and consequents, enabling a more comprehensive evaluation of the access control policies in SymboleoAC. In particular, it allows the controller/performer of a legal position to access conditions that are part of it.

The contract also features an access control policy (**ACPolicy**) with rules that govern how various roles can access and modify specific aspects of the contract. For example, the assessor has write access to the inspection quality attributes of the meat. The regulator acts as the controller of the access control policy, overseeing these rules and having the authority to override, at run-time, the access control permissions set by other resource controllers.

10.2.1 Results – SymboleoAC2SC Compiler

Our evaluation focuses on the feasibility of the conversion from SymboleoAC to executable smart contracts with access control. The SymboleoAC IDE, implemented using Xtext, executes the code generation and produces the corresponding smart contract files. As explained in Sections 9.3 and 9.4, the generated contracts successfully incorporate two security layers, implicit and explicit rules, the new domain elements, and implement the required transactions.

For the meat sale contract, Listing 10.2 shows the generation results for the variables associated with the event Delivered. These variables, of the domain type **Event**, are generated as instances of the `Attribute` ontology class from the SymboleoACJS library. Observe that the performer in this context is mandatory and must be assigned. This assignment is sent to the superclass (i.e., `Resource`) to designate the performer as the controller for that event. Each of the classes of the domain types such as **Asset**, **Role**, **DataTransfer** is generated in the same way.

```
const { Event } = require("symboleo-js-core");
const { Attribute } = require("symboleo-js-core");
class Delivered extends Event {
  constructor(_name,performer,deliveryAddress, delDueDate ) {
    super(performer)
    this._name = _name
    this.deliveryAddress = new Attribute("deliveryAddress",deliveryAddress)
    this.delDueDate = new Attribute("delDueDate",delDueDate)
  }
}
module.exports.Delivered = Delivered
```

Listing 10.2: New source code of generated Events in JavaScript

The contract class (`MeatSale`, Listing C.1 in Appendix C) is generated correctly by SymboleoAC2SC, specifying the controller of each resource (contract, event, asset, role, obligation, power, policy, and so on), along with a list of specified rules as explained in Section 9.4.5. All domain classes variables are generated as instances of the `Attribute` ontology class (as mentioned above for events as an example), and they are available online[2].

For obligations and powers without any antecedent condition (triggeredUnconditional), access rules are applied when the contract is instantiated, as explained in Section 9.4.5, Also, we instantiated all unconditional obligations and powers as legal situations, as explained in Section 9.4.9, and added all antecedents and consequents covering the different types of informational parts (lines 151–165). On the other hand, for obligations and powers with antecedent conditions (triggeredConditional), the rules are applied when they are triggered as explained in Section 9.4.6 and are reflected in the event subscriptions and listeners (`event.js`), as explained below. Similarly, legal situations (i.e., antecedents and consequents) of conditional legal positions are applied when obligations and powers are triggered and are reflected in the event subscriptions and listeners (`event.js`) as well. Listing C.1 shows the MeatSale contract with specified controllers (lines 36, 44, 51, etc.) and specified rules (lines 169–175) for each resource.

The `index.js` file, which contains the main entry point for Hyperledger Fabric and implements all transactions described in Section 9.5, is also generated successfully. The generated smart contract index file (29 transactions, 1704 lines) is available online[3].

The event subscriptions and listeners (`events.js`) were also generated successfully after multiple iterations of refining the code generation logic. In particular, we tested different variants of conditional obligations and powers, where the corresponding rules are instantiated only when the triggering condition occurs, as discussed in Section 9.4.6. We also evaluated several variants of antecedents and consequents of conditional legal position, as explained in Appendix A, involving **DataTransfer** as well as informational parts that appear in the antecedents and consequents of conditional legal positions, as discussed in Section 9.5.7. The complete file with event subscriptions and listeners is available online[4].

[2] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/tree/main/MeatSale/domain

[3] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/MeatSale/index.js

[4] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/MeatSal

Also, the deserialization method discussed in Section 9.4.10 was generated successfully. The complete `serializer.js` file is available online[5].

Finally, since SymboleoACJS targets the Node.js runtime and communicates with the SymboleoAC API, we generated a `package.json` file in which the SymboleoACJS library is declared as a dependency, along with all necessary packages required to run the chaincode in Hyperledger Fabric and deploy the generated smart contract. That file is also available online[6].

10.2.2 Results – Validation Scenarios

We conducted extensive unit testing using Mocha[7] and Chai[8] to validate the generated JavaScript code across a wide range of possible scenarios. In particular, we tested assigning controllers and performers through both explicit and implicit mechanisms, and evaluated multiple combinations of access control rules and constraints. This included comprehensive assessments of pre-authorization, granting and revoking permissions, and resolving rule conflicts during contract execution.

At run-time, for policy evaluation and enforcement within smart contract transactions, SymboleoAC2SC embeds two security checks (i.e., certificate-based and access permission) as part of every transaction to prevent unauthorized access. Listing 10.3 (lines 18–32) shows the transaction method that triggers the delivered event, with access control. This event can only be generated by its performer (`hasPermission()`) or by a role that has write access (`isValid()`). The example from Listing 10.3 was successfully tested with unit tests to ensure its validity and compliance with the access control policy. Two scenarios for the delivered event are considered: i) the event must be triggered/generated by an authorized role only, and ii) an unauthorized role or attacker is trying (unsuccessfully) to generate that event.

```
1  async trigger_delivered(ctx, args) {
2      let roleObj;
3      const inputs = JSON.parse(args);
4      const contractId = inputs.contractId;
5      const event = inputs.event;
6        const contractState = await ctx.stub.getState(contractId)
7        if (contractState == null) {
8             return {successful: false}
9      }
10     const contract = deserialize(contractState.toString())
11     //... omitted code
12     this.initialize(contract)
13
14     if (contract.isInEffect()  ){
15       // First security layer
16       try{
17               // Used in unit tests to retrieve the role instead of a real Fabric certificate.
18               roleObj = contract.authenticate(inputs.role.role, inputs.role.name,
19                                 inputs.role.organization, inputs.role.department,contract)
20
21                    if(roleObj === null ){
```

e/events.js

[5] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/MeatSale/serializer.js

[6] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/MeatSale/package.json

[7] https://mochajs.org/

[8] https://www.chaijs.com/

```
22                     throw new Error('Unauthorized: Unknown access');
23                 }
24
25             }catch(err){
26                 return { successful: false, message: err.message }
27             }/// end of first layer
28             //second layer
29             let controllers = contract.delivered._controller
30         if(!contract.accessPolicy.hasPermesstion('grant','read', contract.delivered,roleObj, contract.delivered.
           getController(controllers.length - 1)) ||
31               !contract.accessPolicy.isValid(new Rule('grant','read', contract.delivered, roleObj, contract.delivered.
           getController(controllers.length - 1))) ){
32                 throw new Error(`access denied...`)
33               }
34
35          contract.delivered.happen(event)
36          //... omitted code
37          return {successful: true}
38        } else {
39          return {successful: false}
40        }
41      }
```

Listing 10.3: Delivered event transaction with access control (JavaScript)

As shown in Figure 10.1 (top), the test results indicate that the delivered event (and others) can only be triggered by an authorized role, such as the seller in the case of that event. If an unauthorized role attempts to generate delivered , access will be denied (bottom), which aligns with the access policy of the contract.

```
Meat Sale chain code tests
  Test Init transaction.
    ✔ should return error on Init.
  Scenario: Test Triggering Event Transactions
    ✔ The event(delivered) should be generated only by authorized role.
    ✔ The event(paid) should be generated only by authorized role.
    ✔ The event(paidLate) should be generated only by authorized role.
    ✔ The event(inspectedQuality) should be generated only by authorized role.
    ✔ The event(passwordNotification) should be generated only by authorized role.
    ✔ The event(UnLoaded) should be generated only by authorized role.


  7 passing (101ms)
```

```
1) Meat Sale chain code tests
     Scenario: Test Triggering Event Transactions
       The event(delivered) should be generated only by authorized role.:
   Error: access denied...
    at HFContract.trigger_delivered (MeatSale/index.js:66:15)
    at async Context.<anonymous> (test/MeatSale.test.js:172:19)
```

Figure 10.1: Test results for successful event triggering and an unauthorized attempt.

Note that for the unit testing environment, we modified the header and the content (see line (18) in Listing 10.3) of all generated transaction functions to accept the caller's role object as an explicit argument, rather than extracting the role from the X.509 certificate using the `ClientIdentity` class as described in Section 9.3 using Fabric API. This adjustment was necessary because Mocha-based unit tests execute the chaincode using a mocked Fabric context, where certificate-based identity attributes are hard to simulate.

Additionally, to evaluate the effectiveness of our access control mechanism, we conducted a series of tests assessing role-based permissions on event access. The objective was to verify whether permissions are enforced correctly when an event happens or does not happen under varying role-based permissions. Corresponding test cases were developed to

analyze the access control policy when granting or restricting read permissions to specific events.

Table 10.2 shows the test cases conducted, and the result for `TC1` is shown in Figure 10.2. The result of `TC2` is shown in Figure 10.3; we checked both when the delivered event happened and when it did not happen in an authorized attempt. The result of `TC3` is shown in Figure 10.4; the `getEventDateAndTime()` transaction conducts two checks: first with the utility function `hasPermission()` and then with `hasPermissionOnLegalPosition()`, in order to verify whether a role has permission to access the event itself and additionally whether it is part of another legal position.

Table 10.2: Event access control test cases and expected outcomes

Test Case	Scenario	Expected Outcome
`TC1`: Role lacks permission	The shipper attempts to read the state of delivered without explicit permission.	Access should be denied.
`TC2`: Role has event-specific permission.	The seller has permission to read delivered.	Access should be granted.
`TC3`: Role has permission on obligation that includes the event.	The seller has permission on an obligation that includes delivered.	Access should be granted to the seller to determine whether the event delivered has happened or not, but the event's value should remain inaccessible.

```
1) Meat Sale chain code tests
     Scenario: access the sate and time after the event was triggered.
       Should successfully allow access to the state and time of the "delivered" event for authorized roles only.:
   Error: access denied...
    at HFContract.trigger_delivered (MeatSale/index.js:66:15)
    at async Context.<anonymous> (test/MeatSale.test.js:303:20)
```

Figure 10.2: (`TC1`) Test results for successfully denying access to the state and time of the delivered event in an unauthorized attempt.

```
  Meat Sale chain code tests
    Scenario: access the sate and time after the event was triggered.
res getDeliveredDateAndTime { time: '2025-03-15T17:15:00.000Z', state: 'Happened' }
      ✔ Should successfully allow access to the state and time of the "delivered" event for authorized roles only. (44ms)
      ✔ Should successfully allow access to the state and time of the "paid" event for authorized roles only.
      ✔ Should successfully allow access to the state and time of the "paidLate" event for authorized roles only.
      ✔ Should successfully allow access to the state and time of the "disclosed" event for authorized roles only.
    Scenario: the sate and time of delivered when the event does not happen
res.......  { time: null, state: 'Not Happened' }
      ✔ should sucessfully allowed only access to sate and time of authrized role only. The state and the time will be false and null
```

Figure 10.3: (`TC2`) Test results for successfully retrieving the state and time of the delivered event when it happens and does not happen, as well as other events, in an authorized attempt.

Furthermore, we extended our evaluation to include the legal position (obligation/power) states and their transitions, ensuring that SymboleoAC correctly enforces access control while maintaining the integrity of the contract and legal position lifecycles. The tests focused on verifying whether an authorized role could retrieve obligation/power states such

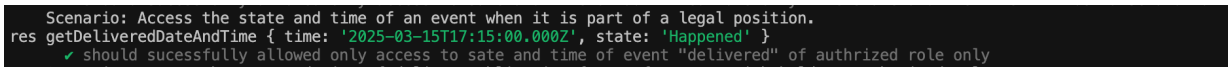


Figure 10.4: (TC3) Test results for successfully retrieving the state and time of the delivered event when it is part of another legal position, in an authorized attempt.

as active , violation , fulfillment , and unsuccessful termination, while unauthorized roles were denied access.

Table 10.3 shows the test cases conducted on the Odelivery obligation. The result of TC4 is shown in Figure 10.5. In addition, the result of TC5 is shown in Figure 10.6.

Table 10.3: Legal position access control test cases and expected outcomes.

Test Case	Scenario	Expected Outcome
TC4: Role has permission to access obligation state.	The seller, who is the performer of the delivery obligation, has the right to access its state.	Access should be granted, and the obligation's state should be returned as active along with its activation timestamp.
TC5: Role lacks permission to access obligation state.	The shipper attempts to read delivery obligation state without permission.	Access should be denied, and an error message should be returned.

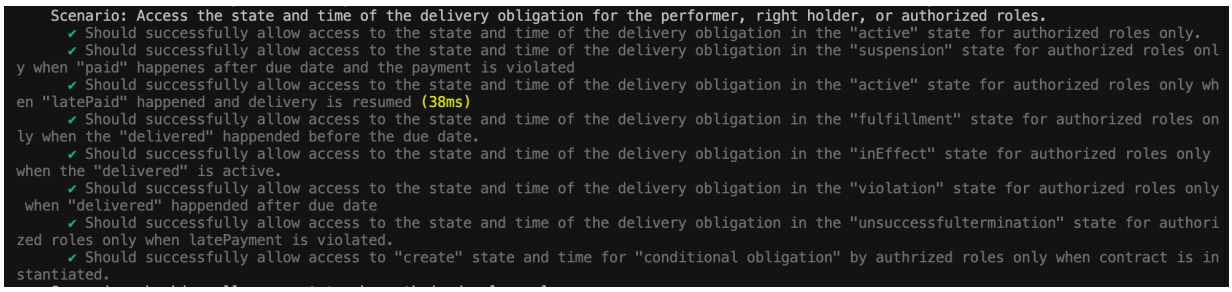


Figure 10.5: (TC4) Test results for successfully retrieving the state and time of Odelivery in an authorized attempt.

The remaining tests covered the following obligation states transition: create, violation , active, fulfillment , discharge, unsuccessful termination, and successful termination, as well as the super states inEffect and suspension. The same set of tests was conducted for power states to ensure consistency in access control enforcement and state transition.

We evaluated the `getLegalPositionStateAndTime()` transaction with different scenarios: (1) Conditional obligation, for example, latePayment; (2) Unconditional obligation, for example, Odelivery; and (3) All state transitions of obligations/powers as mentioned earlier.

Furthermore, we conducted multiple tests to evaluate access control for the informational parts of legal positions when they appear as antecedents or consequents. Specifically, we examined three scenarios: (1) when an obligation or power is part of the antecedent or consequent of another obligation or power, (2) when a condition is part of the antecedent or consequent, and (3) when an event is part of the antecedent or consequent. Table 10.4

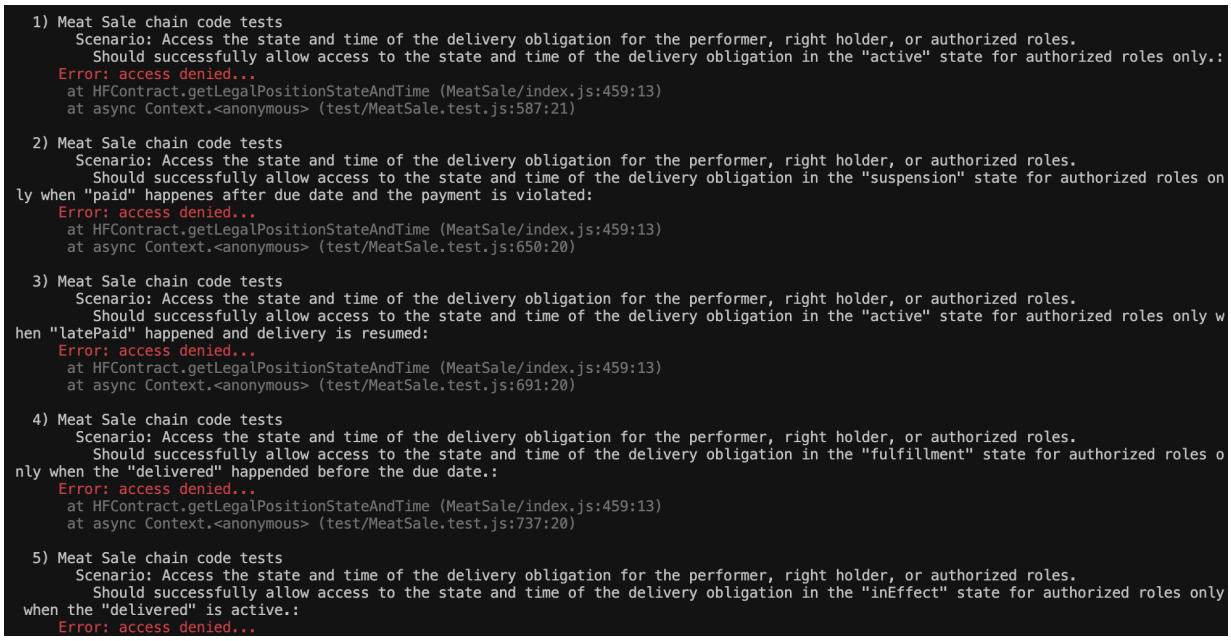


Figure 10.6: (TC5) Test results for successfully denying access to the state and time of Odelivery in unauthorized attempts.

presents different scenarios of access control for the informational parts of legal positions, as explained in Section 7.6. The result of the conducted test is shown in Figure 10.7.

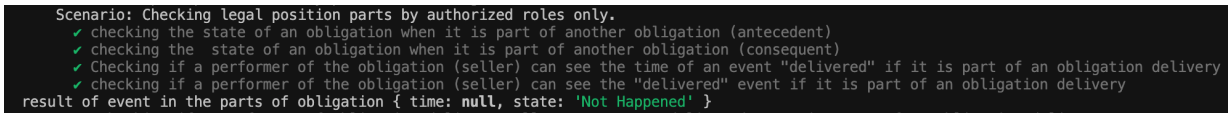


Figure 10.7: (TC6) result for successfully allowing the checking of an obligation's state when it is part of another obligation (antecedent and consequent). (TC7) results for successfully allowing the authorized role to check the time while keeping the value inaccessible. (TC8) result for successfully allowing the authorized role to check the state and time of an event when it is part of another obligation

The complete set of 57 test cases is available online[9]. Experiments confirm that only authorized roles can trigger or inspect events and legal positions, that constraints override conflicting rules, and that policy-driven queries behave as intended.

10.3 COVID-19 Vaccine Procurement Case Study

The contract in Table 10.5 was extracted and adapted from a real vaccine procurement agreement between the US Government and the vaccine manufacturer Pfizer[10], in which

[9] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/test/MeatSale.test.js

[10] https://www.hhs.gov/sites/default/files/pfizer-inc-covid-19-vaccine-contract.pdf

Table 10.4: Legal position informational parts access control test cases and expected outcomes

Test Case	Scenario	Expected Outcome
TC6: Role is the performer of an obligation and has permission to access obligation state and its informational part.	Another role, storage, is added as the rightholder of the delivery obligation. In this way, the buyer, who is the performer of the payment obligation, cannot see or read the state of the delivery obligation. However, because Happens(Fulfilled(obligations.delivery)) is the antecedent of the payment obligation and the buyer is its performer, they can read the state of the delivery obligation as part of fulfilling the payment obligation.	Access should be granted to the buyer to read the state of delivery obligation that is part of the payment obligation.
TC7: Role has access to condition, e.g., temperature data.	The shipper is the performer of the delivered event, while the seller is the performer of the delivery obligation and is responsible for fulfilling it. The seller can only access the temperature data.	The seller can see whether the temperature is above 2°C, but the exact value remains inaccessible.
TC8: Role is the performer of an obligation and has access to event that is part of that obligation.	The seller can see the delivered has happened.	Access should be granted to that role to check only of event has happened or not.

the manufacturer promises to produce a specified number of doses of the COVID-19 vaccine, maintain them in appropriate condition, and deliver them. This contract was selected because it contains a variety of events enabling contract monitoring, there are many obligations and powers, delivery can happen in multiple steps (requiring variable updates), and there are surviving obligations that continue beyond the normal termination of the contract.

Table 10.5: COVID-19 vaccine procurement contract

This contract is entered into effective as of <*effDate*>, between Medical CBRN (Chemical, Biological, Radiological and Nuclear) Defense Consortium Transport (MCDC) as <*Government*> and Pfizer as <*Manufacturer*>.

1. **Manufacturing & Delivery**

   1.1 The Government may request that Pfizer produces and delivers vaccine doses. Any order will provide for a minimum of <*minQuantity*> doses while an aggregate number of doses ordered shall not exceed <*maxQuantity*>.

   1.2 Upon any request, Manufacturer shall inform the Government of appropriate lead times, and Manufacturer and the Government shall mutually agree on an appropriate estimated delivery schedule.

   1.3 Pfizer anticipates providing the vaccine, subject to U.S. Food and Drug Administration (FDA) approval or authorization, as <*temperature*>°C frozen product that needs to be maintained at or below that temperature prior to dosing. The Government acknowledges that Manufacturer's responsibility for cold chain will cease upon delivery.

   1.4 Manufacturer will notify the Government of the date by which doses will become available for delivery. The Government will confirm dosage orders by ship-to location <*deliveryAdd*> in advance of those dates.

   1.5 Even if a vaccine is successfully developed and obtains FDA regulatory approval or authorization, Pfizer shall have no liability for any failure to deliver doses in accordance with the estimated delivery dates to the extent any such change in delivery dates is based on emerging data, regulatory guidance, manufacturing and technical developments, or other risks outside Pfizer's control.

2. **Payment**

   2.1 Due to variances in fill/finish yield, Manufacturer shall invoice for and the Government shall pay for actual quantities delivered, at a rate of $ <*unitPrice*> per dose.

   2.2 Upon release, Manufacturer will ship the doses to the Government. Manufacturer expects to invoice the Government every month for released doses that have been shipped during each such monthly period. The Government will pay all such invoices within thirty (30) days of receipt thereof.

   2.3 The Government will have no right to withhold payment in respect of any delivered doses unless the FDA has withdrawn approval or authorization of the vaccine.

3. **Termination**

   3.1 Except as required by applicable law or regulation or judicial or administrative order, the Government shall not have the authority to issue a Stop-Work Order to halt the work contemplated under this Statement of Work.

   3.2 In the event of termination of this Agreement, or expiration of this Agreement at the end of the period of performance, any Party hereto shall not be released of any liability, including any outstanding payments of the Government for doses previously delivered hereunder, which at the time of termination or expiration had already accrued to the other party in respect to any act or omission prior thereto.

This contract has two parties, Pfizer and the Medical CBRN (Chemical, Biological, Radiological and Nuclear) Defense Consortium Transport (MCDC), respectively playing the roles of Manufacturer and Government. MCDC is obligated to order a specified number of vaccine doses and pay the amount due within 30 days of the invoice date, while Pfizer is obligated to deliver vaccine doses at the required temperature before the specified deadline

and after obtaining approval from the U.S. Food and Drug Administration (FDA).

MCDC can order up to a specified maximum number of doses, and this may happen through multiple requests. In each request, the number of doses must be greater than or equal to the specified minimum number of doses. The request, delivery, and payment must go through specific required steps, including notification, confirmation, and agreement on the delivery lead time and final delivery time and location. After the requesting and delivery steps are completed, Pfizer can issue an invoice and accordingly, MCDC must pay an amount equal to the total price of the doses delivered.

Also, payment cannot be made if the FDA withdraws the approval of the vaccine. This can happen even after the termination of the contract. The occurrence of actions such as request, delivery, and payment are indicated by the requested, delivered, and paid events.

According to the contract text (Clause 3), the MCDC can terminate the contract on its part only if the applicable law or regulation or a judicial or administrative order terminates the contract. Otherwise, this agreement is terminated by both parties at the end of the performance as no more doses can be ordered and the ordered doses have been delivered to the designated locations on the specified condition and dates.

In addition, just like we upgraded the meat sale contract, we extended the original specification to capture a more realistic cyber-physical smart contract setting by introducing multiple trusted **thirdParty** roles, namely Regulator, Admin, WorldCourier (responsible for vaccine transportation), and FDA. This extension allows the contract to explicitly model regulatory oversight, administrative control for blockchain administrative execution, external delivery execution, and approval monitoring, which are essential for representing secure and compliant vaccine procurement workflows in practice.

In addition, we refined the modeling of the specification by treating each role as an explicit *object* that is declared in the `Declarations` section, and then systematically reused in the definition of contractual events, obligations, and access control rules. This improves consistency across the contract model and aligns the generated smart contract code with a clear identity centric execution semantics.

In addition, the contract also features an access-control policy with rules that govern how various roles can access and modify specific aspects of the contract. For example, we revoke Pfizer's write access to specify the value attribute of all sensors; instead, the regulator is the only entity permitted to write to this attribute, since the regulator is the performer. The regulator acts as the controller of the access control policy, overseeing these rules and having the authority to override access control permissions set by other resource controllers.

To support IoT execution, we also restructured how monitoring constraints are represented. In the earlier vaccine procurement contract version, temperature monitoring was directly embedded within the contract domain and obligation conditions (e.g., as a fixed attribute in the delivery event). Instead, we removed direct temperature references from the obligation structure and replaced them with a general **DataTransfer** abstraction (Alert), which enables external sensing and monitoring to be expressed uniformly as off-chain data streams. This design supports multi-sensor monitoring in the contract execution by capturing cold chain compliance using multiple alert types, including temperature, humidity,

shock, lightExposure, and sealOpen. Each alert can be linked to a specific request instance using a reqID identifier, enabling the contract to reason about violations that occur during individual vaccine shipment executions. The monitored threshold values for vaccine cold-chain logistics were selected based on established Canadian handling and transportation guidance for temperature controlled medical products[11]. These sensors continuously track critical data throughout the transportation process as follows:

- temperature sensors monitor cold-chain compliance and raise an alert when the measured value exceeds $-80°$ (i.e., `value > -80`) within a sliding window of 10 readings, if this violation occurs at least 5 times (`count := 5`).
- humidity sensors monitor environmental humidity and trigger an alert when the humidity exceeds 70% within a window of 15 readings, repeated at least 3 times.
- shock sensors detect excessive vibration or impact and raise an alert when the measured value exceeds 5. Since a single occurrence constitutes a violation, no aggregation window or repetition threshold (i.e., count) is required.
- lightExposure sensors detect light exposure during transport and immediately trigger an alert when any non-permitted exposure is detected. This represents a direct violation and does not require window-based or count-based evaluation.
- sealOpen sensors detect unauthorized container opening and raise an alert immediately upon detection. As this event represents a direct breach, no window or count condition is applied.

In all cases, the alerts are controlled by pfizer (as the contract thirdParty responsible for the shipment conditions), while the regulator is the designated performer of these DataTransfers, enabling independent monitoring of transportation risks.

Finally, to ensure alignment between the generated smart contract interface and the off-chain runtime components, we standardized role parameter naming by using the `P` suffix (e.g., pfizerP, mcdcP, regulatorP). This naming convention is consistently reflected both in the contract parameters and in the testing harness, improving traceability between the SymboleoAC specification, the generated smart contract, and the SymboleoAC Node.js execution scripts used in the evaluation.

Listing 10.4 shows the Vaccine Procurement contract specification (Table 10.5) expressed in our new SymboleoAC language.

```
Domain covidVaccineProcurementD
  Manufacturer isA Role with name: String, org: String, dept: String;
  Government isA Role with name: String, org: String, dept: String;
  Regulator isA Role thirdParty with name: String, org: String, dept: String;
  Admin isA Role thirdParty with name: String, org: String, dept: String;
  WorldCourier isA Role thirdParty with name: String, org: String, dept: String;
  FDA isA Role thirdParty with name: String, org: String, dept: String;
  Invoiced isA Event with Env reqID : String, Env noOfDoses : Number, Env date : Date,  performer: Government,
     controller: Government;
  Paid isA Event with Env reqID : String, Env amount : Number, performer: Government, controller: Government;
```

[11] https://shorturl.at/EYyA3

```
10    Requested isA Event with Env reqID : String, Env dosage : Number, Env date : Date, performer: Government, controller
          : Government;
11    LeadtimeInformedNegotiated isA Event with Env reqID : String, Env date : Date, performer: Manufacturer, controller:
          Manufacturer;
12    NotifiedOfDelivery isA Event with Env reqID : String, Env delD : Date, performer: Manufacturer, controller:
          Manufacturer;
13    Location isAn Enumeration(Ottawa, Toronto, Montrial, Vancover);
14    Confirmed isA Event with Env reqID : String, Env shipToLocation: Location, performer: Government, controller:
          Government;
15    Delivered isA Event with Env reqID : String, Env dosage : Number, Env delAddr : Location, Env date: Date, performer:
          WorldCourier, controller: Manufacturer;
16    VaccineDose isA Asset with  price : Number, FDAapproval : Boolean, owner: Manufacturer;
17    StopWork isA Event with performer: Government, controller: Government;
18    //thirdParty stopWork
19    ThirdPartyStopWork isA Event with performer: Regulator, controller: Regulator;
20    Agreed isA Event with Env reqID: String, performer: Government, controller: Government;
21    Risk isA Event with Env reqID: String, Env extendedDel: Date, performer: Manufacturer, controller: Manufacturer;
22    Remain isA Asset with value:Number, owner: Government;
23    PaidAmount isA Asset with value:Number, owner: Government;
24    WithdrewApproval isA Event with performer: FDA, controller: FDA;
25    TerminateAgreementG isA Event with performer: Government, controller: Government;
26    TerminateAgreementM isA Event with performer: Manufacturer, controller: Manufacturer;
27    Alert isA DataTransfer with Env value: Number, Env sensorTimestamp:String, condition: String, window: String, count:
          String, Env reqID : String, controller:Manufacturer, performer:Regulator;
28
29  endDomain
30
31  TimeGranularity is days
32  Contract VaccineProcurementC (pfizerP :Manufacturer, mcdcP:Government, regulatorP: Regulator, adminP: Admin, fdaP: FDA
        , worldcourierP: WorldCourier,
33  approval : Boolean, unitPrice : Number, minQuantity : Number, maxQuantity:Number)
34
35  Declarations
36      regulator: Regulator with name:= regulatorP.name, org:= regulatorP.org, dept:= regulatorP.dept;
37      admin: Admin with name:= adminP.name, org:= adminP.org, dept:= adminP.dept;
38      pfizer: Manufacturer with name:= pfizerP.name, org:= pfizerP.org, dept:= pfizerP.dept;
39      mcdc: Government with name:= mcdcP.name, org:= mcdcP.org, dept:= mcdcP.dept;
40      fda: FDA with name:= fdaP.name, org:= fdaP.org, dept:= fdaP.dept;
41      worldcourier: WorldCourier with name:= worldcourierP.name, org:= worldcourierP.org, dept:= worldcourierP.dept;
42
43
44      requested : Requested  with performer:= mcdc, controller:= mcdc;
45      leadtimeINform : LeadtimeInformedNegotiated  with performer:= pfizer, controller:= pfizer;
46      notifiedOD : NotifiedOfDelivery  with performer:= pfizer, controller:= pfizer;
47      delivered : Delivered  with performer:= worldcourier, controller:= pfizer;
48      invoiced : Invoiced with performer:= mcdc, controller:= mcdc;
49      paid : Paid with performer:= mcdc, controller:= mcdc;
50      confirmed: Confirmed with performer:= mcdc, controller:= mcdc;
51      lawStopWork: StopWork with performer:= mcdc, controller:= mcdc;
52      regulationStopWork:ThirdPartyStopWork with performer:= regulator, controller:= regulator;
53      judicialStopWork:ThirdPartyStopWork with performer:= regulator, controller:= regulator;
54      adminStopWork: ThirdPartyStopWork with performer:= regulator, controller:= regulator;
55      govStopWork: StopWork with performer:= mcdc, controller:= mcdc;
56      vaccineDose : VaccineDose with  price := unitPrice, FDAapproval := approval, owner:=pfizer;
57      agreedFromG: Agreed with performer:= mcdc, controller:= mcdc;
58      outsideRisk: Risk with performer:= pfizer, controller:= pfizer;
59      remain: Remain with value:= maxQuantity, owner:=mcdc;
60      paidAmount:PaidAmount with value:=0, owner:=mcdc;
61      withdrewApproval: WithdrewApproval  with performer:= fda, controller:= fda;
62      mcdcTerminateAgreement: TerminateAgreementG with performer:= mcdc, controller:= mcdc;
63      pfizerTerminateAgreement: TerminateAgreementM with performer:= pfizer, controller:= pfizer;
64      //sensors
65      temperature: Alert with condition := "value > -80", window := "10", count := "5", controller := pfizer, performer
          := regulator;
66      humidity : Alert with condition := "value > 70", window := "15", count := "3", controller := pfizer, performer :=
          regulator;
67      shock : Alert with condition := "value > 5",  window := "", count := "", controller := pfizer, performer :=
          regulator;
68      lightExposure : Alert with condition := "value > 0",  window := "", count := "", controller := pfizer, performer
          := regulator;
69      sealOpen : Alert with condition := "value > 0", window := "",  count := "", controller := pfizer, performer :=
          regulator;
70
71
72
73  Preconditions
74  vaccineDose.FDAapproval==true;
75
76
77  Obligations
78      // to keep the contract in active state until the remain doses is less than the minimum requested quantity (end of
          performance) and the mcdc and pfizer terminate the agreement
79    oRequestVaccineDosage: O(mcdc, pfizer, true, (remain.value< minQuantity)
80                          and Happens(Fulfilled(obligations.oAgreedOnRequest))
81                 and Happens(Fulfilled(obligations.oDeliver)) and Happens(Fulfilled(obligations.oAssign))
82                 and  Happens(mcdcTerminateAgreement) and Happens(pfizerTerminateAgreement)
83    );
84    // A Request must satisfy all the conditions required in a particular order
85    oAgreedOnRequest: Happens(requested)-> O(mcdc,pfizer,Happens(agreedFromG),
86                        ShappensBefore(leadtimeINform,agreedFromG)
87              and leadtimeINform.reqID==requested.reqID and agreedFromG.reqID==requested.reqID
88              and (requested.dosage >= minQuantity and requested.dosage<=remain.value)
```

```
            );
  // A delivery obligation is achieved if the delivering operation satisfies all the conditions required in a
     particular order
  oDeliver: Happens(requested)->O(pfizer,mcdc, Happens(Fulfilled(obligations.oAgreedOnRequest))
             and Happens(confirmed),
            ShappensBefore(notifiedOD,confirmed)
            and Happens(delivered)
         and  delivered.delAddr==confirmed.shipToLocation
        and vaccineDose.FDAapproval==true
        and delivered.reqID==requested.reqID
        and notifiedOD.reqID==requested.reqID and confirmed.reqID==requested.reqID
        and ((delivered.date==notifiedOD.delD) or
             (Happens(outsideRisk) and delivered.date==outsideRisk.extendedDel and requested.reqID==outsideRisk.reqID)
     )
              and  (not Happens(temperature) or temperature.reqID !=requested.reqID)
              and  (not Happens(sealOpen) or sealOpen.reqID != requested.reqID)
              and  (not Happens(humidity) or humidity.reqID!=requested.reqID)
              and  (not Happens(shock) or shock.reqID != requested.reqID)
             and  (not Happens(lightExposure) or lightExposure.reqID != requested.reqID) ) with Controller regulator;

  // Calculate the remains of the doses and the price of the doses delivered when the required doses are delivered,
     fulfilling all the agreed-upon conditions
  oAssign: Happens(requested)->O(mcdc,pfizer,Happens(delivered) and delivered.reqID==requested.reqID,
           HappensAssign(Fulfilled(obligations.oDeliver), remain.value:=remain.value-delivered.dosage; paidAmount.
     value:=delivered.dosage * vaccineDose.price )
           and delivered.reqID==requested.reqID
       );


Surviving Obligations

//Checking the agreed terms necessary to activate and complete the payment process
  oPay: Happens(requested)-> Obligation(mcdc, pfizer,Happens(invoiced) and vaccineDose.FDAapproval==true and invoiced.
     reqID==requested.reqID and Happens(Fulfilled(obligations.oDeliver))
      and ShappensBefore(delivered,invoiced)and delivered.reqID==requested.reqID,
      ShappensBefore(paid, Date.add(invoiced.date , 30, days))
      and invoiced.reqID==requested.reqID
      and invoiced.reqID==paid.reqID
      and paid.amount == paidAmount.value
      ) ;
// FDA approval monitoring where mcdc must pay for vaccine doses already approved by FDA
  oWithdrewApproval: O(pfizer, mcdc,Happens(withdrewApproval),
           Assign(vaccineDose.FDAapproval:=false)
       );
Powers
// MCDC has the power to stop the work if one of the following four events occurs
  pStopWork: Happens(lawStopWork) or Happens(adminStopWork) or Happens(regulationStopWork) or Happens(judicialStopWork
     ) ->
          P(mcdc, pfizer, Happens(govStopWork), Terminated(self));
// Terminate the contract at the end of the performance
  pTermination: Happens(Fulfilled(obligations.oRequestVaccineDosage)) -> P(pfizer,mcdc, true, Terminated(self));



ACPolicy with Controller regulator
Rule1: Grant read To mcdc On vaccineDose by pfizer; // give read permission to government to read information
     vaccineDose
Rule2: Grant read To worldcourier On vaccineDose.FDAapproval by pfizer;
Rule3: Grant read To mcdc On delivered by pfizer;
Rule4: Revoke write To pfizer On temperature.value by regulator; // we took the right of controller to write (
     constraint)
Rule5: Revoke write To pfizer On humidity.value by regulator;
Rule6: Revoke write To pfizer On shock.value by regulator;
Rule7: Revoke write To pfizer On lightExposure.value by regulator;
Rule8: Revoke write To pfizer On sealOpen.value by regulator;
Rule9: Grant read To pfizer On withdrewApproval by fda; // permission by FDA to MCDC to see the state

endContract
```

Listing 10.4: Vaccine Procurement contract specification in SymboleoAC, adapted from [89]

10.3.1 Results – SymboleoAC2SC Compiler

The contract class is generated successfully. For the Vaccine Procurement case study, we iteratively updated the code generation logic multiple times, in particular to support Surviving Obligations, which were not present in the Meat Sale contract of the previous section. Specifically, we extended the generation of unconditional surviving obligations to handle three informational parts that may appear in the antecedents or consequents,

as discussed in Section 9.5.7. In addition, we assign the controller and instantiate the corresponding access control rules as soon as the surviving obligation is created. The complete generated vaccine procurement contract is shown in Appendix D, and is also available online[12].

The `index.js` file, which contains the main entry point for Hyperledger Fabric and implements all transactions described in Section 9.5, is also generated successfully. For the Vaccine Procurement case study, we had to update the SymboleoACJS library functions to ensure compatibility with the generated code. For example, we refined the happen method, since all event attributes are now instantiated using the `Attribute` class in the SymboleoACJS core library. We also updated the code generation (SymboleoAC2SC); e.g. we updated the `getIoTCondition()` transaction (see Section 9.5.4) to support the additional sensor types introduced in the Vaccine Procurement case study, such as sealOpen and lightExposure. These sensors have different alert behaviors and therefore require different rule configurations. Specifically, we revised the generated `select` statement used by the CEP engine to produce EPL rules that support heterogeneous attributes. Although the attributes `window` and `count` were already part of the rule structure, we modified them in the generated `rules.json` to allow them to be empty when they are not required. This is necessary because some sensors trigger alerts immediately and do not rely on windows or counts. These updates ensure the support of different types of sensors and compatibility with the CEP engine while preserving the intended alert semantics defined in the SymboleoAC specification. The full generated smart contract index file is available online[13].

The event subscriptions and listeners (`events.js`[14]) were also generated successfully after multiple iterations of refining the code generation logic. In particular, we tested different variants of conditional obligations and powers, where the corresponding rules are instantiated only when the triggering condition occurs, as discussed in Section 9.4.6. We also evaluated several variants of antecedents and consequents involving **DataTransfer** as well as informational parts that appear in the antecedents and consequents of conditional legal positions, as discussed in Section 9.5.7.

During evaluation with the Vaccine Procurement case study, we refined and regenerated the deserialization method discussed in Section 9.4.10. In particular, we updated the iterative reconstruction logic (i.e., several nested `for` loops) to improve restoring contract instances from their serialized representation. These refinements ensure that the generated runtime correctly handles a wider range of nested structures, including primitive fields (e.g., strings) and embedded objects that may not always follow the same naming or typing conventions. As a result, the deserializer can reliably reconstruct the full contract state for larger specifications and richer data models without leaving attributes undefined. The full generated file is available online[15].

[12] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/tree/main/VaccineProcurementC

[13] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/VaccineProcurementC/index.js

[14] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/VaccineProcurementC/events.js

[15] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/Vaccine

Finally, we generated `package.json`, in which the SymboleoACJS library is declared as a dependency with an updated version following modifications to the core library.

10.3.2 Results – Validation Scenarios

We carried out unit testing to validate the behavior of the generated smart contract across possible relevant scenarios, under both normal and exceptional executions, to ensure that it adheres to the semantics of SymboleoAC. The ten scenarios used on the Vaccine Procurement contract (totalizing 44 test cases, available online[16]), with a coverage of the remaining SymboleoAC concepts (and combinations thereof) not covered in the first case study, are presented here.

- **Scenario 1 – Authorized initialization and restricted contract activation.** The contract is initialized using the `init` transaction, where only an authorized identity can deploy and instantiate the contract state. Upon success, the contract transitions to Active/InEffect. This scenario confirms that contract deployment and activation is protected and cannot be triggered by unauthorized parties.
- **Scenario 2 – Authorized execution path with correct ordering of events (access & state transition).** All required events are triggered in the correct order by their *authorized performers* (e.g., Government triggers requested, Manufacturer triggers leadtimeINform, etc.). The obligations oAgreedOnRequest, oDeliver, and oAssign transition to Fulfillment, while the contract remains Active. This scenario validates that (i) events can only be triggered by authorized roles (ii) the contract state transitions only when authorized roles invoke transactions, and (iii) the required sequencing constraints are enforced.
- **Scenario 3 – Access controlled event state/time query (event visibility).** After delivered is triggered, the performer of Delivered, i.e., pfizer (and any authorized reader via SymboleoAC rules, i.e., mcdc) can successfully invoke transaction `getEventDateAndTime()` to read the event's state (Happened) and timestamp. Unauthorized identities are denied access. This scenario validates event-level access control.
- **Scenario 4 – Two valid requests, then contract termination by authorized role.** Two requests are executed with authorized event triggers. After completing delivery, the contract transitions to SuccessfulTermination when authorized termination events are invoked (e.g., mcdcTerminateAgreement and pfizerTerminateAgreement). The surviving obligation oPay remains in Create until payment is provided by an authorized role. This confirms that only authorized parties can terminate the contract, with surviving obligations remaining properly active.

`ProcurementC/serializer.js`

[16] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/blob/main/test/Vaccine.test.js

- **Scenario 5 – Access to conditional expressions (legal position information parts) based on performer rights.** Authorized roles retrieve the evaluation result of critical informational conditions (e.g., vaccineDose.FDAapproval == true) through `getStateTimeOfParts()`. This confirms that access control applies not only to events and assets, but also to derived informational parts discussed in Section 9.5.7 and used inside obligations and powers.

- **Scenario 6 – Access to legal position state (legal position information parts) based on performer rights.** Authorized roles retrieve the state of a legal position (e.g., Happens(Fulfilled(obligations.oAgreedOnRequest)) that is part of another obligation (e.g., `oDeliver`) through `getStateTimeOfParts()`. For example, Manufacturer (i.e., pfizer) can see the fulfillment of oAgreedOnRequest because it is part of oDeliver and pfizer is the performer of oDeliver. This confirms that access control applies not only to SymboleoAC resources, but also to derived informational parts discussed in Section 9.5.7 and used inside obligations and powers.

- **Scenario 7 – Access to event state and time (legal position information parts) based on performer rights.** Authorized roles retrieve the state and time of an event (e.g., govStopWork event that is part of pStopWork) through the transaction `getStateTimeOfParts()`. The Manufacturer can check whether and at what time the Government triggered the govStopWork event as it is part of the power. This further confirms that access control applies not only to resources, but also to derived informational parts discussed in Section 9.5.7 and used inside obligations and powers.

- **Scenario 8 – Access to state of legal position.** Authorized roles retrieve the state of a legal position (e.g., oAgreedOnRequest) via `getLegalPositionStateAndTime()`. The Government can access the state and time of oAgreedOnRequest as the performer of this obligation.

- **Scenario 9 – Delivery violated due to late delivery date (controlled transitions).** If delivered.date violates the agreed schedule (relative to notifiedOD.delD), the delivery obligation transitions to Violation and the contract transitions to UnsuccessfulTermination. Only authorized roles can submit the delivery transaction, and only authorized roles can inspect the resulting violation evidence.

- **Scenario 10 – Notification delivery to authorized roles.** This scenario validates the notification mechanism generated from the SymboleoAC specification. After successful execution of the relevant transactions, the smart contract emits a notification event containing information such as the transaction status, the authorized access roles, a timestamp, and related metadata. In the unit tests, we demonstrate that these notification events are generated correctly. However, authorized roles access these notifications through the message broker via the SymboleoAC API, as discussed in Section 10.4.4. Note that, in case of contract SuccessfulTermination or UnsuccessfulTermination, notifications are generated not only for the final contract state, but also for all obligations and powers whose states transition as part of reaching successful or unsuccessful termination.

To test the contract under the scenarios described above, we developed a comprehensive set of unit tests. As shown in Figures 10.8 and 10.9, the results confirm that the generated smart contract successfully exhibits the expected behavior.

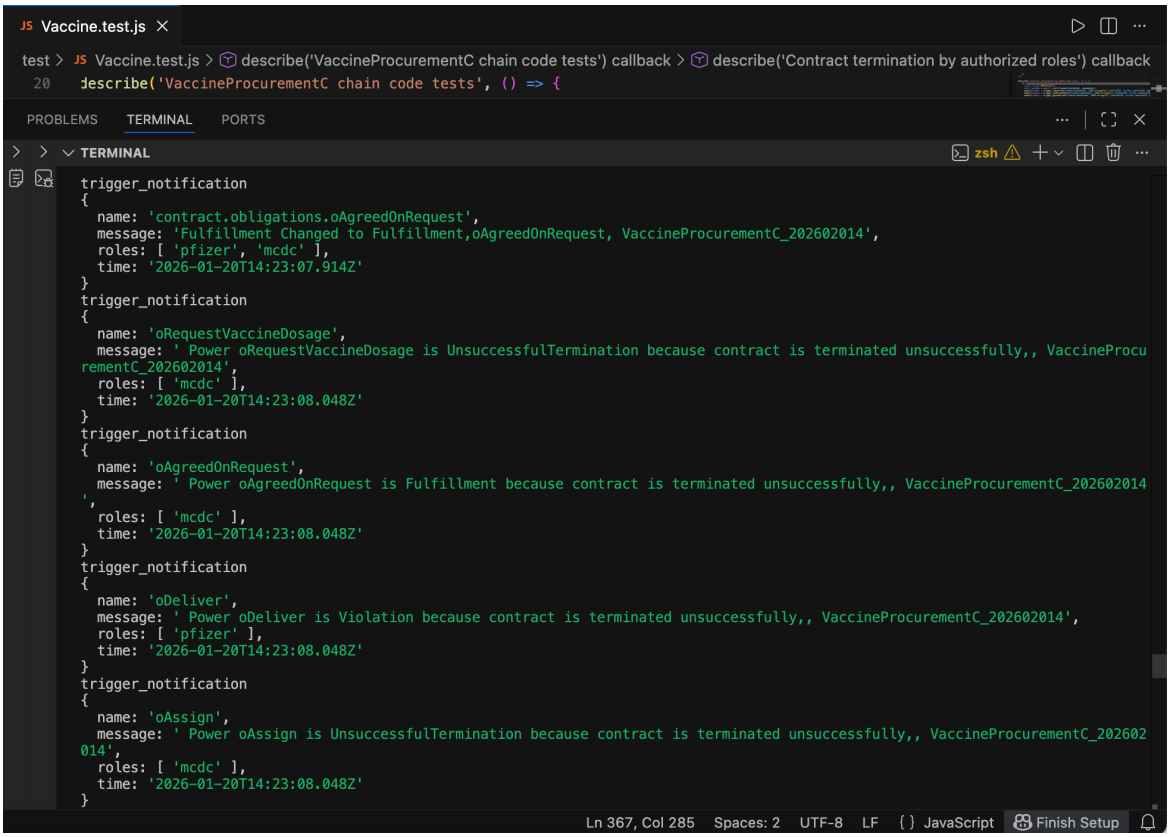


Figure 10.8: Unit test output showing a snippet of the generated notification events for the Vaccine Procurement contract. Each emitted notification includes a status message, the timestamp, and the authorized roles allowed to receive the notification.

10.3.3 Results – Assignment Expression

The new type of predicate function introduced in Section 8.2, HappensAssign, is generated in several forms based on its type and location (antecedent or consequent) in the related object (obligation, surviving obligation, or power). For example, from Listing 10.4, the HappensAssign function is the consequent of the oAssign obligation. Thus, satisfying this function with the related condition (delivered .reqID==requested.reqID) fulfills the obligation. In another example, HappensAssign includes an event (Fulfilled (obligations . oDeliver)) and two equations, (10.1) and (10.2). The first equation subtracts the value of the actual doses delivered from the remaining doses (clause 1.1 in Table 10.5) and the second equation calculates the price of doses delivered enabling a comparison to the amount paid when payment is due (clause 2.1 in Table 10.5).

$$contract.remain.value = contract.remain.value - contract.delivered.dosage \tag{10.1}$$

Figure 10.9: Results of testing the Vaccine Procurement smart contract under the previously discussed scenarios.

$$contract.paidAmount.value = \\ contract.delivered.dosage * contract.vaccineDose.price \tag{10.2}$$

Both equations must be evaluated when the event occurs. Since the HappensAssign is located in the consequent of the oAssign obligation, as shown in Listing 10.4, the equations must be evaluated in each listener function that moves the oAssign obligation to its fulfillment state, including `CreateObligation_oAssign()`, `fulfillObligation_oAssign()`, and `ActivateObligation_oAssign()` functions. Also, a condition to check whether the event Fulfilled (obligations . oDeliver) has happened or not must be evaluated before calculating the two equations and translating the obligation state to fulfillment. Listing 10.5 shows the generated `createObligation_oAssign()` listener function that implements this context.

```
createObligation_oAssign(contract) {
    if (Predicates.happens(contract.requested) ) {
      if (contract.obligations.oAssign == null ||
        contract.obligations.oAssign.isFinished()) {
        const isNewInstance =  contract.obligations.oAssign != null && contract.obligations.oAssign.isFinished()
        contract.obligations.oAssign = new Obligation('oAssign', contract.pfizer, contract.mcdc, contract)
        if (!isNewInstance  ) {
          contract.obligations.oAssign.trigerredUnconditional()
          if (!isNewInstance && Predicates.happens(contract.obligations.oDeliver && contract.obligations.oDeliver._events.Fulfilled)  && contract.delivered.reqID === contract.requested.reqID) {
           contract.remain.value = contract.remain.value - contract.delivered.dosage
           contract.paidAmount.value = contract.delivered.dosage * contract.vaccineDose.price
            contract.obligations.oAssign.fulfilled()
          }
        } else {
          contract.obligations.oAssign.trigerredConditional()
        }
      }
    }
  }
```

Listing 10.5: Improved event listener generated to create the oAssign obligation.

As part of the obligation consequent, the Fulfilled (obligations . oDeliver) event is added to the event listener to call the `fulfillObligation_oAssign()` function every time the event has occurred in order to evaluate the fulfillment conditions. Listing 10.6 shows the generated `fulfillObligation_oAssign()` function where all the conditions and events located in the consequent of the obligation are evaluated, including the above event, before calculating the new values and changing the state to fulfillment.

```
fulfillObligation_oAssign(contract) {
    if (contract.obligations.oAssign != null && (Predicates.happens(contract.obligations.oDeliver && contract.
     obligations.oDeliver._events.Fulfilled)  && contract.delivered.reqID === contract.requested.reqID) ) {
     contract.remain.value = contract.remain.value - contract.delivered.dosage
     contract.paidAmount.value = contract.delivered.dosage * contract.vaccineDose.price
      contract.obligations.oAssign.fulfilled()
    }
  }
```

Listing 10.6: Improved event listener generated to fulfill the oAssign obligation.

10.4 Results – SymboleoAC in a CPSC Environment

In addition to testing the generated smart contracts using unit tests, we also evaluated them within a cyber-physical environment to ensure that they conform to the CPSC architecture described in Chapter 5. Specifically, we deployed and tested the contracts on a Hyperledger Fabric blockchain network using different SymboleoAC specifications.

The smart contracts generated for both the Meat Sale and Vaccine Procurement case studies were successfully deployed on a Hyperledger Fabric test network using Docker[17]. We developed a custom Hyperledger Fabric testing environment, available online[18]. Each contract was packaged and deployed as a separate chaincode instance with its own chaincode name (e.g., `meatsale` for the Meat Sale contract and `vaccineprocurementc` for the Vaccine Procurement contract). Figure 10.10 shows the successful deployment of both chaincodes as reported in the Docker dashboard.

In addition, the SymboleoAC API that targets the Node.js runtime[19], discussed in Chapter 6, was successfully configured to communicate with the deployed blockchain network through the Fabric SDK for Node.js[20], the Fabric Gateway[21], and connection profile[22]. This configuration enables the SymboleoAC API to load identities from the wallet, establish a secure connection to the target channel chaincode (i.e., meat sale or vaccine smart contract), invoke chaincode transactions (e.g., `trigger_delivered()`, `trigger_paid()`, and `getIoTCondition()`), query on-chain contract resources, and continuously listen to

[17] https://github.com/hyperledger/fabric-samples/tree/main/test-network
[18] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-HyperledgerFabric-Test-Netwrok
[19] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-Application-API
[20] https://hyperledger-fabric.readthedocs.io/en/release-2.2/fabric-sdks.html
[21] https://hyperledger.github.io/fabric-gateway/main/api/node/index.html
[22] https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/connectionprofile.html

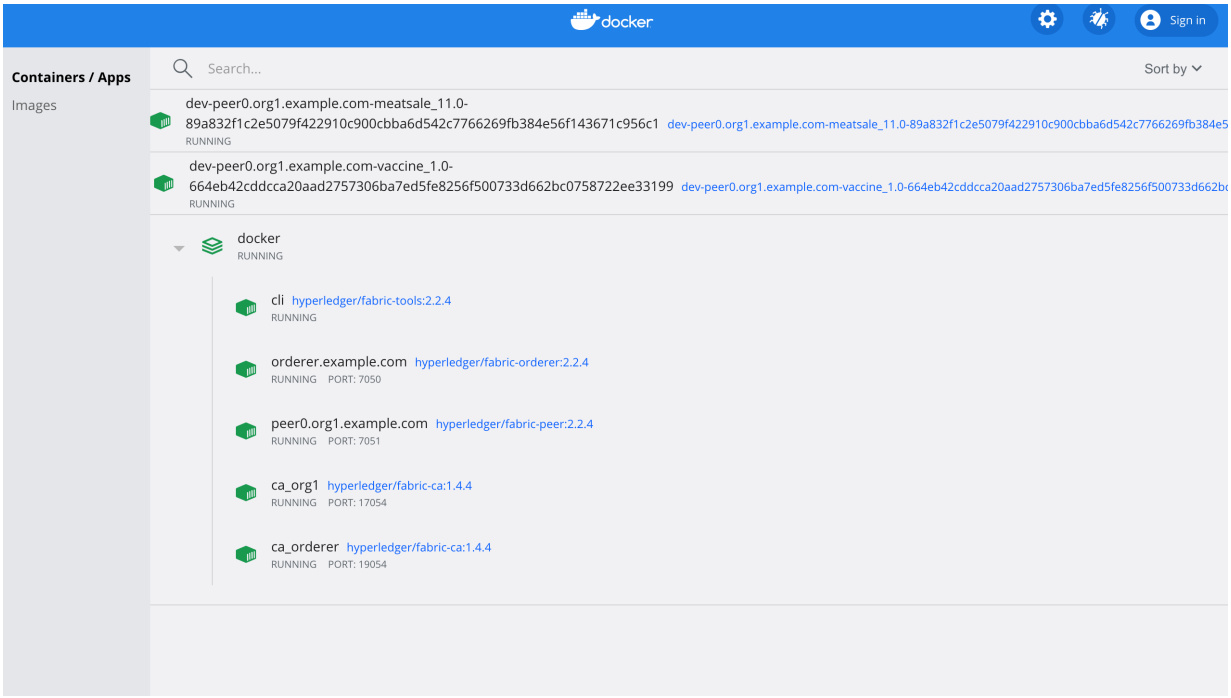


Figure 10.10: Docker containers running multiple deployed chaincodes (Meat Sale and Vaccine Procurement) on the same Hyperledger Fabric network.

emitted blockchain events, ensuring proper authentication and authorization according to the access control rules generated from the corresponding SymboleoAC specification.

Moreover, the SymboleoAC API acts as the main off-chain middleware component that bridges the deployed smart contracts with external services, such as IoT devices, the message broker, and the CEP engine. The correctness of this integration is experimentally validated in this section.

The following subsections present the experimental results in the following order: enrolling users and retrieving IoT rules (Section 10.4.1), enrolling IoT devices (Section 10.4.2), CEP engine and message broker enrollment (Section 10.4.3), notification generation and delivery (Section 10.4.4), and full-cycle execution workflow (Section 10.4.5).

10.4.1 Results – Enrolling Users and Retrieving IoT Rules

To evaluate run-time execution for secure user enrollment and IoT-driven contract execution, we instantiated both the Meat Sale and the Vaccine Procurement contracts and executed an end-to-end workflow involving the deployed smart contracts and the SymboleoAC API. This workflow, which follows with the general steps of Figure 9.2, includes: (1) storing the contract roles on-chain together with their cryptographic hash, (2) enrolling users into the Hyperledger Fabric wallet, and (3) retrieving IoT rules to be consumed by an external CEP engine for real-time event monitoring and contract enforcement.

Step 1: Storing roles on-chain. After deploying and instantiating the smart contract, the SymboleoAC regulator invokes `storeRolesPolicy()` to extract the contract roles from the instantiated contract and store them on-chain. The stored roles record includes a SHA-256 hash.

Step 2: Enrolling users from the stored roles in the ledger. Next, the SymboleoAC API retrieves the stored roles using `getRolePolicy()`. Access to this transaction is restricted to authenticated and authorized identities (in our case study, only the Admin and Regulator roles). The SymboleoAC API verifies integrity by recomputing the hash locally and comparing it against the on-chain SHA-256 hash returned by `getRolePolicy()`. Once verified, the SymboleoAC API iterates over the retrieved roles and registers/enrolls each user through Hyperledger Fabric's Certificate Authority using the utility function `registerAndEnrollUser()`, storing the resulting identities in the wallet.

Step 3: Retrieving IoT rules for CEP. The SymboleoAC API also retrieves the IoT rules using `getIoTCondition()` (restricted to Admin and Regulator). For each DataTransfer declared in SymboleoAC (e.g., temperature and humidity), the transaction returns a rule object that includes: `contractId`, `chaincodeName`, `sensorType`, `sensorId`, the `condition` expression, the `window` and `having` thresholds, and the `chaincodeFunction` that should be invoked when the CEP detects a match (e.g., `trigger_temperature`, `trigger_humidity`). This function triggers the appropriate contractual action, including generating and sending an alert to roles eligible to receive notifications according to the access control policies specified in the contract. The SymboleoAC API also verifies the integrity of the returned rules using SHA-256 and then writes the rules to `rules.json`, enabling the CEP engine to automatically evaluate incoming IoT messages from the message broker and dynamically invoke the correct smart contract transaction for the corresponding contract instance.

Observed output. The results demonstrate the successful execution of the workflow described above. Specifically, the system correctly: (1) stores the contract roles on the ledger, (2) enrolls users into the wallet, and (3) retrieves and exports the IoT rules to `rules.json`. Figure 10.11 and Figure 10.12 show the successful user enrollment and export of IoT rules for the `MeatSale` contract, while Figure 10.13 and Figure 10.14 show the retrieved IoT rules for the `MeatSale` and `VaccineProcurement` contracts. These rules can be used to automatically enroll sensors, configure the CEP engine to deploy EPL rules, and configure the message broker, as described in Section 9.2.

10.4.2 Results – Enrolling IoT devices

For each DataTransfer declared in the SymboleoAC contract, e.g., `temperature` and `humidity` in the Meat Sale case study, a certificate is issued by the Hyperledger Fabric Certificate Authority and stored in the wallet. This enables secure communication between sensors and the message broker, and ensures that the CEP engine accepts data from the message broker only when it originates from trusted sensors.

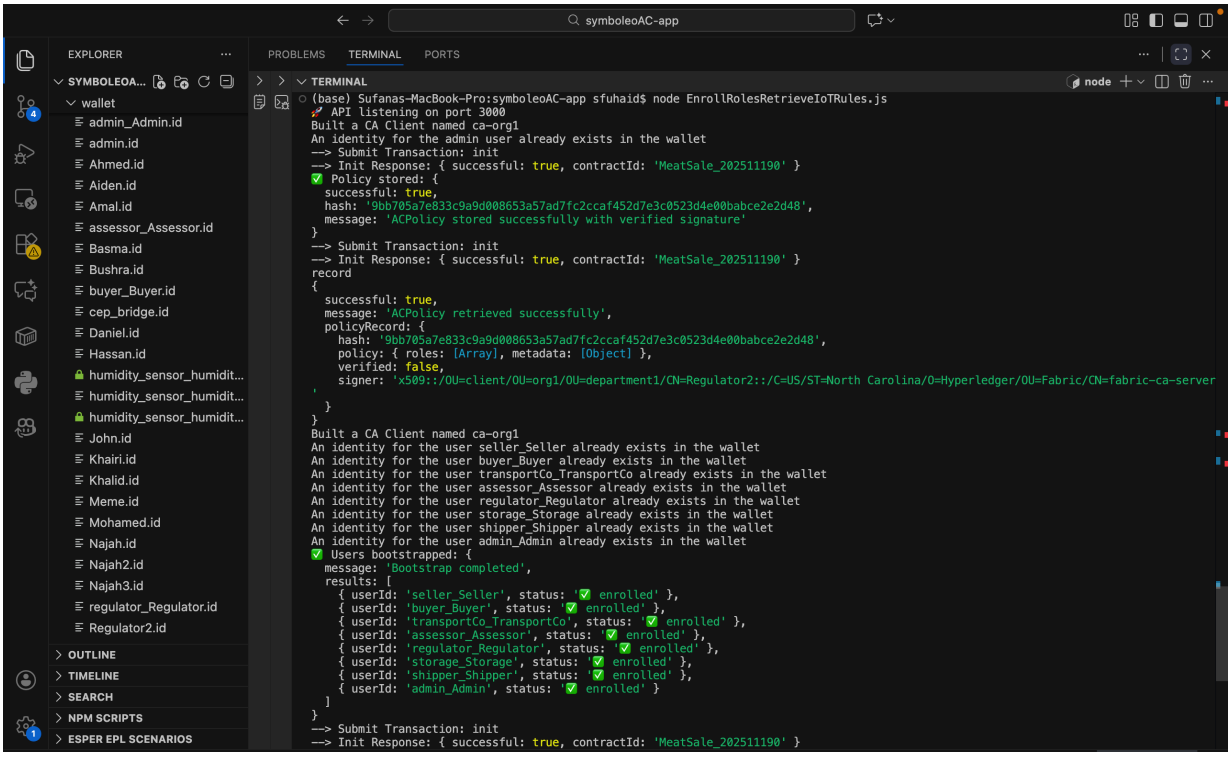


Figure 10.11: Execution logs showing successful role storage on-chain, integrity verification, and user enrollment into the Hyperledger Fabric wallet for the `MeatSale` contract.

Figure 10.12: Execution logs showing retrieval, integrity verification, and export of IoT rules to `rules.json` for CEP and message broker consumption in the `MeatSale` contract.

In Section 10.4.1, we show how **DataTransfer** rules specified in the SymboleoAC contract are generated using SymboleoAC2SC, stored in the ledger, and retrieved by the SymboleoAC API. To enroll sensors, the API invokes method `enrollSensorsFromRules()` from `EnrollSensors.js`, which is restricted to authorized identities (i.e., Admin or Regulator) and enrolls sensors based on the generated `rules.json` file for each contract instance.

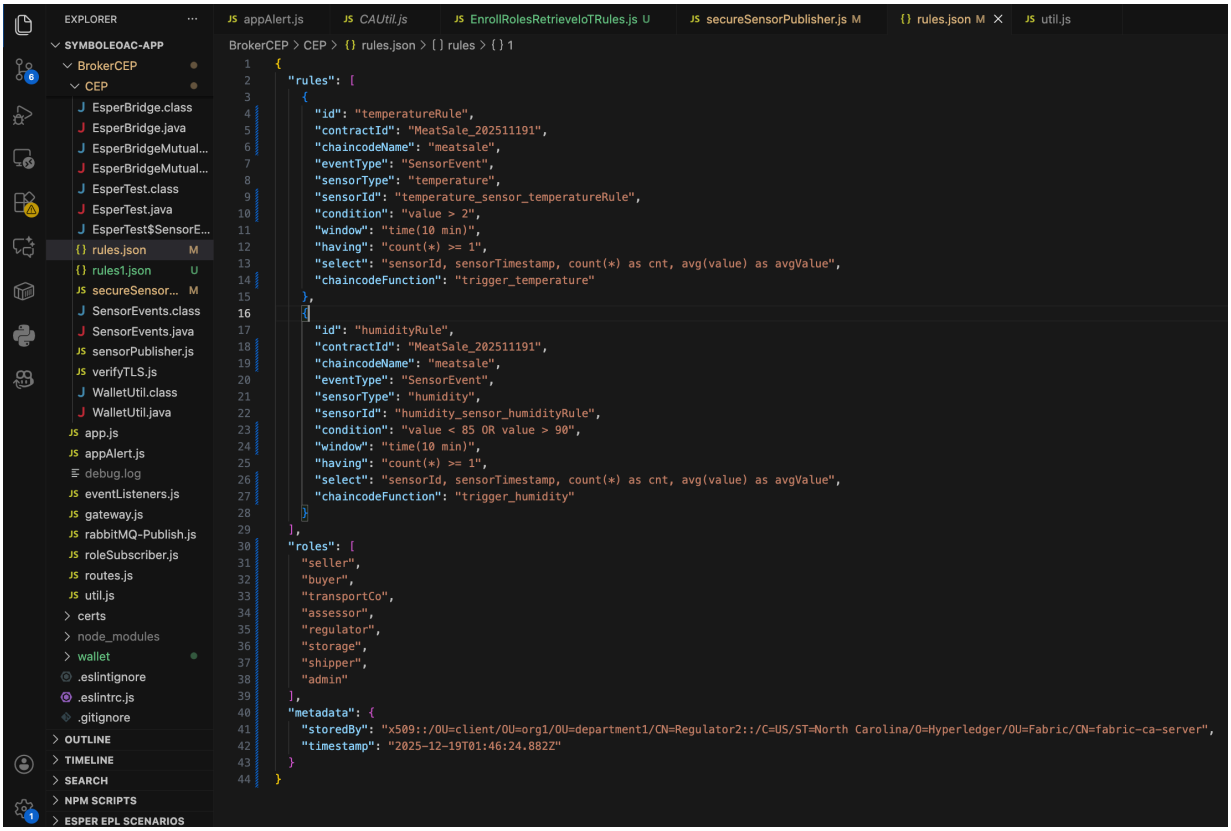


Figure 10.13: Excerpt of the generated `rules.json` file for the `MeatSale` contract, showing the IoT rules derived from SymboleoAC specifications (temperature and humidity). The file also includes metadata identifying the authorized role (Regulator) that stored the rules and the associated timestamp, ensuring traceability and integrity for CEP consumption.

Figure 10.15 shows the enrollment of sensor identities in the Fabric wallet for the Meat Sale contract.

Observed output. The results show successful execution. Specifically, the method `enrollSensorsFromRules()` iterates over the generated DataTransfer rules (i.e., `temperature` and `humidity`) and successfully enrolls each sensor with the Hyperledger Fabric Certificate Authority, storing the resulting identities in the wallet.

10.4.3 Results – CEP Engine and Message Broker Enrollment

To enable secure end-to-end communication in the proposed SymboleoAC architecture (Figure 5.1), both the CEP engine and the message broker must be authenticated components. In addition to enrolling roles and sensors for each SymboleoAC contract, we evaluated the enrollment of the CEP engine (i.e., Esper) and the message broker (i.e., RabbitMQ) using the Hyperledger Fabric Certificate Authority. This step ensures that only authorized components can participate in IoT data consumption, event pattern detection, and alert propagation.

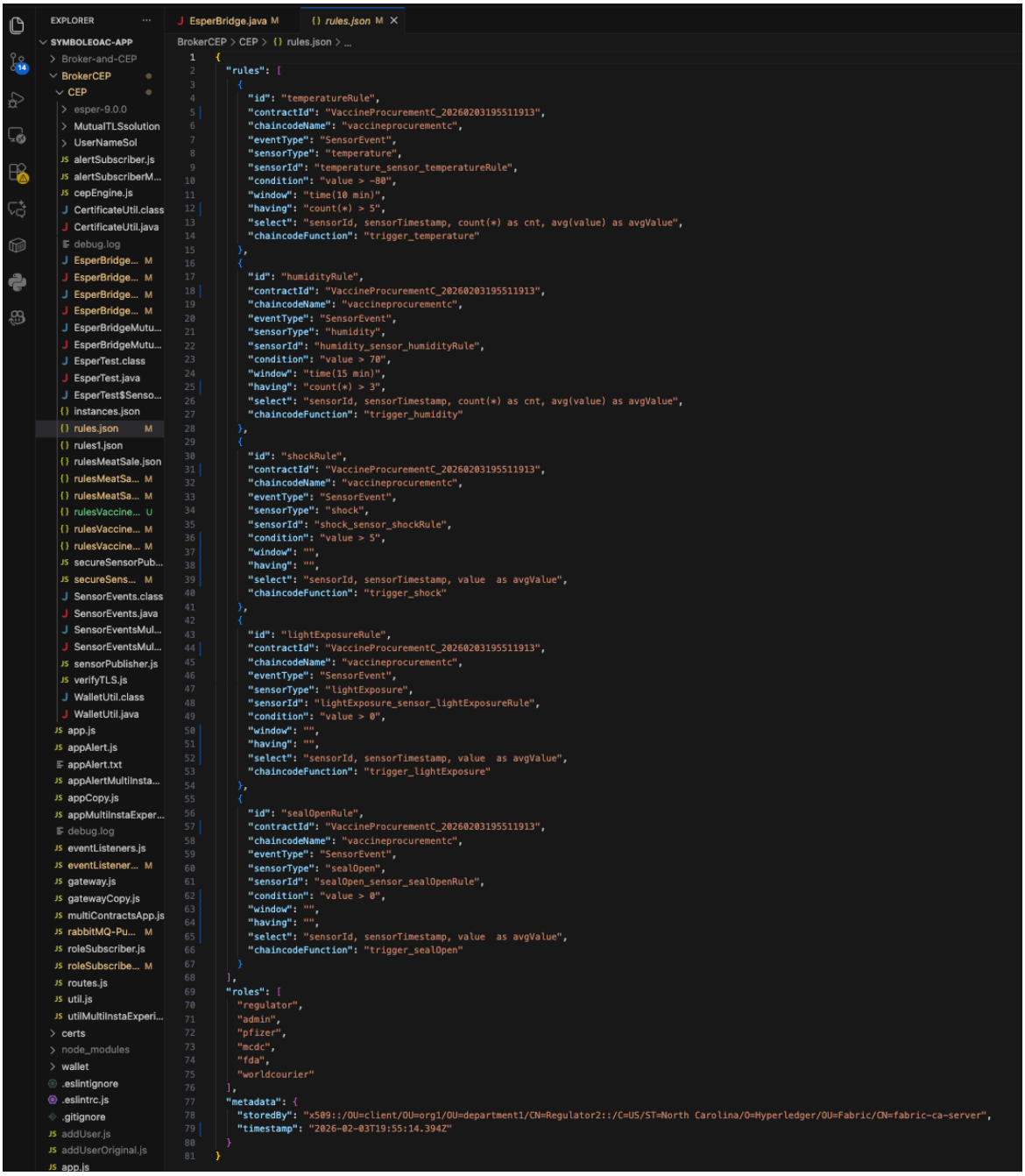


Figure 10.14: Excerpt of the generated `rules.json` file for the `VaccineProcurement` contract, showing the extracted IoT monitoring rules (e.g., temperature, humidity, shock, light exposure, and seal open), including their conditions, windows, thresholds, and the corresponding chaincode functions to invoke.

CEP Engine Enrollment. The CEP engine is enrolled using the `enrollCEPServer()` method from `EnrollCEPServer.js` in the SymboleoAC API. Invocation is restricted to authorized identities (i.e., Admin or Regulator). During enrollment, the CEP engine (Esper) is registered as a distinct identity with the Fabric Certificate Authority as `cep_bridge`,

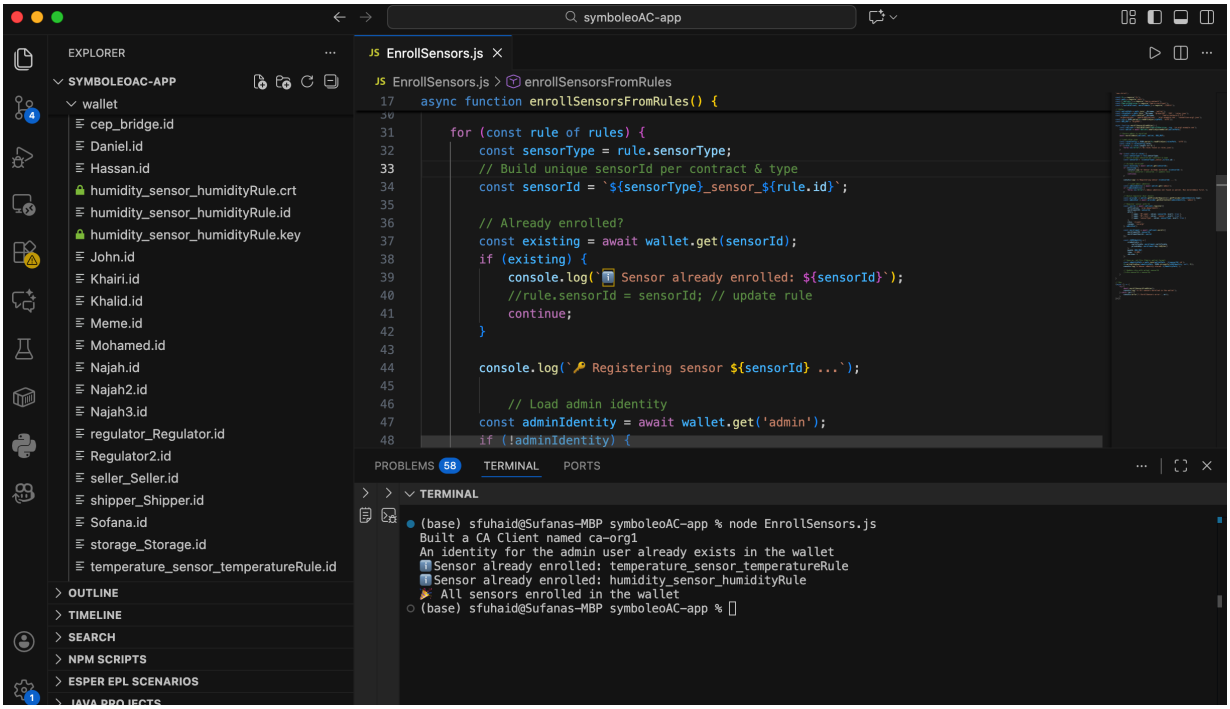


Figure 10.15: Sensor enrollment in the Meat Sale case study, based on `rules.json`.

and an X.509 certificate is issued with the role bridge.

The generated credentials are first stored in the Fabric wallet and then exported to the `certs` directory. This export step is required because the Node.js application can directly consume identities in wallet (`.id`) format, whereas the message broker and the CEP engine require credentials in standard cryptographic formats (i.e., `.key` and `.crt` files, and a keystore, respectively). Consequently, a PKCS#12 keystore (`cep_bridge.p12`) is generated to support Java-based TLS authentication.

Since the CEP engine (Esper) is implemented in Java, it requires credentials to be provided via a keystore rather than as separate PEM files (i.e., `.key` and `.crt` files). The `.p12` file contains the CEP engine's certificate and private key, enabling mutual TLS authentication between the CEP engine and the message broker. As a result, only a trusted and authenticated CEP component is permitted to consume sensor data and emit alerts. Figure 10.16 illustrates the enrollment of the CEP engine.

Message Broker Enrollment. Similarly, the RabbitMQ message broker is enrolled using the `enrollRabbitMQ()` method from `EnrollRabbitMQ.js`. An identity (`rabbitmq-server`) is registered with the Fabric CA and associated with the role server. The issued X.509 certificate and private key are stored in wallet and then exported as PEM files in the `certs` folder and configured within RabbitMQ to enable TLS-based authentication. This ensures that sensors and the CEP can communicate with a trusted and authenticated message broker. Figure 10.17 shows the enrollment of the message broker.

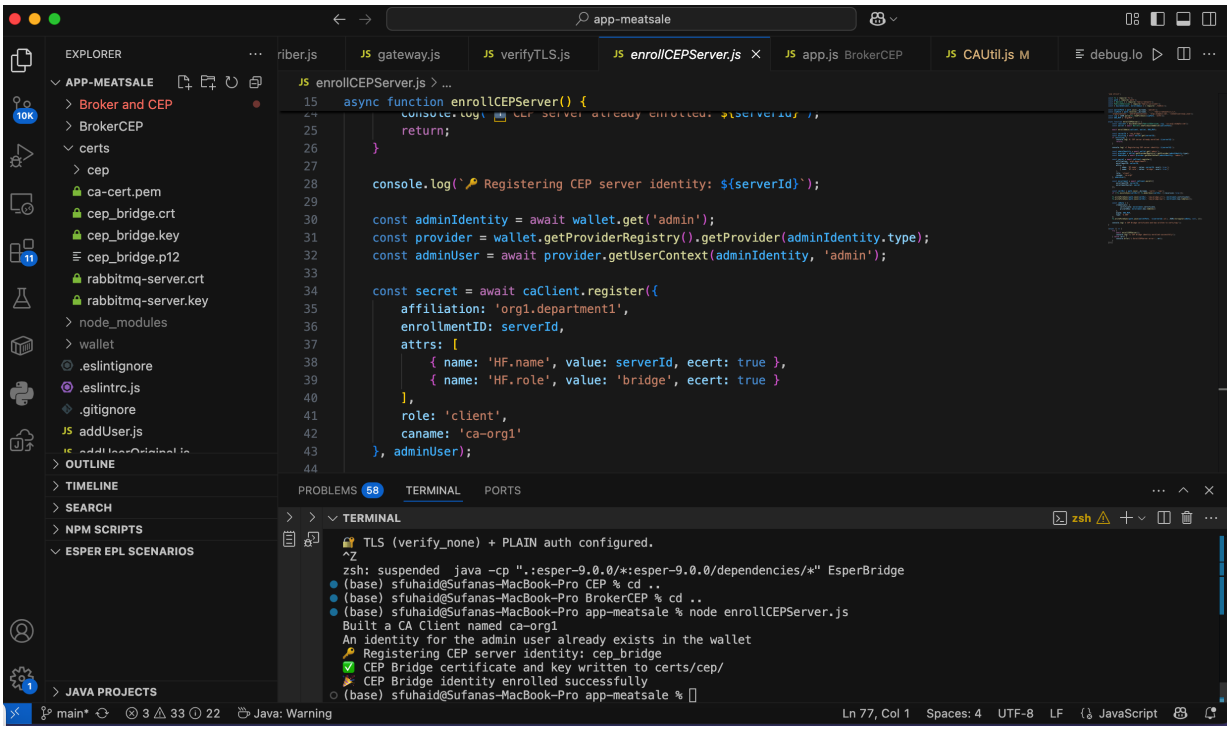


Figure 10.16: Enrolling the CEP bridge (`cep_bridge`) and issuing its X.509 certificate for mutual TLS.

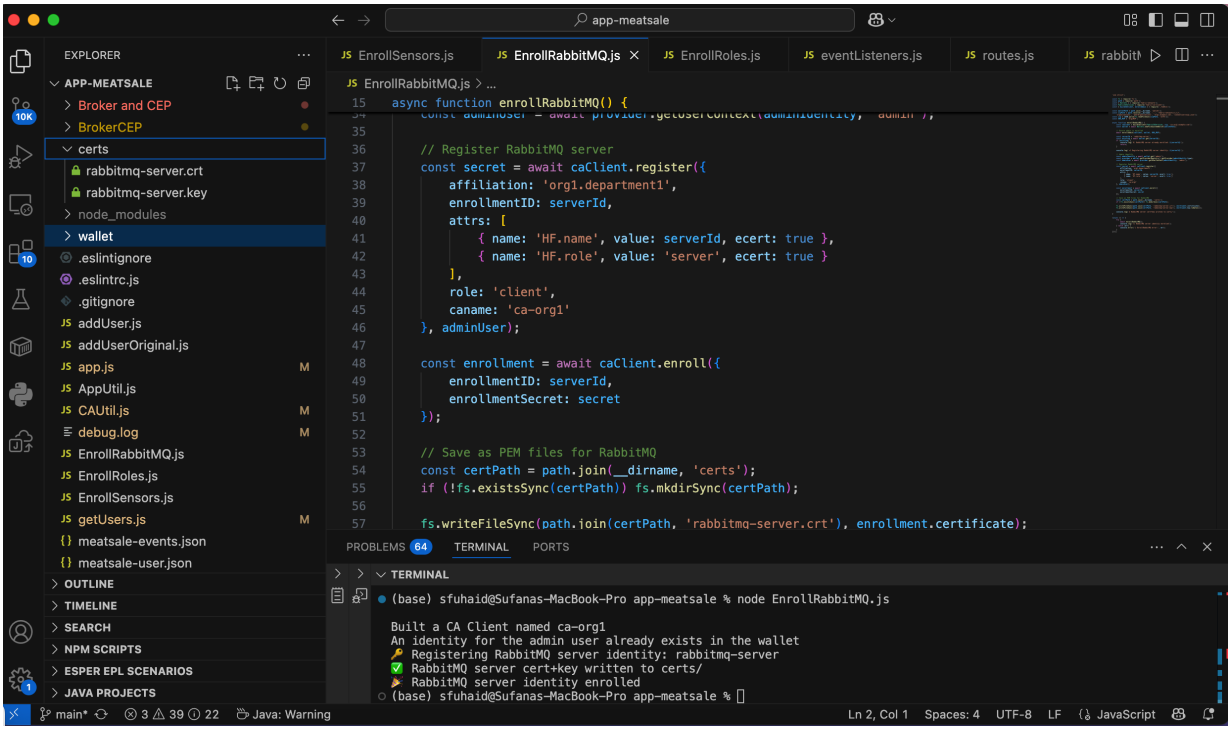


Figure 10.17: Enrolling the RabbitMQ broker (`rabbitmq-server`) and generating its certificate and key for TLS.

Observed output. The results demonstrate that the enrollment process completes successfully for both the CEP engine and the message broker, with identities correctly issued by the Hyperledger Fabric Certificate Authority. All communication paths, namely, sensor to broker, broker to CEP, CEP back to the message broker, broker to smart contract, and smart contract back to the message broker, are protected using certificate-based authentication and mutual TLS.

The results further confirm that the SymboleoAC architecture supports secure integration of off-chain components with on-chain smart contract execution. Importantly, the evaluation demonstrates that the proposed approach supports the assignment of multiple identities for multiple instances of CEP engines and message brokers. Each broker or CEP instance can be enrolled with its own dedicated X.509 identity and associated role.

10.4.4 Results – Notification Generation and Delivery

In this section, we evaluate the notification mechanism generated by SymboleoAC and its run-time enforcement for both case studies. Here, we present the result for the Meat Sale case study. Notification events are generated by the smart contract when protected transactions successfully update the contract state in the ledger and are used to inform authorized roles about state changes of interest.

The emitted notification events are captured by the SymboleoAC API using an event listener, published to a message broker, and delivered to role specific subscribers. The following results examine both successful and unsuccessful notification scenarios in order to demonstrate correct event propagation as well as enforcement of SymboleoAC access control constraints.

Scenario A: Authorized notification flow. In this scenario, `trigger_paid()` and `trigger_unLoaded()` are invoked by an authorized role, namely the buyer and the assessor respectively, who are the designated performers of the paid and unLoaded events defined in Listing 10.1. In this scenario, we assume that the buyer did not pay on time; therefore, the states of violateObligation_payment and violateObligation_delivery are updated to violation by an authorized role, i.e., buyer and seller respectively. Upon successful execution, the smart contract emits notification events containing the relevant payload, including the obligation's previous state, the obligation's transition state, the timestamp, and the authorized roles allowed to subscribe to the notification, as described in Section 9.5.2. In this case, both the seller and the buyer are authorized recipients, as they are pre-authorized according to the access control rules, acting respectively as the performer and the rightholder. These notification events are captured by `startEventListeners() method` from `eventListeners.js` in the SymboleoAC API, published to the `eventExchange` by `rabbitMQ-Publish.js`'s methods using routing keys of the form `role.<role>` as described in Chapter 6, and subsequently delivered to the buyer and the seller, who subscribed to the corresponding queues via the `startPerRoleSubscribers()` method available in `roleSubscriber.js`.

Figure 10.18 illustrates this authorized notification flow showing the smart contract emitting a notification event, its propagation through the SymboleoAC API, and deliv-

ery to the buyer and seller via the role-specific subscribers. The process starts by creating a list of queues in the message broker based on the roles defined in the contract specification. Upon successful execution, the message broker publishes notification messages to the corresponding queues (i.e., the buyer and seller queues). The buyer and seller then subscribe to their respective queues and receive the notifications.

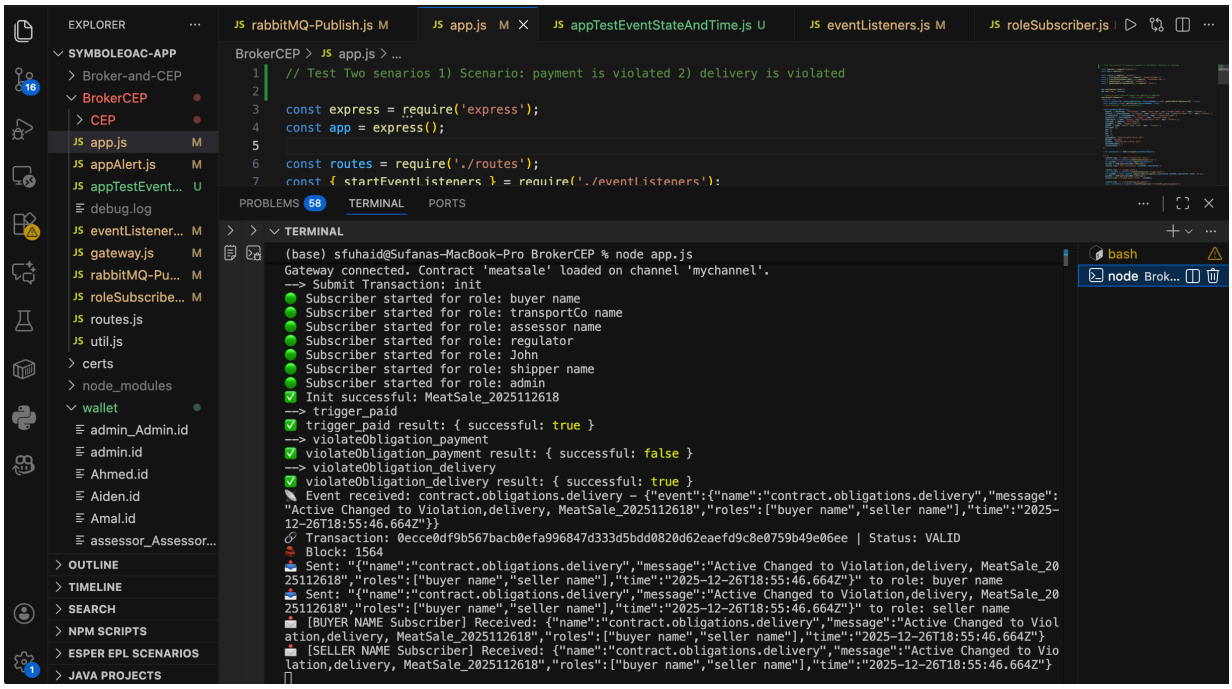


Figure 10.18: Authorized notification workflow in the Meat Sale case study: after a successful state change, the smart contract emits a notification event to the intended roles (buyer and seller).

Scenario B: Unauthorized invocation (access denied). In this scenario, we call the same transaction but with the incorrect role. The trigger_paid() function is invoked by an identity that does not satisfy the required access control (e.g., it is not the legal position's designated performer or controller, or it has not been granted write permission) according to the SymboleoAC policy of the Meat Sale case study. This scenario is used to evaluate the enforcement of the two-layer security mechanism (see Section 9.3). In this case, the smart contract rejects the invocation with an `access denied` error during the test, and the transaction is not committed. Consequently, no notification event is emitted on-chain, and the SymboleoAC API does not publish any message to the RabbitMQ message broker. This confirms that unauthorized roles cannot modify the contract state or trigger notifications for restricted operations. Figure 10.19 shows that a queue has been created per role, and shows the access denied scenario where an unauthorized identity attempts to invoke a protected transaction, with no notification generated to the message broker.

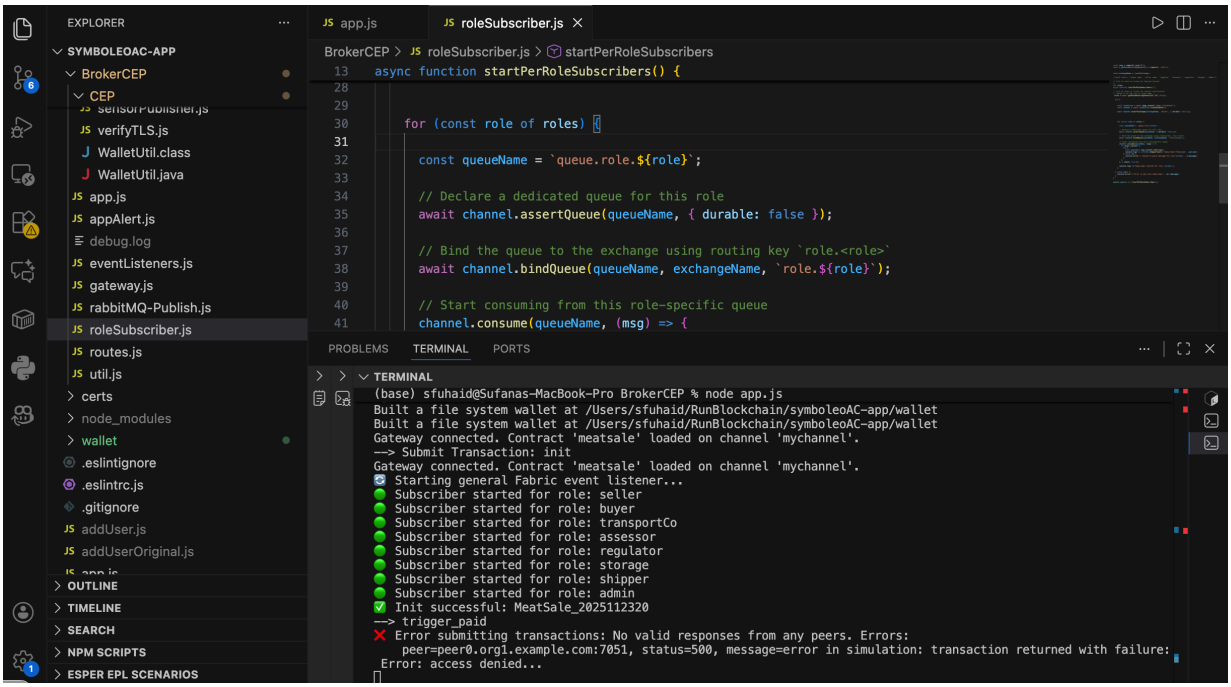


Figure 10.19: Unauthorized invocation of the trigger_paid() transaction in the Meat Sale case study. The smart contract rejects the transaction with an `access denied` error due to violation of SymboleoAC access control rules, preventing state changes, notification event emission, and message broker publication.

10.4.5 Results – Full Cycle Execution Workflow: IoT Data Stream, Broker-CEP Filtering, and Smart Contract Enforcement

To evaluate the execution of DataTransfers defined in the SymboleoAC contract specification, we evaluated both the `Meat Sale` and `Vaccine Procurement` contract case studies to assess the robustness and design choices of the proposed SymboleoAC architecture (see Figure 5.1).

We evaluate our approach for integrating IoT data into smart contract execution through a message broker and a CEP engine, including a security scenario involving a fake (unauthorized) sensor. Specifically, we executed an end-to-end workflow that connects:

1. Virtual IoT sensor readings to RabbitMQ using the SymboleoAC API methods from `secureSensorPublisher.js`;
2. An external CEP engine (Esper) that evaluates the `rules.json` conditions retrieved from the deployed smart contract using the SymboleoAC API methods available from `EsperBridge.java`; and
3. A secure alert subscriber implemented using the SymboleoAC API methods from `alertSubscriber.js`, which forward detected violations back to the smart contract by invoking the corresponding transaction (e.g., `trigger_temperature()`).

4. The smart contract updates the legal positions and contract states accordingly and notifies the authorized roles by issuing alerts and reporting any resulting state transitions. Authorized roles can subsequently consume these alerts and notifications through the broker using the role subscriber API (`roleSubscriber.js`).

In this section, we show the result of the Meat Sale contract.

Scenario A: Authorized IoT sensors. In this scenario, multiple authorized IoT sensors (e.g., temperature and humidity sensors) are enrolled through Hyperledger Fabric and possess valid X.509 identities. These sensors continuously publish readings to the message broker queue `sensor_data` using `secureSensorPublisher.js`'s methods, over TLS, as shown in Figure 10.21. Upon successful authentication, the sensor data are accepted by the message broker and forwarded to the external CEP engine implemented in `EsperBridge.java`, which is running and waiting for sensor data, as shown in Figure 10.20.

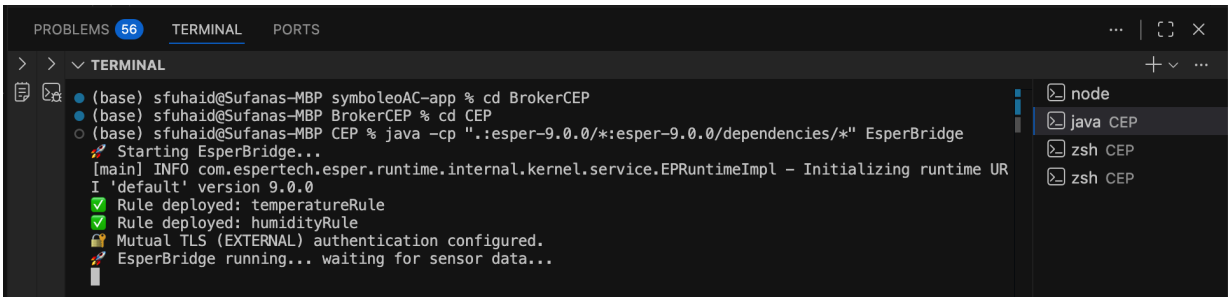


Figure 10.20: Esper CEP initialized and ready, after loading the temperature and humidity rules from `rules.json`, automatically extracted from the contract specification.

The CEP engine dynamically loads the `rules.json` file retrieved from the deployed smart contract and evaluates incoming sensor streams using sliding windows and aggregation values, as shown in Figure 10.22. When a rule condition is satisfied (e.g., repeated threshold violations), the CEP engine generates an alert and securely publishes it back to alerts exchange in the message broker. The alert subscriber, implemented in `alertSubscriber.js`, consumes the alert using mutual TLS authentication, invokes the corresponding smart contract transaction (e.g., `trigger_temperature` or `trigger_humidity`) in Hyperledger Fabric, and generates a notification, as shown in Figure 10.23. The alert subscriber methods available in (`alertSubscriber.js`) starts by creating a queue per role subscriber, and then receives alerts from the CEP engine detecting IoT violations using methods available in `appAlert.js`, triggers smart contract transactions, emits on-chain notification events, and delivers them via RabbitMQ to authorized role-specific subscribers (e.g., seller and regulator) using methods available in `eventListeners.js`, `rabbitMQ-Publish.js`, and `roleSubscriber.js` .

Figure 10.21: Secure publication of multiple IoT sensor readings (temperature and humidity) to RabbitMQ. Authorized sensors successfully publish readings via TLS authentication.

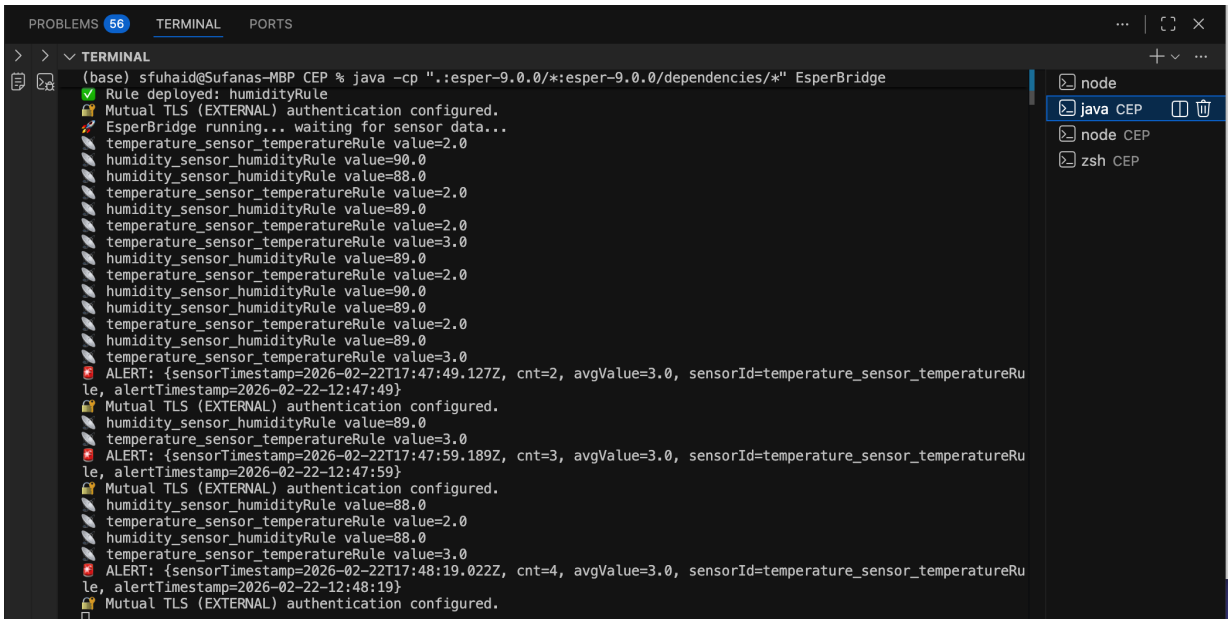


Figure 10.22: The CEP engine evaluates incoming sensor streams against the temperature and humidity rules derived from the SymboleoAC contract specification, showing different `Alerts`.

In Figure 10.23, the envelope icon next to "Alert received" indicates the reception of filtered alerts from the CEP through the RabbitMQ message broker; the satellite icon

next to “Event received” denotes notification event reception from the smart contract; the outgoing envelope icon next to “Message sent” represents notification publication to the message broker; and the subscriber envelope icon next to “Subscriber received” indicates delivery to the corresponding authorized role subscriber queue.

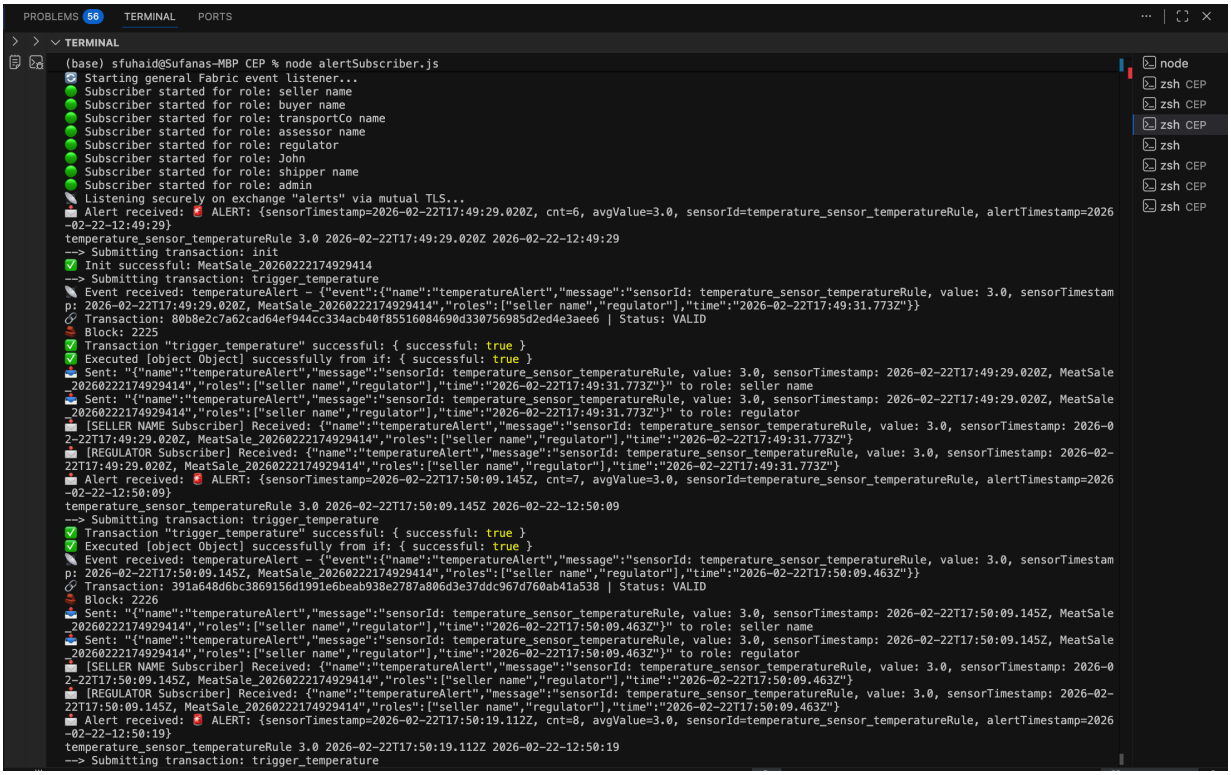


Figure 10.23: End-to-end notification workflow. The alert subscriber receives alerts from the CEP engine detecting IoT violations, and the authorized role triggers smart contract transactions (e.g., `trigger_temperature`), which emit on-chain notification events delivered via the message broker (RabbitMQ) to authorized role-specific subscribers (e.g., seller and regulator).

The results show that continuous sensor readings are correctly aggregated by the CEP engine, violations are detected in real time, and authorized alerts successfully trigger on-chain state transitions. This suggests the correctness of the end-to-end integration between IoT devices, message broker and CEP event processing, and SymboleoAC smart contract execution.

Scenario B: Fake sensor scenario. To evaluate resilience against malicious inputs, we introduce a fake sensor, named `vibration_sensor`, that attempts to publish data using an unregistered sensor identity. Figure 10.24 shows the execution of multiple readings of virtual sensors. The fake sensor fails to authenticate due to the absence of a valid identity in the wallet, and its data are rejected before entering the CEP. Consequently, no alert is generated, no smart contract transaction is invoked, and no notification is emitted. This confirms that unauthorized sensors cannot influence contract execution, even if they attempt to publish valid data.

Figure 10.24: Multiple IoT sensor readings (temperature and humidity) published to RabbitMQ. Authorized sensors transmit data via TLS and password-based authentication, while an unauthorized (fake) sensor (`vibration_sensor`) is rejected due to a missing valid identity.

Scenario C: Unauthorized invocation (access denied). In this scenario, we follow the same setup as Scenario A, except that after the alert queue receives a violation alert from the CEP engine, the role attempting to invoke the **DataTransfer** transaction (e.g., `trigger_humidity()`) is not the designated performer and is not an authorized alert subscriber.

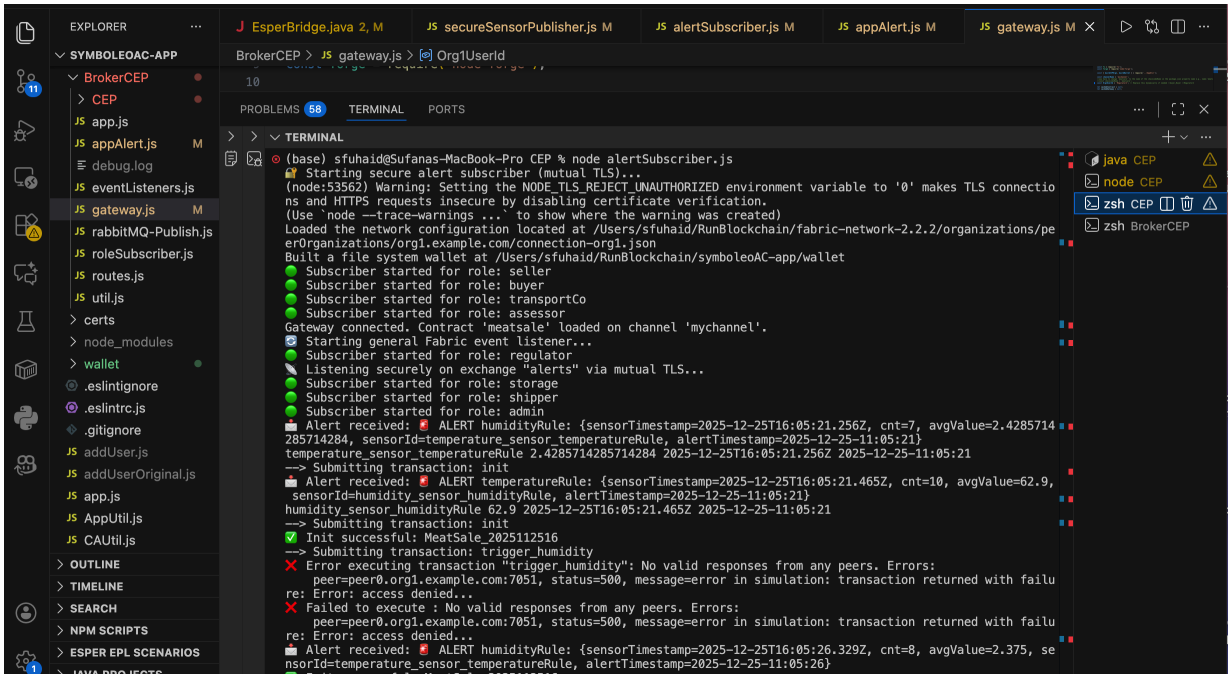


Figure 10.25: Access control enforcement for DataTransfer transactions. An unauthorized identity attempts to invoke the `trigger_humidity` transaction, which is rejected with an `access denied` error because the caller does not match the designated performer (regulator) defined in the SymboleoAC policy. No state change or notification event is committed.

10.5 Multiple Concurrent Instances of Multiple Contracts, with Shared Parties

In this section, we conduct an experiment to evaluate whether the proposed SymboleoAC framework can support running *multiple concurrent instances* of different smart contracts, thereby assessing the generalizability of our approach and tools. In addition, we examine whether multiple contracts can share one or more parties while still maintaining independent contract logic and access control rules. Figure 10.26 shows the experimental setup in which two Meat Sale contract instances (`MeatSale1`, `MeatSale2`) and two Vaccine Procurement contract instances (`Vaccine1`, `Vaccine2`) are deployed simultaneously on the same Hyperledger Fabric network.

10.5.1 Experiment Description

Each contract instance maintains its own `RolePolicy` and associated IoT rules, which are retrieved by the SymboleoAC API and exported into a dedicated `rules.json` file. Also, some parties (buyers) are shared across different contract instances. To perform this experiment, we follow the steps outlined below:

- First, we modified the Meat Sale and Vaccine Procurement contract specifications

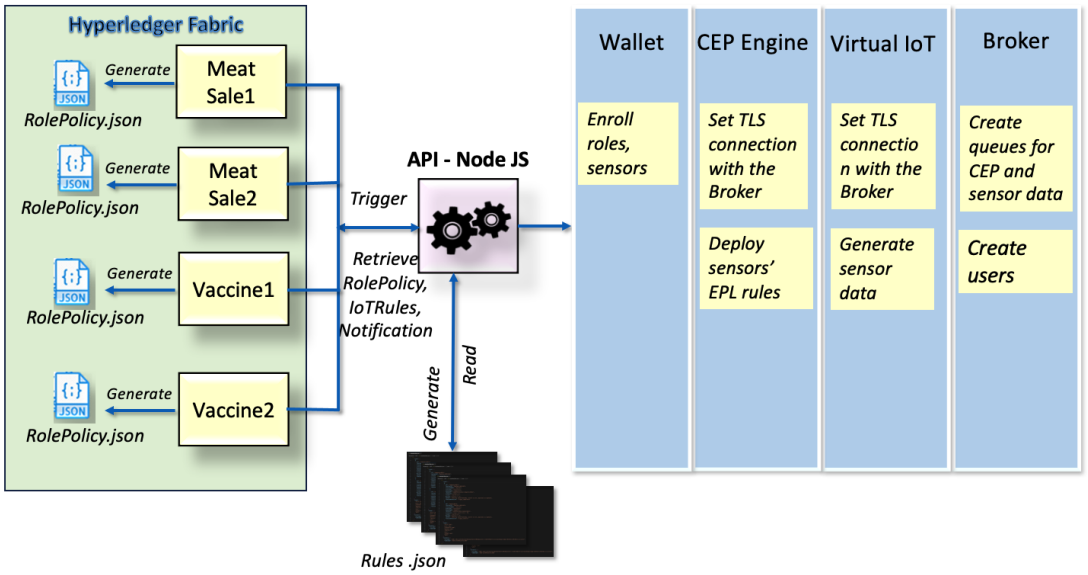


Figure 10.26: Experimental setup for evaluating multiple concurrent smart contract instances in SymboleoAC. The figure illustrates two `MeatSale` instances and two `VaccineProcurement` instances deployed on Hyperledger Fabric, each maintaining independent role polices and IoT rules.

(both available online[23]) so that they share one or more roles across contracts. In particular, the Buyer role is instantiated as a different participant across different Meat Sale contract instances, while a single Buyer participant is shared across multiple Vaccine Procurement contract instances. In contrast, the Seller role is instantiated as a distinct participant in each of the four contract instances along with other roles as well. Consequently, the experimental setup consists of the following instances: $\mathsf{MeatSale}_1(\mathsf{Seller}_1, \mathsf{Buyer}_1)$, $\mathsf{MeatSale}_2(\mathsf{Seller}_2, \mathsf{Buyer}_2)$, $\mathsf{VaccineProcurement}_1(\mathsf{Seller}_3, \mathsf{Buyer}_1)$, and $\mathsf{VaccineProcurement}_2(\mathsf{Seller}_4, \mathsf{Buyer}_1)$. This setup enables a systematic evaluation of the SymboleoAC framework's ability to manage multiple concurrent contract instances while supporting shared parties and instance-specific access control policies. This experiment also demonstrates that the same participant can engage in multiple contracts simultaneously while being subject to contract-specific access control constraints.

- Second, for code generation (SymboleoAC2SC), we ensured that every generated transaction invocation in `index.js` explicitly passes the corresponding `contractId` as an argument, enabling instance-specific execution at run time and when interacting with external services such as through the SymboleoAC API. In addition, we updated the core (SymboleoACJS) library to allow contract instance identifiers to include seconds and milliseconds, ensuring uniqueness across rapidly-created instances and preventing identifier collisions during multi-instance execution.

[23] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-IDE/tree/main/samples

- Third, both smart contracts (chaincodes) were deployed on the same Hyperledger Fabric network using Docker while keeping the default configuration and port assignments (e.g., peer ports `7051/7052`). This confirms that multiple chaincodes can be executed concurrently without requiring port modifications, as the isolation is managed at the chaincode and container level in Docker. Figure 10.27 illustrates all active chaincodes running in Docker, including the original Meat Sale contract (`meatsale`), the original Vaccine Procurement contract (`vaccineprocurementc`), the modified Meat Sale contract (`meatsalesharedparty`), and the modified Vaccine Procurement contract (`vaccineprocurementsharedparty`).

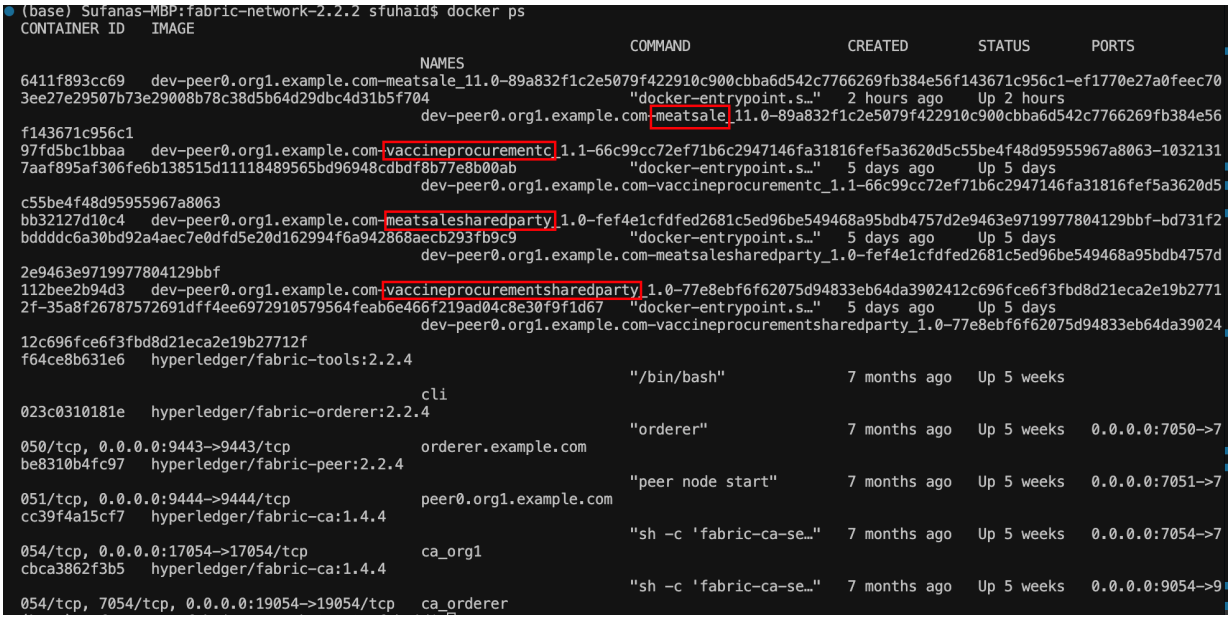


Figure 10.27: Docker containers showing multiple deployed chaincodes running concurrently on the same Hyperledger Fabric network, including the modified Meat Sale and Vaccine Procurement shared-party contracts.

- Fourth, within the SymboleoAC API, we update the run-time parameters used for the Meat Sale contract instances, including the ordered quantity, payment amount, currency, role information, and delivery details (e.g., delivery date). Similarly, we update the corresponding run-time parameters used for the Vaccine Procurement contract instances. This step ensures that contract execution reflects realistic differences between instances and validates that shared-party participation remains consistent under varying contractual terms. Figure 10.28 shows an example of the parameters used for vaccine instances, while the remaining parameters are available online[24]. Note that, for this experiment, we replace the Government role in the Vaccine Procurement contract (Section 10.3) with the Buyer role and its corresponding parameters from the Meat Sale case study, enabling the evaluation of shared parties across contracts.

[24] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-Application-API/blob/main/EnrollRolesRetrieveIoTRulesMultiInstaExperiment.js

```
//parameters vaccine shared party
const initParamsVaccine1 = {
"pfizerP":  {name:"PfizerEU", org:"Pfizer Pharma GmbH", dept: "manufacturing"},
"buyerP": { warehouse: "70 Glouxter", name: "buyer name", org: "Canada Import Inc", dept: "finance" },
"regulatorP": {name: "regulator", org: "Canada Import Inc", dept: "finance"},
"adminP": {name: "admin", org: "org1", dept: "finance"},
"fdaP": {name:"fda", org:"FDA", dept: "inspection"},
"worldcourierP":{name:"worldcourier", org:"worldcourier Company", dept: "logistics"},
"approval": true,
"unitPrice": 19.50,
 "minQuantity": 100,
 "maxQuantity" : 500,
 "temperature":-80
};

//parameters vaccine shared party 2 (another instance)
const initParamsVaccine2 = {
"pfizerP":  {name:"PfizerHQ", org:"Pfizer Global Export Operations", dept: "supplyChain"},
"buyerP": { warehouse: "70 Glouxter", name: "buyer name", org: "Canada Import Inc", dept: "finance" },
"regulatorP": {name: "regulator", org: "Canada Import Inc", dept: "finance"},
"adminP": {name: "admin", org: "org1", dept: "finance"},
"fdaP": {name:"fda", org:"FDA", dept: "inspection"},
"worldcourierP":{name:"worldcourier", org:"worldcourier Company", dept: "logistics"},
"approval": true,
"unitPrice": 25,
 "minQuantity": 200,
 "maxQuantity" : 600,
 "temperature":-80
};
```

Figure 10.28: Example of two `VaccineProcurementSharedParty` contract parameter sets sharing the same Buyer information defined in the Meat Sale case study.

- Fifth, we generalized the role enrollment and IoT rule retrieval procedures described in Sections 10.4.1 by extending the `EnrollRolesRetrieveIoTRules.js` class to support multi-instance execution across multiple chaincodes. The updated script sequentially initializes multiple contract instances (two Meat Sale instances and two Vaccine Procurement instances) by invoking the `init` function and retrieving the corresponding `contractId`. For each instance, `init` is invoked only once, and the Fabric gateway connection is reused for subsequent CEP and message broker.
  - Within the API, the `admin` or the SymboleoAC contract `regulator` invokes `storePolicy()`, which in turn executes the `storeRolesPolicy()` transaction four times (Vaccine_1, Meat_1, Vaccine_2, Meat_2). Each invocation:
    - Creates a new contract instance and returns its `contractId`, and
    - Stores the corresponding role policy (i.e., `RolePolicy.json`, as shown in Figure 10.26) for that instance.
  - The script then retrieves role definitions and registers and enrolls the corresponding user identities in the wallet.
  - Finally, it retrieves the IoT rules associated with each contract instance.

  The script records generated instance identifiers (`contractIds`) in an `instances.json` file[25], enabling subsequent automated CEP processing and message broker configuration. We successfully completed the registration and enrollment of users, as well as the retrieval of IoT rules for each contract instance. For simplicity, we do not include new screenshots, as the process is similar to what was shown in Section 10.4.1.

[25] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-Application-API/blob/main/BrokerCEP/CEP/instances.json

Figure 10.29a shows the generated `instances.json` file, while Figure 10.29b further illustrates the generated IoT rules for one contract instance. The remaining rule files are available online[26].

- Sixth, we updated the sensor enrollment script discussed in Sections 10.4.2. During sensor registration and enrollment, we integrate the sensor names defined in the specification with the corresponding contract instance identifier. This enables the system to automatically identify incoming sensor data and correctly associate it with the appropriate contract instance. For simplicity, we do not include new screenshots, as the process is similar to what was shown in Section 10.4.2.

- Seventh, we added the sensors associated with each contract instance to the message broker. Using the sensor names automatically generated from the specification, we manually registered these sensors as users in the message broker, assigned passwords, and configured their access permissions, including the exchanges and queues they are authorized to publish to, as discussed earlier. Figure 10.30 illustrates the newly added sensors from each contract instance in the RabbitMQ metadata.

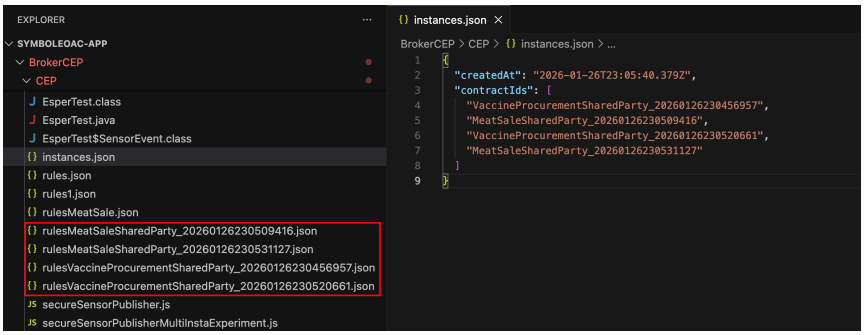
- Finally, we updated the script that implements the complete data streaming pipeline, from IoT sensors to the message broker, then to the CEP engine, back to the message broker, and finally to the smart contract, as discussed in Sections 10.4.5. Specifically, we extended the `secureSensorPublisher.js` script to send virtual sensor data for four contract instances. In addition, we updated the CEP configuration script `EsperBridge.java` so it automatically reads the list of active instances from `instances.json`, retrieves the relevant `contractId`s, and iterates over the per-instance rule files. For each contract instance, the CEP engine dynamically generates the corresponding EPL rules, loads them into the engine, and deploys them for execution. Figure 10.31 illustrates the deployment of 14 sensors across four different contract instances, automatically generated from the `rulescontractId.json` files associated with each instance. Figure 10.32 shows the successful initialization of role-based subscribers queues for all users across multiple contract instances in the message broker. The SymboleoAC API files for the multi-instance experiment are available online[27], with the prefix MultiInstaExperiment added to the file names.

10.5.2 Results – Validation Scenarios

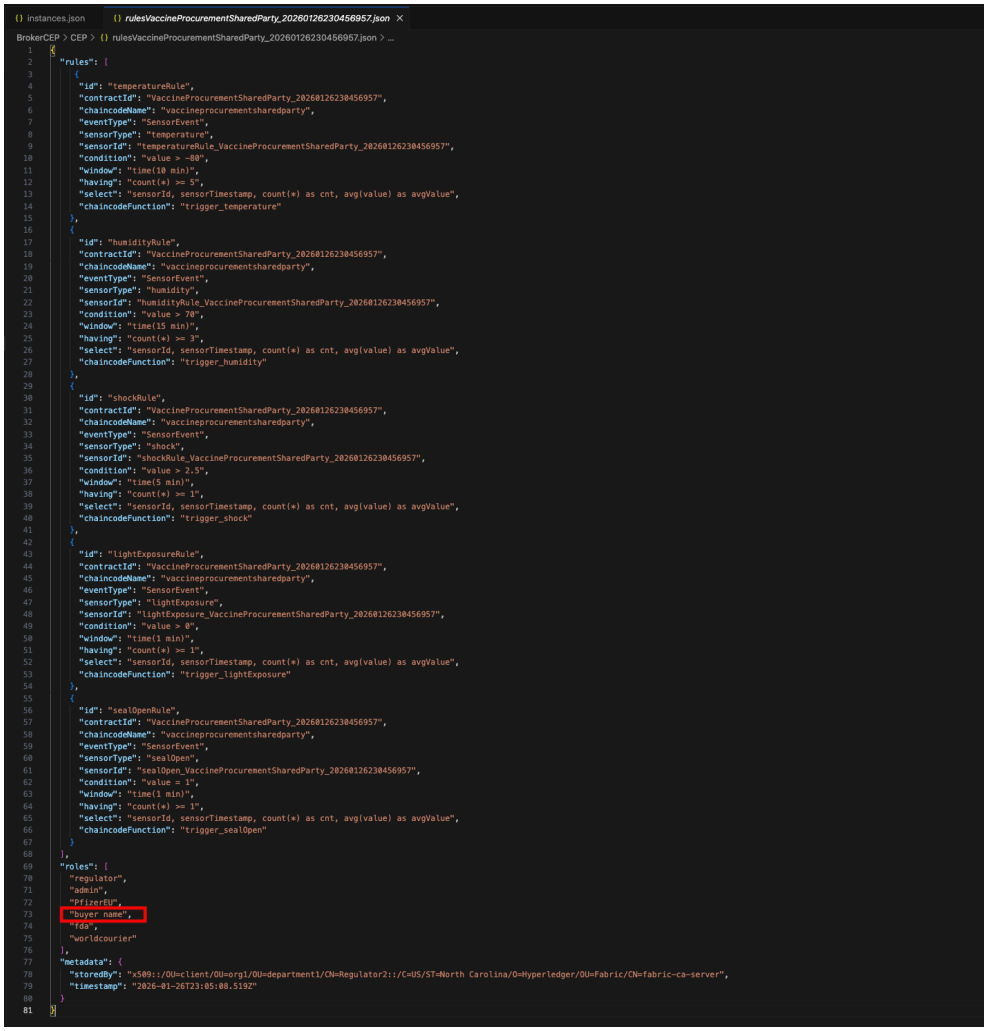
To evaluate the visibility, isolation, and correctness of running multiple instances of multiple contracts concurrently, we conducted a set of validation scenarios. The objective is to check whether each sensor reading is correctly associated with its corresponding contract instance, that CEP-generated alerts trigger the appropriate smart contract transaction (e.g., `trigger_lightExposure` from the Vaccine Procurement instances) only when invoked by an

[26] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-Application-API/tree/main/BrokerCEP/CEP

[27] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-Application-API



(a) Generated `instances.json` file (left), listing all created contract instances (`contractId`s), and the corresponding per-instance IoT rule files (right).



(b) Example of a generated IoT rules file for a Vaccine Procurement contract instance (e.g., `rulesVaccineProcurementSharedParty_20260126230456957.json`), illustrating instance-specific CEP rules, the associated `contractId`, the target chaincode, and the authorized roles.

Figure 10.29: Automatically generated `instances.json` and per-instance IoT rule files produced by the multi-instance enrollment and rule-retrieval process.

```
(base) sfuhaid@Sufanas-MBP ~ % rabbitmqctl list_users

Listing users ...
user    tags
humidityVaccine20260126230520661        []
humidityMeat20260126230531127   []
lightExposure_sensor_lightExposureRule  []
sealOpen_sensor_sealOpenRule    []
shockVaccine20260126230520661   []
tempMeat20260126230531127       []
sealOpenVaccine20260126230456957        []
lightVaccine20260126230456957   []
humidityMeat20260126230509416   []
shockRuleVaccine20260126230456957       []
cep_bridge      []
lightVaccine20260126230520661   []
tempVaccine20260126230520661    []
tempMeat20260126230509416       []
guest   [administrator]
sealOpenVaccine20260126230520661        []
shock_sensor_shockRule  []
humidityRuleVaccine20260126230456957    []
humidity_sensor_humidityRule    []
rabbitmq-server []
tempRuleVaccine20260126230456957        []
temperature_sensor_tempRule    _[]
```

Figure 10.30: Registered sensor users in RabbitMQ for multiple contract instances. Each sensor is automatically named based on the generated SymboleoAC specification and manually registered as a message broker user with specific access permissions.

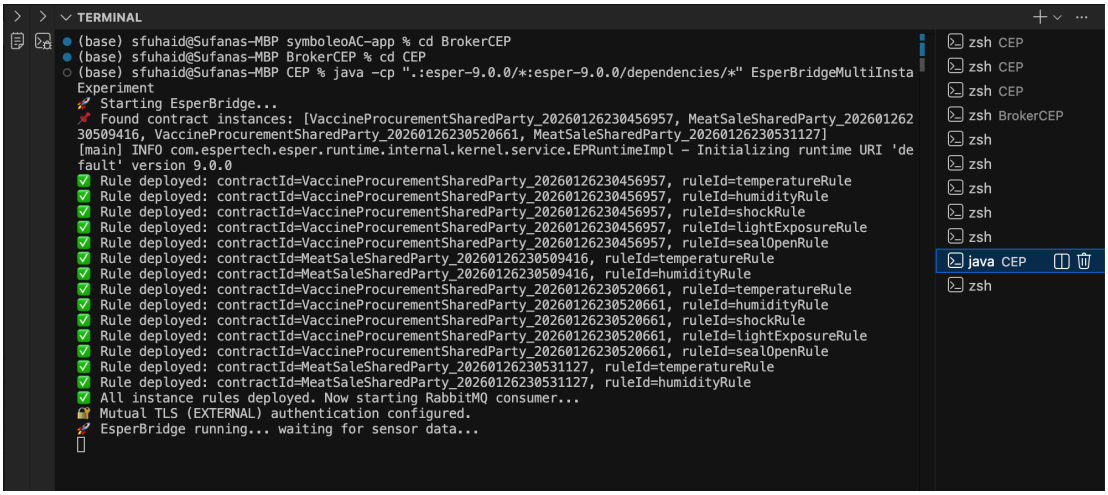


Figure 10.31: Output of the CEP engine during the multi-instance experiment. The CEP engine automatically reads the list of contract instances from `instances.json`, deploys the corresponding EPL rules for each instance.

authorized role, and that notifications are delivered only to the authorized subscribers of that specific instance broker per-role queue (e.g., Seller, and PfizerEU, Regulator).

For the experiment, we deployed a total of 14 sensors across different contract instances, as shown in Figure 10.31. However, for the testing scenarios, we focused on demonstrating one sensor per instance sending data to the message broker, being filtered by the CEP engine, triggering a smart contract transaction, and generating a notification event, as summarized in Table 10.6. The result screenshots are shown in Appendix E.

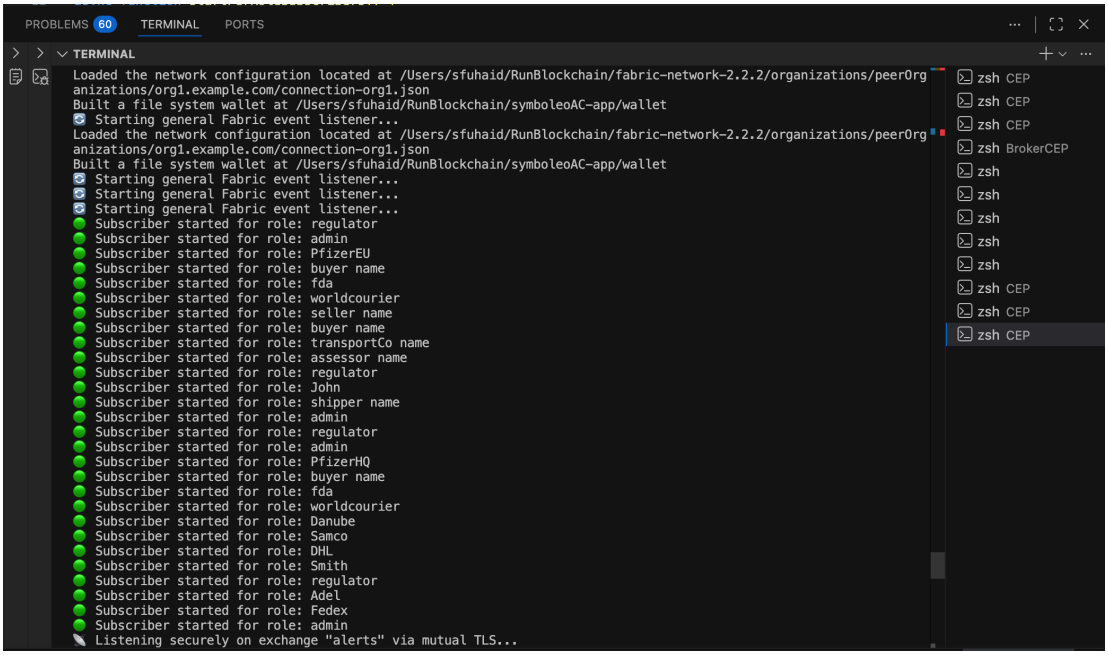


Figure 10.32: Broker initialization showing subscribers for all roles across multiple smart contract instances. For each contract instance, role-based subscribers (e.g., regulator, admin, buyer, shipper, assessor) are automatically started and securely connected to the `alerts` exchange via mutual TLS.

Table 10.6: Virtual sensors per contract instance triggering transactions and notifying authorized roles in the experiment.

Sensor ID	Contract Instance	Triggered Transaction	Notification Delivered to Roles
lightExposure_Vaccine_20260126230456957	Vaccine_20260126230456957	`trigger_lightExposure`	Regulator, PfizerEU
temperature_Vaccine_20260126230520661	Vaccine_20260126230520661	`trigger_temperature`	Regulator, PfizerEU
temperature_MeatSale_20260126230509416	MeatSale_20260126230509416	`trigger_temperature`	Seller, Regulator
temperature_MeatSale_20260126230531127	MeatSale_20260126230531127	`trigger_temperature`	Danube, Regulator

Scenario 1: Vaccine Procurement – Light exposure (Instance 1)

In this scenario, the virtual sensor `lightExposure_Vaccine_20260126230456957` publishes readings to the message broker using TLS and sensor credentials, as shown in Figure E.1. The CEP engine evaluates the corresponding EPL rule and detects a light exposure violation, as shown in Figure E.2. The alert subscriber then consumes the CEP alert, invokes the correct transaction (`trigger_lightExposure()`) on the corresponding contract instance, and finally the smart contract emits an on-chain notification event that is delivered only to the authorized subscribers for that instance (Regulator and PfizerEU), as shown in Figure E.3.

Scenario 2: Vaccine Procurement – Temperature (Instance 2)

In this scenario, the virtual sensor `temperature_Vaccine_20260126230520661` publishes temperature readings securely to the message broker, as shown in Figure E.1. The CEP engine evaluates the incoming IoT data and detects no violation for this instance, as shown in Figure E.2; therefore, no alert is issued. As a result, the alert subscriber does not catch any alert and consequently does not invoke the corresponding transaction (i.e., `trigger_temperature()`) on the target contract instance. Subscribers associated with that instance do not receive any notification, as shown in Figure E.3.

Scenario 3: Meat Sale – Temperature (Instance 1)

In this scenario, the virtual sensor `temperature_MeatSale_20260126230509416` sends data to the message broker securely, as shown in Figure E.1. The CEP engine detects a temperature violation and publishes an alert for this specific instance, as shown in Figure E.2. The alert subscriber consumes the alert and invokes `trigger_temperature()` on the matching smart contract instance. The smart contract then emits a notification event, which is delivered only to the authorized roles for that instance (e.g., Seller and Regulator), as shown in Figure E.3.

Scenario 4: Meat Sale – Temperature (Instance 2)

This scenario repeats the same workflow for a different Meat Sale instance, using the sensor `temperature_MeatSale_20260126230531127`, as shown in Figure E.1. The key objective is to confirm isolation across instances: the CEP alert is generated for the correct instance, as shown in Figure E.2, the API invokes `trigger_temperature` on that instance only, and notifications are delivered only to the subscribers (i.e., Danube and Regulator) authorized for that instance, queue as shown in Figure E.3.

Observed Results

The results confirm the correctness and isolation of the multi-instance experiment, which in turn supports the viability of the SymboleoAC architecture for deploying multiple instances of different cyber-physical smart contracts. The CEP engine generated alerts only when violation conditions were satisfied and produced no false positives (e.g., Scenario 2). When violations occurred, the alert subscriber invoked the correct smart contract transaction for the corresponding instance only. Finally, notification events were delivered exclusively to the authorized roles of that specific contract instance, with no cross-instance interference observed. These results validate instance-level isolation, accurate CEP filtering, and secure role-based notification delivery.

10.6 Conclusion

To sum up, the results obtained from multiple variants of the two case studies indicate that SymboleoAC2SC is capable of generating smart contracts with embedded access control according to the SymboleoAC specification, and is also able to interact with the outside world in a dynamic way. To ensure that the generated smart contracts operate correctly, we deployed them in a real blockchain environment and conducted multiple unit tests covering a wide range of possible scenarios. The smart contracts were successfully deployed, correctly invoked by authorized roles, consumed events from the outside world through the message broker and CEP engine, and generated events in a secure and trusted manner.

The generated smart contracts for the Meat Sale and Vaccine Procurement case studies, as well as the upgraded contract specifications with shared parties, are available online[28]. In addition, the SymboleoAC specifications are available online[29], as well as the reusable library[30], the SymboleoAC API[31], and the blockchain testing environment[32].

The next chapter reflects on our experience with the developed artifacts, including the limitations of our approach, comparisons with related work, and threats to validity.

[28] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-IDE

[29] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-IDE/tree/main/samples

[30] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-JS-Core

[31] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-Application-API

[32] https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC-HyperledgerFabric-Test-Netwrok

Chapter 11

Discussion

This chapter reflects on SymboleoAC and further analyzes some of the results obtained so far. It also discusses the similarities and differences between the proposed work and closely related research presented in Chapter 3, from the perspectives of the ontology, architecture, and tooling methodology. The chapter also discusses the limitations of the proposed approach and the associated threats to validity.

11.1 Reflective Analysis

This section provides a reflective analysis of SymboleoAC and its supporting artifacts, including the SymboleoAC architecture, the SymboleoACJS library, the SymboleoAC2SC code generator, and the SymboleoAC API. The objective of this reflection is to examine the design decisions, integration challenges, and strengths throughout the development and evaluation of SymboleoAC and these artifacts. While the next sections analyze quantitative aspects such as size and complexity, this section focuses on qualitative insights and architectural implications.

The SymboleoAC architecture (Chapters 5 and 6) was designed to enable the deployment of legal smart contracts in cyber-physical systems. When designing the architecture, we ensured a clear separation of concerns between on-chain logic (smart contracts and access control enforcement) and off-chain components (IoT sensors, message broker, CEP engine, and API layer). This separation proved to be essential for achieving scalability, maintainability, and extensibility. The layered architecture of SymboleoAC, comprising the code generation layer, runtime layer, and the integration layer, enables the independent evolution of the architecture. Changes in a SymboleoAC contract specification automatically propagate through SymboleoAC2SC to the generated smart contract, while the SymboleoACJS library provides shared semantics and access control enforcement. Off-chain components are configured and interact through artifacts generated by the SymboleoAC API, in addition to limited one-time manual setup, as discussed in Section 9.2. Also, this architectural separation reinforces extensibility. For example, new case studies (e.g., shared party contracts or multi-instance contracts) were introduced without redesigning

the core architecture. Instead, they were supported by extending specification constructs and regenerating the corresponding artifacts, as illustrated in Section 10.5.

The SymboleoACJS library plays a foundational role by encapsulating the SymboleoAC ontology and runtime semantics of SymboleoAC. Rather than fully generating contracts from scratch, SymboleoAC2SC produces contract with embedded access control code that depends on this reusable runtime library.

The SymboleoAC2SC code generator is central to the thesis contribution. The expressiveness of SymboleoAC allows concise specifications to produce complex runtime behavior. Embedding access control rules directly into generated methods eliminates manual policy implementation errors. Automatic generation of IoT rules enables seamless integration with CEP and message broker components.

The SymboleoAC API serves as the off-chain middle ware layer that operationalizes the generated contracts. From a reflective standpoint, the API illustrates the practical complexity of deploying CPSCs in real environments, with real components. While smart contracts enforce on-chain logic, significant infrastructure is required to securely connect external components. The API encapsulates this complexity, allowing contracts to remain at a specification level and focused on domain semantics. The API also revealed the boundary between automation and manual configuration. While IoT rules, access control policies, and event notifications are generated automatically from specifications, certain tasks, such as message broker installation, CEP dependency configuration, and initial Docker deployment, remain one-time manual efforts. Clarifying these boundaries improves reproducibility and clarity.

Additionally, the evaluation of the full execution cycle, from IoT data publication to message broker transmission and CEP filtering, smart contract invocation, and notification delivery, demonstrated the feasibility of authenticated, policy-controlled, and event-driven contract enforcement. The integration of message broker and CEP components reduces on-chain computational cost while preserving correctness through controlled filtering and rule evaluation for IoT sensors.

Supporting multiple contract instances and shared party configurations further validated the flexibility of the architecture. These experiments showed that the framework can maintain independent contract logic and access control policies even when parties overlap across instances of contracts, whether the latter are the same or different ones. However, integrating multiple distributed components also highlighted synchronization and fault-handling challenges. Coordinating IoT data streams, CEP rule evaluation, blockchain transaction latency, and event notifications requires careful arrangement. These challenges emphasize that CPSC systems inherently involve distributed systems complexity beyond traditional smart contracts.

Overall, SymboleoAC and its related artifacts show that it is feasible to operationalize formal contract specifications in a CPSC environment.

11.2 Size and Complexity

Smart contracts were successfully generated from compact SymboleoAC specifications using SymboleoAC2SC. Without such specifications and tools, generating equivalent smart contracts and component configurations would represent substantial implementation effort. In the Meat Sale case study, the contract specification comprises only 64 Lines of Code (LOC), whereas the generated smart contract contains 2,312 executable JavaScript LOC (excluding comments, blank lines, the SymboleoACJS reusable library, and the SymboleoAC API), as shown in Figure 11.1. In addition, as illustrated in Figure 11.3 the SymboleoACJS core library consists of 22 JavaScript classes and approximately 4,500 executable LOC, providing the foundational access control mechanisms and runtime support required by the generated contracts and utility functions. A similar ratio (approximately **1:28**, specification LOC to generated JavaScript LOC) was also observed for the Vaccine Procurement contract, from 121 specification LOC to 3,354 generated JavaScript LOC, as shown in Figure 11.2. This ratio is twice that of the plain Symboleo artifacts produced in the past (1:14 in [81, 101]), which means that properly handling access control, event notifications, and interactions with off-chain components essentially doubled the size of the code needed. Moreover, this also demonstrates the expressiveness of SymboleoAC's syntax, which allows concise specifications to produce large and feature-rich implementations.

File	Code	Comment	Blank	Total
./domain/assets/Meat.js	9	0	3	12
./domain/assets/PerishableGood.js	14	0	2	16
./domain/contract/MeatSale.js	178	3	28	209
./domain/events/Delivered.js	12	0	1	13
./domain/events/InspectedQuality.js	13	0	1	14
./domain/events/Notified.js	10	0	0	10
./domain/events/Paid.js	15	0	1	16
./domain/events/PaidLate.js	14	0	1	15
./domain/events/PasswordNotification.js	11	0	1	12
./domain/events/UnLoaded.js	10	0	1	11
./domain/events/datatransfer/Alert.js	16	0	1	17
./domain/roles/Admin.js	13	0	2	15
./domain/roles/Assessor.js	13	0	2	15
./domain/roles/Buyer.js	14	0	2	16
./domain/roles/Regulator.js	13	0	2	15
./domain/roles/Seller.js	14	0	2	16
./domain/roles/Shipper.js	13	0	2	15
./domain/roles/Storage.js	14	0	2	16
./domain/roles/TransportCo.js	13	0	2	15
./domain/types/Currency.js	5	0	0	5
./domain/types/MeatQuality.js	6	0	0	6
./events.js	235	19	15	269
./index.js	1332	142	210	1684
./serializer.js	325	22	54	401
TOTAL	2312	186	335	2833

Figure 11.1: LOC breakdowns for the generated Meat Sale smart contract, showing executable code, comment lines, blank lines, and total lines per generated file, along with overall totals.

Furthermore, the proposed transformation approach reduces the complexity of the SymboleoAC2SC process by enabling a direct mapping between specified access control rules in SymboleoAC and their corresponding executable method implementations.

File	Code	Comment	Blank	Total
./src-gen/VaccineProcurementC/domain/assets/PaidAmount.js	12	0	2	14
./src-gen/VaccineProcurementC/domain/assets/Remain.js	12	0	2	14
./src-gen/VaccineProcurementC/domain/assets/VaccineDose.js	13	0	2	15
./src-gen/VaccineProcurementC/domain/contract/VaccineProcurementC.js	208	3	42	253
./src-gen/VaccineProcurementC/domain/events/Agreed.js	11	0	1	12
./src-gen/VaccineProcurementC/domain/events/Confirmed.js	12	0	1	13
./src-gen/VaccineProcurementC/domain/events/Delivered.js	14	0	1	15
./src-gen/VaccineProcurementC/domain/events/Invoiced.js	13	0	1	14
./src-gen/VaccineProcurementC/domain/events/LeadtimeInformedNegotiated.js	12	0	1	13
./src-gen/VaccineProcurementC/domain/events/Notified.js	10	0	0	10
./src-gen/VaccineProcurementC/domain/events/NotifiedOfDelivery.js	12	0	1	13
./src-gen/VaccineProcurementC/domain/events/Paid.js	12	0	1	13
./src-gen/VaccineProcurementC/domain/events/Requested.js	13	0	1	14
./src-gen/VaccineProcurementC/domain/events/Risk.js	12	0	1	13
./src-gen/VaccineProcurementC/domain/events/StopWork.js	10	0	1	11
./src-gen/VaccineProcurementC/domain/events/TerminateAgreementG.js	10	0	1	11
./src-gen/VaccineProcurementC/domain/events/TerminateAgreementM.js	10	0	1	11
./src-gen/VaccineProcurementC/domain/events/ThirdPartyStopWork.js	10	0	1	11
./src-gen/VaccineProcurementC/domain/events/WithdrewApproval.js	10	0	1	11
./src-gen/VaccineProcurementC/domain/events/datatransfer/Alert.js	17	0	1	18
./src-gen/VaccineProcurementC/domain/roles/Admin.js	13	0	2	15
./src-gen/VaccineProcurementC/domain/roles/FDA.js	13	0	2	15
./src-gen/VaccineProcurementC/domain/roles/Government.js	13	0	2	15
./src-gen/VaccineProcurementC/domain/roles/Manufacturer.js	13	0	2	15
./src-gen/VaccineProcurementC/domain/roles/Regulator.js	13	0	2	15
./src-gen/VaccineProcurementC/domain/roles/WorldCourier.js	13	0	2	15
./src-gen/VaccineProcurementC/domain/types/Location.js	6	0	0	6
./src-gen/VaccineProcurementC/events.js	345	30	30	405
./src-gen/VaccineProcurementC/index.js	2151	239	290	2680
./src-gen/VaccineProcurementC/serializer.js	341	22	54	417
TOTAL	3354	294	449	4097

Figure 11.2: LOC breakdowns for the generated Vaccine Procurement smart contract, showing executable code, comment lines, blank lines, and total lines per generated file, along with overall totals.

File	Code
./ACPolicy.js	863
./AbstractEvent.js	8
./Asset.js	141
./Attribute.js	21
./DataTransfer.js	109
./Event.js	115
./InternalEvents.js	52
./LegalPosition.js	328
./LegalSituation.js	100
./Obligation.js	398
./Operation.js	91
./Party.js	416
./Power.js	402
./Predicates.js	103
./Resource.js	191
./Role.js	248
./Rule.js	128
./Situation.js	16
./StateTransition.js	21
./SymboleoContract.js	731
./TimeInterval.js	7
./TimePoint.js	2
TOTAL	4491

Figure 11.3: Executable LOC breakdowns of the SymboleoACJS library. The figure reports per-file code size and the overall total of 4,491 LOC.

The JavaScript code generated by SymboleoAC2SC is complete and immediately deployable, requiring no manual intervention. This substantially reduces development effort and time, minimizes implementation errors, and streamlines the overall smart contract development workflow. In addition to generating classes with all required attributes and methods, the SymboleoACJS framework provides the run-time infrastructure needed to execute SymboleoAC contracts by managing both explicit access control and implicit

access control mechanisms, such as pre-authorization rules, and notifications, as well as by securely communicating with off-chain components, including the message broker and CEP engine, and invoking roles in accordance with the formal SymboleoAC semantics.

The SymboleoAC API plays a critical role in operationalizing the proposed SymboleoAC architecture by incorporating all off-chain components, including role enrollment and identity management, secure transaction invocation, event subscription, and integration with external components such as IoT sensors, message brokers, and CEP engines. The SymboleoAC API size reflects the breadth of infrastructure required to support secure, event-driven, and IoT-aware smart contracts rather than the complexity of individual contracts. Importantly, as for the SymboleoACJS library, this code is implemented once and reused across all contract specifications, thereby further reducing development cost and preventing the duplication of integration logic for each contract. This API also reduces the cost of blockchain execution, as it is costly to have all functionality deployed on-chain.

The lines of code associated with the SymboleoAC API (Figure 11.4), including role and sensor enrollment, broker and CEP integration, and secure event handling, represent a one-time engineering effort comprising 1,547 executable lines of code. The only contract specific artifact within the off-chain layer is the rules.json configuration file, which is automatically generated based on the contract specification.

File	Code
AppUtil.js	32
CAUtil.js	139
enrollCEPServer.js	62
EnrollRabbitMQ.js	53
EnrollRolesRetrieveIoTRules.js	207
EnrollSensors.js	73
BrokerCEP/app.js	85
BrokerCEP/appAlert.js	122
BrokerCEP/eventListeners.js	29
BrokerCEP/gateway.js	121
BrokerCEP/rabbitMQ-Publish.js	24
BrokerCEP/roleSubscriber.js	31
BrokerCEP/util.js	26
BrokerCEP/CEP/alertSubscriber.js	147
BrokerCEP/CEP/WalletUtil.java	30
BrokerCEP/CEP/SensorEvents.java	23
BrokerCEP/CEP/secureSensorPublisher.js	69
BrokerCEP/CEP/EsperBridge.java	194
BrokerCEP/CEP/rules.json	80
TOTAL	1547

Figure 11.4: Breakdown of lines of code (LOC) for the SymboleoAC API, showing executable code lines overall totals.

Globally, this means that a SymboleoAC specification of size S LOC will likely lead to the generation of validated JavaScript code of size $28 \times S$ LOC, which builds on top of a reusable and validated run-time infrastructure (library and API) of about 6,000 LOC.

The SymboleoAC2SC compiler currently targets the Hyperledger Fabric blockchain, an open-source, modular, and permissioned platform. This choice is motivated by the characteristics of the application domains considered in this thesis, where multiple stakeholders operate within a controlled trust environment and where transaction volume and operational costs are critical concerns. Hyperledger Fabric's permissioned architecture enables efficient access control, scalable participation, and cost-effective execution, making it a suitable foundation for achieving access-controlled, IoT-aware smart contracts in cyber-physical environments.

In addition, the SymboleoAC API, together with the message broker and CEP components, facilitate controlled and secure interaction with the external environment, including IoT sensors and off-chain services. By isolating integration logic from contract specifications and generated code, the framework simplifies communication with the outside world. This layered architecture supports the evolution and maintenance of event-driven cyber-physical smart contract systems and facilitates the integration with external components and services.

11.3 Contract Monitoring and Enforcement

The approach proposed in this thesis enables contract monitoring and enforcement by combining formal contract specifications with access control and off-chain integration, thereby addressing research questions **RQ2**, **RQ3**, and **RQ4**. Unlike traditional smart contracts that primarily focus on transaction execution, our SymboleoAC solution ensures that contract monitoring is an integral part of contract enforcement. Monitoring is achieved not only by persisting contract states and events on the blockchain ledger, but also by explicitly controlling who is authorized to observe, query, and react to contract evolution. Through SymboleoAC, access to contract resources such as states, events, attributes, and other resources is restricted to authorized roles, ensuring that monitoring activities themselves comply with contractual privacy and security requirements. In addition, the integration of notification mechanisms allows relevant roles to be informed of contract obligation or power fulfillment, violations, or other contract state changes of interest, thereby improving transparency and responsiveness during contract execution.

Furthermore, contract monitoring in SymboleoAC extends beyond on-chain state inspection to include the controlled ingestion and evaluation of off-chain data, particularly from IoT sensors, through message brokers, and CEP engines, directly supporting **RQ1** and **RQ2**. By specifying access control over sensor data, message broker topics, and event sources, and by relying on certificate-based authentication, the framework ensures that only trusted components can contribute to contract monitoring and enforcement. Sensor readings are filtered and correlated using CEP, and EPL rule violation alerts are forwarded to the smart contract, reducing on-chain overhead while preserving security and correctness. As a result, violations can be detected early, enforcement actions can be triggered automatically, and accountability is strengthened by recording both contract relevant events and the identities of authorized actors involved. Together, these capabilities demonstrate how SymboleoAC enables secure, fine grained, and scalable monitoring.

11.4 Comparison with Related Work

A review of similar CPSC systems and existing RBAC model approaches was provided in Chapter 3. In this section we highlight the benefits of SymboleoAC and its associated components, including SymboleoAC2SC, SymboleoACJS the SymboleoAC architecture, and the SymboleoAC API, in comparison with existing approaches, using the same criteria adopted in Tables 3.3 and 3.7 about related work.

Tables 11.1 and 11.2 evaluate the SymboleoAC approach. The effectiveness of SymboleoAC2SC stems from the richness of SymboleoAC specifications, which describe contractual behavior and access control at a high level of abstraction. This richness enables the generator to produce fully functional smart contract code rather than incomplete templates. Contract-specific code is generated that use the SymboleoACJS library and the SymboleoAC API, which together provide the shared run-time support required for execution. Consequently, the resulting smart contracts can be deployed directly without the need for additional manual coding.

Table 11.1 positions SymboleoAC within the other CPSC systems summarized in Chapter 3. In contrast to existing approaches that typically separate access control from smart contract logic, SymboleoAC adopts a hybrid data architecture with on-chain enforcement and off-chain event management through a message broker (publish/subscribe mechanism). The use of Hyperledger Fabric and Docker-based deployment further demonstrates practical feasibility beyond conceptual modeling. Unlike many related systems that either focus solely on blockchain logic or external event handling, SymboleoAC integrates hybrid event management with formal contract specifications and automated code generation, thereby offering a deployable CPSC solution.

Table 11.1: Assessment of SymboleoAC against the CPSC literature criteria from Table 3.3.

Year	**Data Location**	**SC Location**	**Platform**	**Deployment**	**SC Languages**	**Events**	**Event Mngt. Approach**
2026	hybrid	on-chain	Hyperledger Fabric	Docker	JavaScript	hybrid	publish/ subscribe

Additionally, the existing works include approaches focusing on restricting access to states or assets only, whereas SymboleoAC supports restrictions on many other types of contractual resources. Existing approaches remain at the modeling stage and do not extend to code generation for implementing access control policies (e.g., in smart contracts, or in other CPSC components). Additionally, in SymboleoAC, we introduced a dedicated SymboleoAC architecture that supports integration with other Cyber-Physical Smart Contract components and extends access control to enable secure communication across system layers. In addition, SymboleoAC is equipped with a comprehensive toolchain, including an IDE that facilitates the specification of access control and contractual rules, automated code generation through SymboleoAC2SC, a reusable dependent runtime library (SymboleoACJS), and a SymboleoAC API that operationalizes the generated contracts and manages their interaction with off-chain components.

Table 11.2: Assessment of SymboleoAC against the access control criteria from Table 3.7.

Domain	AC Model	Onto.	Modeling	Code Gen.	Lang./Tool	Protected Resources
Legal Contract	Decentralized	Yes	Xtext	Yes	JavaScript, Java	Obligation, power, event, state, transition, asset, role, attribute, operation, IoT

11.5 Limitations

While this thesis addresses several challenges toward achieving the stated objectives, and while SymboleoAC goes well beyond the existing state of the art in many ways, this thesis also presents limitations that remain open for future work.

- *Security scope.* Our role-based access control approach does not address social engineering attacks. If an attacker gains access to valid credentials (e.g., a password or private key) outside the system, there is little that RBAC alone can do to prevent misuse. While some forms of social attacks may be partially mitigated through well-defined access control policies, by restricting what compromised users can do, others, such as phishing or identity theft, remain outside the protective scope of our framework. From a more holistic and standards-based perspective, according the core functions identified in the *NIST Cybersecurity Framework 2.0* [93] (from 2024), this thesis primarily focuses on the *Govern*, *Identify*, and *Protect* functions through identity management, authentication, authorization, access control policies, encrypted communication, and permissioned blockchain mechanisms. However, physical attacks on IoT devices, hardware tampering, auditing, social engineering, incident response and recovery, as well as broader organizational security concerns remain outside the scope of this work.

- *Deployment.* Our current implementation assumes a permissioned blockchain network (Hyperledger Fabric) with a trusted Certificate Authority. The framework has not been adapted to or evaluated in public blockchain environments such as Ethereum.

- *Automation.* The code generation process currently requires some manual deployment steps (most only needing to be done once, independently of the number of contracts) on a blockchain platform with off-chain components. A fully automated pipeline for deployment and testing remains an area for future work.

- *Scalability.* The evaluation focuses on small- to medium-scale deployments involving a limited number of contracts, roles, sensors, and CEP rules. While the SymboleoAC architecture is designed to be extensible, we have not evaluated its performance and scalability under large scale scenarios with hundreds of concurrent sensors, contracts, or high-frequency event streams. Moreover, support for multiple CEP engines and message brokers (another common scalability strategy) is not yet available.

- *Performance.* The introduction of multiple security layers including certificate-based authentication, SymboleoAC access control checks, message brokering, and CEP processing may introduce run-time latency. Although this latency is acceptable for safety assurance and compliance, a detailed performance and latency analysis has not been conducted.

- *Static role assignment and permissions.* Roles and permissions are defined at design time in the SymboleoAC specification. Dynamic role evolution (e.g., role delegation, revocation based on context, and so on) is not currently supported and is open for future enhancement.

- *Fault Tolerance.* The current evaluation does not address failure of off-chain components, such as message broker outages or CEP crashes. Handling failures of off-chain components also remains part of future work.

- *Reliability.* While the generated code, API, and integration components were thoroughly tested at the unit and end-to-end system levels, this thesis does not provide formal verification that the generated smart contracts and distributed interactions are reliable and robust (e.g., to components that stop working properly). The evaluation primarily focused on technical feasibility and correct execution behavior from a practical perspective rather than on exhaustive formal verification of all possible system states and interactions.

- *Quantum-related threats.* The current implementation of SymboleoAC architecture relies on classical cryptographic mechanisms, including TLS 1.3, X.509 certificates, and blockchain cryptography, which may become vulnerable to upcoming generations of quantum computers. Emerging quantum-related threats, such as *Harvest Now, Decrypt Later*, where encrypted communications collected today may be decrypted in the future [21, 79], and *Trust Now, Forge Later*, where digital signatures created today could be forged retroactively [54], were not mitigated explicitly in this thesis. While post-quantum security mechanisms are not currently incorporated into the SymboleoAC architecture, substituting existing protocols and encryption mechanisms with quantum-safe mechanisms should be possible within our framework.

11.6 Threats to Validity

This thesis is subject to several threats to validity. A threat to *external validity* arises from the limited number of case studies evaluated and the restricted set of application domains considered. Although we evaluated the SymboleoAC architecture, code generation, and supporting tools using multiple variants of the Meat Sale and Vaccine Procurement contracts with different levels of complexity in supply chain scenarios, these case studies cover only a subset of possible application domains. As a result, the observed results may not fully generalize to other domains or contract types.

An additional threat to *external validity* comes from the dependency on the underlying technologies and deployment environment, including the blockchain platform, message

broker, CEP engine, and runtime infrastructure (e.g., Docker and Node.js). While Docker provides a reproducible deployment setup, the current implementation of SymboleoAC is integrated with Hyperledger Fabric, RabbitMQ as the message broker, and Esper as the CEP engine. Although it is conceptually possible to deploy the proposed architecture on other blockchain platforms (e.g., Ethereum), alternative message brokers, or different CEP engines, doing so would require substantial reengineering tailored to the target platform, language, and execution environment.

Specifically, different blockchain platforms rely on: (1) different smart contract languages (e.g., Solidity, Go, Java, or JavaScript), (2) different identity management models (account-based versus certificate-based), (3) different execution environments (e.g., Ethereum Virtual machine versus Fabric chaincode), (4) platform-specific APIs and software development environments, and (5) different transaction models. Similarly, adopting alternative message brokers or CEP engines would require adapting communication mechanisms, event and rule formats, as well as integration logic.

Consequently, migrating SymboleoAC to another blockchain environment, message broker, or CEP engine would require rewriting the generated smart contracts (e.g., in Solidity for Ethereum[1]), changing identity registration and management mechanisms, and modifying the integration with the Node.js, as well as adapting broker and CEP interfaces. While these challenges do not hinder the conceptual validity of the proposed approach, they limit the immediate generalizability of the current implementation across different blockchain platforms, message brokers, CEP engines, and runtime environments.

Internal validity concerns arise from the fact that our evaluation relies on multiple technologies, including smart contracts, a message broker, a CEP engine, and Node.js, each with its own configuration parameters, programming languages, and runtime. Misconfigurations or implementation bugs in any of these components could influence the observed results. To mitigate this threat, we validated and tested each component independently and relied on cryptographic hashes to verify correct execution and ensure data integrity.

Another threat to *internal validity* arises from the development and evaluation process itself. The artifacts presented in this thesis, including the SymboleoAC language extensions, SymboleoAC2SC, SymboleoACJS, and the SymboleoAC API, were primarily designed, implemented, and tested by the thesis author, with periodic inspections and feedback provided by the supervisors, and were also used and tested by Master's students involved in uOttawa's Contract Specification Modelling Lab. However, the lack of a larger number of independent developers or third-party evaluators may limit the objectivity of the results. This threat is partially mitigated by the peer-reviewed publications produced from this thesis and by the open access to the tools' source code and tests, but it could further be mitigated in future work by encouraging external developers to use, evaluate, and extend SymboleoAC.

A threat to *construct validity* arises from whether the selected case studies adequately operationalize the concept of secure and IoT aware smart contracts. In this work, security and IoT awareness are realized through formal access-control specifications, event-driven

[1] https://www.soliditylang.org/

monitoring, CEP-based filtering of sensor data, and secure integration with a blockchain platform via a message broker. While this approach covers key aspects such as security and automation, other dimensions, such as usability are not evaluated. In future work, this threat could be mitigated by conducting a usability study that compares our approach with other IoT-enabled smart contract approaches.

Chapter 12

Conclusion and Future Work

This chapter concludes the thesis, reflects on the thesis research questions and contributions, and discusses directions for future work.

12.1 Answers to the RQs and Contributions

This thesis addresses four research questions (Section 1.4) concerned with the specification, security, automation, and deployment of Cyber-Physical Smart Contracts (CPSCs). Together, these questions span the architectural, language, semantic, and tooling challenges required to support secure legal CPSCs. The main contributions of the thesis collectively provide answers to research questions **RQ1**–**RQ4**, as summarized below.

RQ1: What is the CPS architecture needed to deploy legal smart contracts?

To answer **RQ1**, this thesis reviewed existing architectures from the literature (Chapter 3), and then proposes a CPSC architectural framework (Chapter 5), together with a set of supporting tools for configuring, deploying, and executing legal smart contracts in cyber-physical environments (Chapter 6). The architecture integrates on-chain and off-chain components, including blockchain based smart contracts, message brokers, Complex Event Processing (CEP) engines, and IoT devices, enabling secure and reactive contract execution. The API application provided in Chapter 6, with an implementation based on a specific message broker (RabbitMQ) and a specific CEP (Esper), helps configure, enroll, and orchestrate these components.

At the core of this architecture is the SymboleoAC specification language (Chapter 8), used at design time for formally modeling Symboleo contracts with access control and execution logic. The architecture helps ensure that contract monitoring, enforcement, and interaction with physical devices are performed in a trustworthy, secure, and automated way. Evaluation results demonstrate that the proposed architecture enforces access control policies at run-time, ensuring that only authorized roles can access or update contract resources during execution. This architecture supports the secure deployment and operation of CPSCs in the outside world.

RQ2: How can SymboleO specifications be extended to generate smart contract code that monitors for compliance the execution of a contract, semi-automates selected steps of the execution, and controls the execution?

To address **RQ2**, this thesis extends the SymboleO specification language, both syntactically and semantically, resulting in SymboleOAC (Chapter 8). While the original SymboleO language supports monitoring and verification of legal contract execution, it lacks constructs and automation support for execution control, interaction, and system-level automation.

SymboleOAC fills this gap by introducing (i) variable assignment within events used by contractual obligations and powers, and (ii) notification mechanisms that securely propagate contract state (including violations, fulfillment, and event) as notification events. These notifications allow smart contracts and off-chain components to react dynamically to changes in contract state, thereby supporting semi-automated execution across the CPSC ecosystem.

RQ3: How can SymboleO be extended to specify security and privacy quality criteria when assessing compliance?

In response to **RQ3**, this thesis introduces the SymboleOAC access control ontology (Chapter 7) for contract specification, explicitly addressing security and privacy concerns in compliance assessment and enforcement. Specifically, we have combined RBAC with SymboleO's original ontology to enable defining rules and restricting access to contract resources (including assets, events, and their attributes and operations).

The semantics of SymboleOAC also includes rules that determine the controller of each resource, and default pre-authorization rules that come into effect when a resource is instantiated. By embedding these access control rules directly into the formal contract model, SymboleOAC ensures that compliance monitoring is not only legally correct but also security aware. This integration allows access control policies to be assessed, enforced, and audited as part of the contract execution itself, rather than being delegated to external systems.

RQ4: How can the newly extended SymboleO specifications be converted into cyber-physical smart contracts using automated code generation?

To answer **RQ4**, this thesis presents an automated code generation approach (Chapter 9) that converts SymboleOAC specifications into executable CPSCs. This includes the development of SymboleOACJS, a reusable JavaScript library that operationalizes the SymboleOAC ontology and semantics, and a compiler that generates smart contract code (SymboleOAC2SC) for the Hyperledger Fabric platform.

The generated code is compliant with both implicitly and explicitly specified access control rules and interoperates with the proposed CPSC architecture in SymboleoAC API, including message brokers and CEP components. This automation, extensively evaluated and validated in Chapter 10, ensures consistency between design-time specifications and run-time execution, reduces implementation efforts and manual error, and enables the effective deployment of secure CPSCs.

The many contributions, artifacts, and publications produced through this thesis are already listed in Section 1.6.

Collectively, the contributions of this thesis provide an end-to-end solution for modeling, securing, and executing legal smart contracts in cyber-physical systems. By combining architectural design (**RQ1**), specification language and semantic extensions (**RQ2** and **RQ3**), and automated code generation (**RQ4**), this work advances the state of the art in trustworthy, security-aware, and IoT-enabled smart contract engineering.

12.2 Future Work

Several important items remain for future work, along different themes.

Language and Tool Support

- *Large Language Models (LLMs).* Leverage LLMs to support the automated generation of SymboleoAC specifications, including access control and contractual rules, from natural-language contract documents. This would help improve the usability and effectiveness of a SymboleoAC-based methodology. Promising results are already available for regular Symboleo [113].
- *Runtime functions for SymboleoAC.* Extend the current runtime functions of the smart contract transactions so that they support dynamic rule management (e.g., `addRule()`, `removeRule()`, `updateRule()`, `addPolicy()`, and so on) at runtime, without requiring updating the specifications or regenerating the smart contract (Chapter 9).
- *Model Checking Support.* Extend existing Symboleo model checking tools (i.e., SymboleoPC [90]) to support the SymboleoAC conceptual and language extensions.

Architecture

- *Reputation-Based Security.* Extend the current message broker implementation by implementing a reputation-based enforcement mechanism, where repeated unauthorized publish/subscribe attempts lead to reputation penalties and automatic blacklisting once a threshold is reached, thereby strengthening accountability and non-repudiation in the SymboleoAC architecture (Chapter 5).

- *Robustness and Availability in CPSCs.* In Chapter 3, we highlighted that availability and robustness remain underexplored challenges in Cyber-Physical Smart Contracts. Although SymboleoAC better ensures privacy through secure interactions and access control, it does not currently address high-availability deployment, distributed fault tolerance, or resilience against broker/CEP crashes, network partitions, or sensor outages. This may require extending the current SymboleoAC architecture (Chapter 5) to support redundant message broker instances and clustered CEP engines, among other options.

- *Performance, Throughput, and Scalability in CPSCs.* Investigate the throughput and scalability of the SymboleoAC framework under varying workloads, including increased sensor data rates, larger numbers of concurrent contract instances, and multiple interacting participants. This includes analyzing performance metrics such as transaction latency, event processing time, throughput (events/transactions per second), and resource utilization across system components, as well as exploring scalability strategies such as horizontal scaling of message brokers, distributed deployment of CEP engines, and parallel execution of smart contracts to support large-scale cyber-physical smart contract applications.

- *Usability and Usefulness of CPSCs.* Perform evaluations with with practitioners and domain experts to assess the framework's usability, ease of adoption, and effectiveness in real-world cyber-physical smart contract environments.

Validation

- *Enhanced Specification Validation.* Implement additional IDE validation rules for SymboleoAC specifications (Chapter 8); for example, the performer and controller for all resources should be mandatory.

- *Integration with Real IoT Devices and Smart Containers.* In this thesis, sensors were implemented as virtual devices to simulate IoT data streams (see Chapter 10). These virtual sensors could be replaced by *real* IoT sensors, such as smart containers equipped with embedded temperature, humidity, and shock sensors, enabling the proposed SymboleoAC framework to operate with real sensors in a cyber-physical smart contract environment.

- *Enhanced Tool Validation.* Further improve the validation of the SymboleoAC language and code generation tool (Chapter 8), especially in terms of usability.

IoT Devices and Data Transfers

- *Scalability with Multiple Sensors.* Increase the number of sensors considered in evaluations (Chapter 10), including scenarios with multiple sensors of the same type (e.g., multiple `sealOpen` sensors).

- *Synchronization and Alert Management for Concurrent Sensor Events.* Investigate mechanisms for managing multiple sensor alerts occurring concurrently in order to address synchronization and alert flooding issues between CEP, message broker, and smart contract components. For example, when identical alerts are generated repeatedly by the same sensor, these alerts could be aggregated, ignored, or merged into a single event. When alerts originate from different sensors, strategies such as prioritization, rate limiting, correlation, or alert reduction may be applied instead of ignoring incoming alerts. This raises several open research questions, such as: What should happen when multiple alerts are received simultaneously? How should conflicting or redundant alerts be handled? How can alert handling be optimized without losing critical safety information? Can corresponding CEP rules in EPL be generated automatically?

- *Handling Heterogeneous and Sensitive Sensor Data.* Investigate advanced mechanisms for handling data generated by multiple heterogeneous sensors, particularly in complex scenarios such as the Vaccine Procurement case study, where sensors monitor conditions (e.g., `sealOpen` and `lightExposure`). Future work should consider how the number of deployed sensors and the sensitivity of the transmitted information affect data processing, prioritization, and enforcement decisions. In particular, different sensor types may require differentiated handling strategies based on the criticality of the monitored asset and the potential impact of each sensor reading on contractual compliance.

Note: a snapshot of the code-related artifacts, which reflects their status at the time the final version of this thesis was completed (May 2026), is available online, on Zenodo [5].

References

[1] Areej Abuamra and Issam Al-Azzoni. On the automatic generation of smart contracts with access control for digital health. In *2024 Global Digital Health Knowledge Exchange & Empowerment Conference (gDigiHealth.KEE)*, pages 1–5, 2024. doi:10.1109/gDigiHealth.KEE62309.2024.10761303.

[2] Issam Al-Azzoni and Saqib Iqbal. Model-driven approach for generating smart contracts for access control. In *2023 Fifth International Conference on Blockchain Computing and Applications (BCCA)*, pages 112—115, 2023. doi:10.1109/BCCA58897.2023.10338863.

[3] Hamda Al Breiki, Lamees Al Qassem, Khaled Salah, Muhammad Habib Ur Rehman, and Davor Sevtinovic. Decentralized access control for IoT data using blockchain and trusted oracles. In *2019 IEEE International Conference on Industrial Internet (ICII)*, pages 248–257, 2019. doi:10.1109/ICII.2019.00051.

[4] Sofana Alfuhaid. Towards secure and interactive smart contract code from formal symboleo specifications. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 58–62, 2025. doi:10.1109/ICSE-Companion66252.2025.00024.

[5] Sofana Alfuhaid. Thesis supplement – From formal Symboleo specifications to secure and interactive smart contract code, 2026. Zenodo. doi:10.5281/zenodo.20127371.

[6] Sofana Alfuhaid, Daniel Amyot, Amal Ahmed Anda, and John Mylopoulos. A mapping review on cyber-physical smart contracts: Architectures, platforms, and challenges. *IEEE Access*, 11:65872–65890, 2023. doi:10.1109/ACCESS.2023.3290899.

[7] Sofana Alfuhaid, Amal Ahmed Anda, Daniel Amyot, Marco Roveri, and John Mylopoulos. SymboleoAC: An access control model for legal contracts. In *17th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling (PoEM)*, volume 538 of *LNBIP*, pages 227–243. Springer, 2024. doi:10.1007/978-3-031-77908-4_14.

[8] Sofana Alfuhaid, Amal Ahmed Anda, Daniel Amyot, Marco Roveri, and John Mylopoulos. SymboleoAC: An access control model for smart legal contracts. *Software and Systems Modeling*, 2025. doi:10.1007/s10270-025-01327-9.

[9] Sofana Alfuhaid, Amal Ahmed Anda, Daniel Amyot, Marco Roveri, and John Mylopoulos. Towards an architecture and code generator for end-to-end access control in cyber-physical smart contracts. In *CyPress: 5th Workshop on Software Techniques for Engineering Cyber-Physical Systems*, CASCON '25, pages 687–691. IEEE CS, 2025. doi:10.1109/CASCON66301.2025.11422826.

[10] Ali Alzubaidi, Karan Mitra, and Ellis Solaiman. Smart contract design considerations for sla compliance assessment in the context of iot. In *2021 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pages 74–81, 2021. doi:10.1109/SmartIoT52359.2021.00021.

[11] David Ameller, Xavier Franch, Cristina Gómez, Silverio Martínez-Fernández, João Araújo, Stefan Biffl, Jordi Cabot, Vittorio Cortellessa, Daniel Méndez Fernández, Ana Moreira, Henry Muccini, Antonio Vallecillo, Manuel Wimmer, Vasco Amaral, Wolfgang Böhm, Hugo Bruneliere, Lola Burgueño, Miguel Goulão, Sabine Mavin, and Luca Berardinelli. Dealing with non-functional requirements in model-driven development: A survey. *IEEE Transactions on Software Engineering*, 47:818–835, 04 2021. doi:10.1109/TSE.2019.2904476.

[12] Daniel Amyot, Luigi Logrippo, John Mylopoulos, Marco Roveri, Amal Ahmed Anda, Alireza Parvisimosaed, Sofana Abdullah Alfuhaid, Sepehr Sharifi, Aidin Rasti, Regan Meloche, and Daniel Sousa-Diaz. Engineering smart contracts with Symboleo: A progress report. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*, CASCON '23, pages 235–237, USA, 2023. IBM Corp. URL: https://dl.acm.org/doi/10.5555/3615924.3623631.

[13] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys'18, pages 1–15. ACM, 2018. doi:10.1145/3190508.3190538.

[14] Senthamiz Selvi Arumugam, Venkatesh Umashankar, Nanjangud C Narendra, Ramamurthy Badrinath, Anusha Pradeep Mujumdar, Jan Holler, and Aitor Hernandez. IoT enabled smart logistics using smart contracts. In *2018 8th International Conference on Logistics, Informatics and Service Sciences (LISS)*, pages 1–6, 2018. doi:10.1109/LISS.2018.8593220.

[15] Ada Bagozi, Devis Bianchini, Valeria De Antonellis, Massimiliano Garda, and Michele Melchiori. Exploiting blockchain and smart contracts for data exploration as a service. In *Proceedings of the 21st International Conference on Information Integration and Web-Based Applications & Services*, iiWAS2019, pages 393–402. ACM, 2019. doi:10.1145/3366030.3366075.

[16] Gavina Baralla, Andrea Pinna, Roberto Tonelli, Michele Marchesi, and Simona Ibba. Ensuring transparency and traceability of food local products: A blockchain application to a smart tourism region. *Concurrency and Computation: Practice and Experience*, 33(1):e5857, 2021. doi:10.1002/cpe.5857.

[17] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Financial Cryptography and Data Security*, pages 494–509. Springer, 2017. doi:10.1007/978-3-319-70278-0_31.

[18] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, jan 2006. doi:10.1145/1125808.1125810.

[19] Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. GemRBAC-DSL: A high-level specification language for role-based access control policies. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, SACMAT '16, pages 179–190. ACM, 2016. doi:10.1145/2914642.2914656.

[20] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016.

[21] Javier Blanco-Romero, Florina Almenares Mendoza, Carlos García Rubio, Celeste Campo, and Daniel Díaz Sánchez. On the practical feasibility of harvest-now, decrypt-later attacks. *arXiv preprint arXiv:2603.01091*, 2026. doi:10.48550/arXiv.2603.01091.

[22] Jayant D Bokefode, Swapnaja A Ubale, Sulabha S Apte, and Dattatray G Modani. Analysis of DAC MAC RBAC access control based models for security. *International Journal of Computer Applications*, 104(5):6–13, October 2014. doi:10.5120/18196-9115.

[23] Juan Boubeta-Puig, Guadalupe Ortiz, and Inmaculada Medina-Bulo. A model-driven approach for facilitating user-friendly design of complex event patterns. *Expert Systems with Applications*, 41(2):445–456, 2014. doi:10.1016/j.eswa.2013.07.070.

[24] Juan Boubeta-Puig, Jesús Rosa-Bilbao, and Jan Mendling. CEPchain: A graphical model-driven solution for integrating complex event processing and blockchain. *Expert Systems with Applications*, 184:115578, 2021. doi:10.1016/j.eswa.2021.115578.

[25] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform, 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[26] Arun C. R., Ashis K. Pani, and Prashant Kumar. Blockchain-enabled smart contracts and the Internet of Things: Advancing the research agenda through a narrative review. *Multimedia Tools and Applications*, 84(8):5097–5147, 2025. doi:10.1007/s11042-024-18931-4.

[27] Diletta Cacciagrano, Flavio Corradini, Gianmarco Mazzante, Leonardo Mostarda, and Davide Sestili. Off-chain execution of IoT smart contracts. In *Advanced Information Networking and Applications*, pages 608–619, Cham, 2021. Springer. doi:10.1007/978-3-030-75075-6_50.

[28] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the Internet of Things. *IEEE Access*, 4:2292–2303, 2016. doi:10.1109/ACCESS.2016.2566339.

[29] Covidence. Covidence. Better systematic review management, 2021. URL: https://www.covidence.org/.

[30] Debashis Das, Sourav Banerjee, Pushpita Chatterjee, Uttam Ghosh, Utpal Biswas, and Wathiq Mansoor. Security, trust, and privacy management framework in cyber-physical systems using blockchain. In *2023 IEEE 20th Consumer Communications & Networking Conference (CCNC)*, pages 1–6, 2023. doi:10.1109/CCNC51644.2023.10060483.

[31] Michael Dawson. Smart contract compliance monitoring and enforcement, 2020. US Patent App. 16/176,843. URL: https://patents.google.com/patent/US20200134711A1/en.

[32] Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. *Software: Practice and Experience*, 33(5):397–421, 2003. doi:10.1002/spe.513.

[33] B.D. Deebak and Fadi AL-Turjman. A robust and distributed architecture for 5g-enabled networks in the smart blockchain era. *Computer Communications*, 181:293–308, 2022. doi:10.1016/j.comcom.2021.10.015.

[34] Christian Dienbauer, Benedikt Pittl, Werner Mach, and Erich Schikuta. A penalty-aware cloud monitoring system based on blockchains. In *Proceedings of the 22nd International Conference on Information Integration and Web-Based Applications & Services*, iiWAS'20, pages 154–162. ACM, 2020. doi:10.1145/3428757.3429130.

[35] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 227–238, 2017. doi:10.1145/3093742.3093908.

[36] Mohammed Ennahbaoui and Said Elhajji. Study of access control models, 2013. URL: https://www.iaeng.org/publication/WCE2013/WCE2013_pp1215-1220.pdf.

[37] Ghareeb Falazi, Andrea Lamparelli, Uwe Breitenbuecher, Florian Daniel, and Frank Leymann. Unified integration of smart contracts through service orientation. *IEEE Software*, 37(5):60–66, 2020. doi:10.1109/MS.2020.2994040.

[38] Andrew Forward, Omar Badreddin, Timothy C. Lethbridge, and Julian Solano. Model-driven rapid prototyping with Umple. *Software: Practice and Experience*, 42(7):781–797, 2012. doi:10.1002/spe.1155.

[39] Christopher K. Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 210–215, 2016. doi:10.1109/FAS-W.2016.53.

[40] Guido Governatori, Florian Idelberger, Zoran Milosevic, Regis Riveret, Giovanni Sartor, and Xiwei Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law*, 26:377–409, 12 2018. doi:10.1007/s10506-018-9223-3.

[41] Cristine Griffo, João Paulo A. Almeida, and Giancarlo Guizzardi. Conceptual modeling of legal relations. In Juan C. Trujillo, Karen C. Davis, Xiaoyong Du, Zhanhuai Li, Tok Wang Ling, Guoliang Li, and Mong Li Lee, editors, *Conceptual Modeling*, pages 169–183, Cham, 2018. Springer. doi:10.1007/978-3-030-00847-5_14.

[42] Cristine Griffo, João Paulo A. Almeida, Giancarlo Guizzardi, and Julio Cesar Nardi. Service contract modeling in enterprise architecture: An ontology-based approach. *Information Systems*, 101:101454, 2021. doi:10.1016/j.is.2019.101454.

[43] Vinay Gugueoth, Sunitha Safavat, Sachin Shetty, and Danda Rawat. A review of IoT security and privacy using decentralized blockchain techniques. *Computer Science Review*, 50:100585, 2023. doi:10.1016/j.cosrev.2023.100585.

[44] Kemal Guler, Dirk Beyer, and Cipriano Santos. Computer-implemented method for automatic contract monitoring, 2005. US Patent App. 10/443,930. URL: https://patents.google.com/patent/US20050015319A1/en.

[45] Mohammad Hamdaqa, Lucas Alberto Pineda Met, and Ilham Qasse. iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Information and Software Technology*, 144:106762, 2022. doi:10.1016/j.infsof.2021.106762.

[46] Lei Hang and DoHyeun Kim. Design and implementation of an integrated IoT blockchain platform for sensing data integrity. *Sensors*, 19(10), 2019. doi:10.3390/s19102228.

[47] Lei Hang and Do-Hyeun Kim. Reliable task management based on a smart contract for runtime verification of sensing and actuating tasks in IoT environments. *Sensors*, 20(4), 2020. doi:10.3390/s20041207.

[48] Lei Hang, Israr Ullah, and Do-Hyeun Kim. A secure fish farm platform based on blockchain for agriculture data integrity. *Computers and Electronics in Agriculture*, 170:105251, 2020. doi:10.1016/j.compag.2020.105251.

[49] Haya Hasan, Esra AlHadhrami, Alia AlDhaheri, Khaled Salah, and Raja Jayaraman. Smart contract-based approach for efficient shipment management. *Computers & Industrial Engineering*, 136:149–159, 2019. doi:10.1016/j.cie.2019.07.022.

[50] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004. URL: http://www.jstor.org/stable/25148625, doi:10.2307/25148625.

[51] Tharaka Hewa, An Braeken, Madhusanka Liyanage, and Mika Ylianttila. Fog computing and blockchain-based security service architecture for 5G industrial IoT-enabled cloud manufacturing. *IEEE Transactions on Industrial Informatics*, 18(10):7174–7185, 2022. doi:10.1109/TII.2022.3140792.

[52] Olivér Hornyák and George Farid Alkhoury. Smart Contracts in the Automotive Industry. In *Vehicle and Automotive Engineering*, pages 148–157. Springer, 2020. doi:10.1007/978-981-15-9529-5.

[53] Vincent C. Hu, D. Richard Kuhn, David F. Ferraiolo, and Jeffrey Voas. Attribute-based access control. *Computer*, 48(2):85–88, 2015. doi:10.1109/MC.2015.33.

[54] Marin Ivezic. What is trust now, forge later (TNFL)?, May 2026. *PostQuantum.com*. https://postquantum.com/quantum-security-reference/what-is-trust-now-forge-later/.

[55] Faisal Jamil, Shabir Ahmad, Naeem Iqbal, and Do-Hyeun Kim. Towards a remote monitoring of patient vital signs based on IoT-based blockchain integrity management platforms in smart hospitals. *Sensors*, 20(8), 2020. doi:10.3390/s20082195.

[56] Faisal Jamil, Muhammad Ibrahim, Israr Ullah, Suyeon Kim, Hyun Kook Kahng, and Do-Hyeun Kim. Optimal smart contract for autonomous greenhouse environment based on IoT blockchain network in agriculture. *Computers and Electronics in Agriculture*, 192:106573, 2022. doi:10.1016/j.compag.2021.106573.

[57] Faisal Jamil, Naeem Iqbal, Imran, Shabir Ahmad, and Dohyeun Kim. Peer-to-peer energy trading mechanism based on blockchain and machine learning for sustainable electrical power supply in smart grid. *IEEE Access*, 9:39193–39217, 2021. doi:10.1109/ACCESS.2021.3060457.

[58] Audun Jøsang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Computers & Security*, 24(7):618–634, 2005. doi:10.1016/j.cose.2005.07.001.

[59] Nadine Kashmar, Mehdi Adda, Mirna Atieh, and Hussein Ibrahim. Access control metamodel for policy specification and enforcement: From conception to formalization. *Procedia Computer Science*, 184:887–892, 2021. The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) Affiliated Workshops. doi:10.1016/j.procs.2021.03.111.

[60] Anne V. D. M. Kayem, Selim G. Akl, and Patrick Martin. *A Presentation of Access Control Methods*, pages 11–40. Springer US, Boston, MA, 2010. doi:10.1007/978-1-4419-6655-1_2.

[61] Axel Kern and Claudia Walhorn. Rule support for role-based access control. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, SACMAT '05, pages 130–138. ACM, 2005. doi:10.1145/1063979.1064002.

[62] Dae-Kyoo Kim, Indrakshi Ray, Robert France, and Na Li. Modeling role-based access control using parameterized UML models. In Michel Wermelinger and Tiziana Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering*, pages 180–193. Springer, 2004. doi:10.1007/978-3-540-24721-0_13.

[63] Petar Kochovski, Vlado Stankovski, Sandi Gec, Francescomaria Faticanti, Marco Savi, Domenico Siracusa, and Seungwoo Kum. Smart contracts for service-level agreements in edge-to-cloud computing. *Journal of Grid Computing*, pages 673–690, 12 2020. doi:10.1007/s10723-020-09534-y.

[64] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858, 2016. doi:10.1109/SP.2016.55.

[65] Mahtab Kouhizadeh, Sara Saberi, and Joseph Sarkis. Blockchain technology and the sustainable supply chain: Theoretically exploring adoption barriers. *International Journal of Production Economics*, 231:107831, 2021. doi:10.1016/j.ijpe.2020.107831.

[66] Mirco Kuhlmann, Karsten Sohr, and Martin Gogolla. Employing UML and OCL for designing and analysing role-based access control. *Mathematical Structures in Computer Science*, 23(4):796–833, 2013. doi:10.1017/S0960129512000266.

[67] Prabhat Kumar, Randhir Kumar, Govind P. Gupta, Rakesh Tripathi, Alireza Jolfaei, and A.K.M. Najmul Islam. A blockchain-orchestrated deep learning approach for secure data transmission in IoT-enabled healthcare system. *Journal of Parallel and Distributed Computing*, 172:69–83, 2023. doi:10.1016/j.jpdc.2022.10.002.

[68] Randhir Kumar, Prabhat Kumar, Ahamed Aljuhani, A. K. M. Najmul Islam, Alireza Jolfaei, and Sahil Garg. Deep learning and smart contract-assisted secure data sharing for iot-based intelligent agriculture. *IEEE Intelligent Systems*, pages 1–8, 2022. doi:10.1109/MIS.2022.3201553.

[69] Randhir Kumar, Prabhat Kumar, Moayad Aloqaily, and Ahamed Aljuhani. Deep-learning-based blockchain for secure zero touch networks. *IEEE Communications Magazine*, 61(2):96–102, 2023. doi:10.1109/MCOM.001.2200294.

[70] Randhir Kumar, Prabhat Kumar, Rakesh Tripathi, Govind P. Gupta, Neeraj Kumar, and Mohammad Mehedi Hassan. A privacy-preserving-based secure framework using

blockchain-enabled deep-learning in cooperative intelligent transport system. *IEEE Transactions on Intelligent Transportation Systems*, 23(9):16492–16503, 2022. doi:10.1109/TITS.2021.3098636.

[71] Jan Ladleif, Ingo Weber, and Mathias Weske. External data monitoring using oracles in blockchain-based process execution. In *Business Process Management: Blockchain and Robotic Process Automation Forum*, pages 67–81, Cham, 2020. Springer. doi:10.1007/978-3-030-58779-6_5.

[72] Christoph Lehnert, Grischan Engel, and Thomas Greiner. Distributed ledger and smart contract based approach for IoT sensor applications. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6, 2020. doi:10.1109/COINS49042.2020.9191409.

[73] Elva Leka, Besnik Selimi, and Luis Lamani. Systematic literature review of blockchain applications: Smart contracts. In *2019 International Conference on Information Technologies (InfoTech)*, pages 1–3, 2019. doi:10.1109/InfoTech.2019.8860872.

[74] Timothy C. Lethbridge, Andrew Forward, Omar Badreddin, Dusan Brestovansky, Miguel Garzon, Hamoud Aljamaan, Sultan Eid, Ahmed Husseini Orabi, Mahmoud Husseini Orabi, Vahdat Abdelzad, Opeyemi Adesina, Aliaa Alghamdi, Abdulaziz Algablan, and Amid Zakariapour. Umple: Model-driven development for open source and education. *Science of Computer Programming*, 208:102665, 2021. doi:10.1016/j.scico.2021.102665.

[75] Xiaolong Liu, Khan Muhammad, Jaime Lloret, Yu-Wen Chen, and Shyan-Ming Yuan. Elastic and cost-effective data carrier architecture for smart contract in blockchain. *Future Generation Computer Systems*, 100:590–599, 2019. doi:10.1016/j.future.2019.05.042.

[76] Jannik Lockl, Vincent Schlatt, André Schweizer, Nils Urbach, and Natascha Harth. Toward trust in Internet of Things (IoT) ecosystems: Design principles for blockchain-based IoT applications. *IEEE Transactions on Engineering Management*, 67(4):1256–1270, 2020. doi:10.1109/TEM.2020.2978014.

[77] Orlenys Lopez-Pintado, Marlon Dumas, Luciano Garcia-Banuelos, and Ingo Weber. Interpreted Execution of Business Process Models on Blockchain. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 206–215. IEEE, 2019. doi:10.1109/EDOC.2019.00033.

[78] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison Wesley Longman, 2002. URL: https://books.google.ca/books?id=l43engEACAAJ.

[79] Jillian Mascelli and Megan Rodden. "Harvest Now Decrypt Later": Examining post-quantum cryptography and the data privacy risks for distributed ledger networks. *Finance and Economics Discussion Series*, 2025. doi:10.17016/FEDS.2025.093.

[80] Luke McGuinness and Neal Haddaway. Prisma2020: R package and ShinyApp for producing PRISMA 2020 compliant flow diagrams, 2020. doi:10.5281/zenodo.4287835.

[81] Regan Meloche, Durga Sivakumar, Amal Ahmed Anda, Sofana Alfuhaid, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. A web-based environment for the specification and generation of smart legal contracts. In *Compliance for Artificial Intelligence Systems: Strategies, Principles and Methods*, volume 14377 of *LNAI*, pages 1–13. Springer, 2026. doi:10.1007/978-3-032-12795-2_2.

[82] David Moher, Alessandro Liberati, Jennifer Tetzlaff, and Douglas G Altman. Preferred reporting items for systematic reviews and meta-analyses: the PRISMA statement. *BMJ*, 339, 2009. doi:10.1136/bmj.b2535.

[83] Carlos Molina-Jiménez, Ellis Solaiman, Ioannis Sfyrakis, Irene Ng, and Jon Crowcroft. On and off-blockchain enforcement of smart contracts. In *Euro-Par 2018: Parallel Processing Workshops*, pages 342–354. Springer, 2019. doi:10.1007/978-3-030-10549-5_27.

[84] Erica Mourão, João Felipe Pimentel, Leonardo Murta, Marcos Kalinowski, Emilia Mendes, and Claes Wohlin. On the performance of hybrid search strategies for systematic literature reviews in software engineering. *Information and Software Technology*, 123:106294, 2020. doi:10.1016/j.infsof.2020.106294.

[85] John Mylopoulos, Sofana Alfuhaid, Daniel Amyot, Amal Ahmed Anda, Luigi Logrippo, Regan Meloche, Ashkan Rahimi-Kian, Sahil Rajpal, Marco Roveri, Durga Sivakumar, and Daniel Sousa-Dias. Engineering smart contracts with SYMBOLEO: Progress report 2024. In *2024 34th International Conference on Collaborative Advances in Software and COmputiNg (CASCON)*, pages 1–5. IEEE CS, 2024. doi:10.1109/CASCON62161.2024.10838220.

[86] Nils Neidhardt, Carsten Köhler, and Markus Nüttgens. Cloud service billing and service level agreement monitoring based on blockchain. In *Proceedings of the 9th International Workshop on Enterprise Modeling and Information Systems Architectures (EMISA 2018)*, volume 2097, pages 65–69. CEUR-WS, 2018. URL: http://ceur-ws.org/Vol-2097/paper11.pdf.

[87] Sina Rafati Niya, Sanjiv S. Jha, Thomas Bocek, and Burkhard Stiller. Design and implementation of an automated and decentralized pollution monitoring system with blockchains, smart contracts, and LoRaWAN. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–4, 2018. doi:10.1109/NOMS.2018.8406329.

[88] Chitu Okoli. A guide to conducting a standalone systematic literature review. *Communications of the Association for Information Systems*, 37, 2015. doi:10.17705/1CAIS.03743.

[89] Alireza Parvizimosaed. *Symboleo: Specification and Verification of Legal Contracts*. PhD thesis, University of Ottawa, Canada, October 2022. doi:10.20381/ruor-28399.

[90] Alireza Parvizimosaed, Marco Roveri, Aidin Rasti, Amal Ahmed Anda, Sofana Alfuhaid, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. SymboleoPC: checking properties of legal contracts. *Software and Systems Modeling*, 24:1093–1126, 2025. doi:10.1007/s10270-024-01180-2.

[91] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Subcontracting, assignment, and substitution for legal contracts in Symboleo. In *Conceptual Modeling: 39th International Conference, ER 2020*, pages 271–285, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-62522-1_20.

[92] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, Marco Roveri, Aidin Rasti, Ali Roudak, and John Mylopoulos. Specification and analysis of legal contracts with Symboleo. *Software and Systems Modeling*, 21(6):2395–2427, 2022. doi:10.1007/s10270-022-01053-6.

[93] Cherilyn Pascoe, Stephen Quinn, and Karen Scarfone. The NIST cybersecurity framework (CSF) 2.0, 2024. NIST Cybersecurity White Papers (CSWP), National Institute of Standards and Technology, Gaithersburg, MD, USA. doi:10.6028/NIST.CSWP.29.

[94] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007. doi:10.2753/MIS0742-1222240302.

[95] Elena Planas, Salvador Pérez, Marco Brambilla, and Jordi Cabot. Modeling and enforcing access control policies in conversational user interfaces. *Software and Systems Modeling*, 22:1–20, 11 2023. doi:10.1007/s10270-023-01131-3.

[96] Matevž Pustišek, Min Chen, Andrej Kos, and Anton Kos. Decentralized machine autonomy for manufacturing servitization. *Sensors*, 22(1), 2022. doi:10.3390/s22010338.

[97] Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson. Immutable in principle, upgradeable by design: exploratory study of smart contract upgradeability. *Empirical Software Engineering*, 31(2):39, December 2025. doi:10.1007/s10664-025-10779-y.

[98] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: The next computing revolution. In *Design Automation Conference*, pages 731–736, 2010. doi:10.1145/1837274.1837461.

[99] Aidin Rasti. From Symboleo to smart contracts : A code generator. Master's thesis, University of Ottawa, Canada, 2022. doi:10.20381/ruor-28394.

[100] Aidin Rasti, Daniel Amyot, Alireza Parvizimosaed, Marco Roveri, Luigi Logrippo, Amal Ahmed Anda, and John Mylopoulos. Symboleo2SC: From legal contract specifications to smart contracts. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, pages 300–310. ACM, 2022. doi:10.1145/3550355.3552407.

[101] Aidin Rasti, Amal Ahmed Anda, Sofana Alfuhaid, Alireza Parvizimosaed, Daniel Amyot, Marco Roveri, Luigi Logrippo, and John Mylopoulos. Automated generation of smart contract code from legal contract specifications with Symboleo2SC. *Software and Systems Modeling*, 24:1127–1156, 2025. doi:10.1007/s10270-024-01187-9.

[102] Abderahman Rejeb, John G. Keogh, and Horst Treiblmaier. Leveraging the Internet of Things and blockchain technology in supply chain management. *Future Internet*, 11(7), 2019. doi:10.3390/fi11070161.

[103] Dan Rice. Method and system for monitoring a smart contract on a distributed ledger, 2019. US Patent App. 16/019,203. URL: https://patents.google.com/patent/US20190392178A1/en.

[104] Jesús Rosa-Bilbao, Juan Boubeta-Puig, and Adrian Rutle. CEPEDALoCo: An event-driven architecture for integrating complex event processing and blockchain through low-code. *Internet of Things*, 22:100802, 2023. doi:10.1016/j.iot.2023.100802.

[105] Ron Ross, Richard Graubart, Deborah Bodeau, and Richard McQuaid. Systems security engineering: Considerations for a multidisciplinary approach in the engineering of trustworthy secure systems. Technical Report Special Publication 800-160, Vol. 1, National Institute of Standards and Technology (NIST), 2016. doi:10.6028/NIST.SP.800-160v1.

[106] Thiago R P M Rúbio, Zafeiris Kokkinogenis, Henrique Lopes Cardoso, Rosaldo J F Rossetti, and Eugénio Oliveira. Regulating Blockchain Smart Contracts with Agent-Based Markets. In *EPIA Conference on Artificial Intelligence*, pages 399–411. Springer, 2019. doi:10.1007/978-3-030-30241-2_34.

[107] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control*, RBAC '00, pages 47–63. ACM, 2000. doi:10.1145/344287.344301.

[108] R.S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994. doi:10.1109/35.312842.

[109] Sepehr Sharifi, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Symboleo: Towards a specification language for legal contracts. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 364–369, 2020. doi:10.1109/RE48521.2020.00049.

[110] Seyed Sepehr Sharifi. Smart contracts: From formal specification to blockchain code. Master's thesis, University of Ottawa, Canada, 2020. doi:10.20381/ruor-25092.

[111] Anshu Shukla, Swarup Kumar Mohalik, and Ramamurthy Badrinath. Smart contracts for multiagent plan execution in untrusted cyber-physical systems. In *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, pages 86–94, 2018. doi:10.1109/HiPCW.2018.8634034.

[112] Manan Shukla, Jianjing Lin, and Oshani Seneviratne. BlockIoT-RETEL: Blockchain and IoT based read-execute-transact-erase-loop environment for integrating personal health data. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 237–243, 2021. doi:10.1109/Blockchain53845.2021.00039.

[113] Gurdarshan Singh, Sahil Rajpal, Amal Ahmed Anda, Sofana Alfuhaid, Daniel Amyot, Marco Roveri, and John Mylopoulos. Towards an LLM-based auto-corrector agent for SYMBOLEO specifications. In *CyPress: 5th Workshop on Software Techniques for Engineering Cyber-Physical Systems*, CASCON '25, pages 672–676. IEEE CS, 2025. doi:10.1109/CASCON66301.2025.00124.

[114] Alexander Smirnov and Nikolay Teslya. Robot interaction through smart contract for blockchain-based coalition formation. In *Product Lifecycle Management to Support Industry 4.0*, pages 611–620, Cham, 2018. Springer. doi:10.1007/978-3-030-01614-2_56.

[115] Ellis Solaiman, Todd Wike, and Ioannis Sfyrakis. Implementation and evaluation of smart contracts using a hybrid on- and off-blockchain architecture. *Concurrency and Computation: Practice and Experience*, 33(1):e5811, 2020. doi:10.1002/cpe.5811.

[116] Daniel Sousa-Dias, Daniel Amyot, Ashkan Rahimi-Kian, and John Mylopoulos. A review of cybersecurity concerns for transactive energy markets. *Energies*, 16(13), 2023. doi:10.3390/en16134838.

[117] Nick Szabo. The idea of smart contracts, 1997. URL: https://bit.ly/2PXmUfz.

[118] Sharvari T and Sowmya Nag K. A study on modern messaging systems- kafka, rabbitmq and nats streaming, 2019. URL: https://arxiv.org/abs/1912.03715, arXiv:1912.03715.

[119] Mona Taghavi, Jamal Bentahar, Hadi Otrok, and Kaveh Bakhtiyari. A blockchain-based model for cloud service quality monitoring. *IEEE Transactions on Services Computing*, 13(2):276–288, 2020. doi:10.1109/TSC.2019.2948010.

[120] Shirin Tahmasebi, Jafar Habibi, and Abolhassan Shamsaie. A scalable architecture for monitoring IoT devices using ethereum and fog computing. In *2020 4th International Conference on Smart City, Internet of Things and Applications (SCIOT)*, pages 66–76, 2020. doi:10.1109/SCIOT50840.2020.9250193.

[121] Rafael Brundo Uriarte, Huan Zhou, Kyriakos Kritikos, Zeshun Shi, Zhiming Zhao, and Rocco De Nicola. Distributed service-level agreement management with smart contracts and blockchain. *Concurrency and Computation: Practice and Experience*, 33(14):e5800, 2020. doi:10.1002/cpe.5800.

[122] Shubham Vyas, Rajesh Kumar Tyagi, Charu Jain, and Shashank Sahu. Literature review : A comparative study of real time streaming technologies and apache kafka. In *2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT)*, pages 146–153, 2021. doi:10.1109/CCICT53244.2021.00038.

[123] Shuai Wang, Liwei Ouyang, Yong Yuan, Xiaochun Ni, Xuan Han, and Fei-Yue Wang. Blockchain-enabled smart contracts: Architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(11):2266–2277, 2019. doi:10.1109/TSMC.2019.2895123.

[124] Roel J. Wieringa. *Design science methodology for information systems and software engineering*. Springer Berlin, Heidelberg, 2014. doi:10.1007/978-3-662-43839-8.

[125] Kwame-Lante Wright, Martin Martinez, Uday Chadha, and Bhaskar Krishnamachari. Smartedge: A smart contract for edge computing. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1685–1690, 2018. doi:10.1109/Cybermatics_2018.2018.00281.

[126] Vinden Wylde, Nisha Rawindaran, John Lawrence, Rushil Balasubramanian, Edmond Prakash, Ambikesh Jayal, Imtiaz Khan, Chaminda Hewage, and Jon Platts. Cybersecurity, data privacy and blockchain: A review. *SN computer science*, 3(2):127, 2022. doi:10.1007/s42979-022-01020-4.

[127] Maximilian Wöhrer and Uwe Zdun. Domain specific language for smart contract development. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2020. doi:10.1109/ICBC48266.2020.9169399.

[128] Can Zhang, Liehuang Zhu, and Chang Xu. BSDP: Blockchain-based smart parking for digital-twin empowered vehicular sensing networks with privacy protection. *IEEE Transactions on Industrial Informatics*, pages 1–10, 2022. doi:10.1109/TII.2022.3223211.

[129] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, pages 270–282. ACM, 2016. doi:10.1145/2976749.2978326.

[130] Huan Zhou, Cees de Laat, and Zhiming Zhao. Trustworthy cloud service level agreement enforcement with blockchain based smart contract. In *2018 IEEE International*

Conference on Cloud Computing Technology and Science (CloudCom), pages 255–260, 2018. doi:10.1109/CloudCom2018.2018.00057.

Appendix A

SymboleoAC2SC Code Generator

This appendix presents the details of the remaining code generation components introduced in Section 9.4.

Conditional LegalSituation – Antecedent and Consequent

As a continuation of Section 9.4.9 on unconditional legal situations, the antecedent and consequent of a conditional legal position are executed only when the specified legal expression evaluates to *true*. These elements are created in the listener rather than at contract compilation time. Just as we add their rules when these positions are instantiated, we similarly add their antecedent and consequent components only after the corresponding legal position has been created.

Listing A.1 introduces the `generateEventVariableCondition()` function which represents two types of information parts, i.e., `eventCondition` or `stateCondition` depending on whether the event refers to a variable, power, or obligation. Building on this, Listing A.2 shows the `generatePredicateFunctionCondition()` function, which maps predicate expressions such as `happens`, `happensBefore`, or `happensAfter` to the corresponding informational structures generated by `generateEventVariableCondition()`. Listing A.3 shows the `generateLegalpositionCondition()` function, which represents the third part of informational parts `Condition`.

Listing A.4 introduces the `compilePowerCondition()` function, which generates the appropriate `stateCondition` structure for power consequents. This includes handling obligation related states (e.g., *suspension*, *discharge*, *unsuccessful termination*) as well as contract level states.

Finally, Listing A.5 presents the improved `compileEventsFile()` function, where these generated conditions are incorporated into the event listeners responsible for creating new `LegalSituation` instances.

```
def String generateEventVariableCondition(Event event) {
    switch (event) {

        VariableEvent:
            return "{_type: 'eventCondition', resource: \""
                + generateDotExpressionString(event.variable, '')
                + "\", resourceType: \""
                + generateDotExpressionType(event.variable)
                + "\"}"
```

```
        PowerEvent:
            return '''{_type: 'stateCondition', resourceType: "power",
                resource: "<<event.powerVariable.name>>",
                state: "<<event.eventName>>"}'''

        ObligationEvent:
            return '''{_type: 'stateCondition',
                resourceType: "<< isSurvivingObligation(event.obligationVariable.name) ? "survivingObligation" : "
     obligation" >>",
                resource: "<<event.obligationVariable.name>>",
                state: "<<geteventName(event.eventName)>>"}'''
    }
}
```

Listing A.1: New Xtend function `generateEventVariableCondition()`.

```
def String generatePredicateFunctionCondition(PredicateFunction predicate) {

    switch (predicate) {

      PredicateFunctionHappens: return '''«generateEventVariableCondition(predicate.event)» '''

      PredicateFunctionHappensAfter: return '''«generateEventVariableCondition(predicate.event)»'''

      PredicateFunctionWHappensBefore: return '''«generateEventVariableCondition(predicate.event)» '''

      PredicateFunctionSHappensBefore: return '''«generateEventVariableCondition(predicate.event)»'''

      PredicateFunctionHappensWithin: return '''«generateEventVariableCondition(predicate.event)» '''

      PredicateFunctionAssignment: return '''«generateEventVariableCondition(predicate.event)» '''

      PredicateFunctionAssignmentOnly: return '''true'''

    }

  }
```

Listing A.2: New Xtend function `generatePredicateFunctionCondition()`.

```
// Returns different types of conditions of antecedents and consequents
def String generateLegalpositionCondition(Proposition proposition, String addAC) {
    switch (proposition) {

        PEquality:
            return addAC
                + "{ leftSide:'"
                + generateLegalpositionCondition(proposition.left, addAC)
                + "', op:'"
                + getEqualityOperator(proposition.op)
                + "', rightSide:'"
                + generateLegalpositionCondition(proposition.right, addAC)
                + "', _type:'Condition'} )\n"

        PComparison:
            return addAC
                + "{ leftSide:'"
                + generateLegalpositionCondition(proposition.left, addAC)
                + "', op:'"
                + proposition.op
                + "', rightSide:'"
                + generateLegalpositionCondition(proposition.right, addAC)
                + "', _type:'Condition'} )\n"

        PArithmetic:
            return addAC
                + "{ leftSide:'"
                + generateLegalpositionCondition(proposition.left, addAC)
                + "', op:'"
                + proposition.op
                + "', rightSide:'"
                + generateLegalpositionCondition(proposition.right, addAC)
                + "', _type:'Condition'} )\n"

        // ... omitted cases (POr, PAnd, PAtomRecursive, etc.)
    }
}
```

Listing A.3: New Xtend function `generateLegalpositionCondition()`.

```
def String compilePowerCondition(PowerFunction powerFunction) {
// conditions in power consequent
    var con = ""

    switch (powerFunction) {
        PFObligationSuspended:
            con = "resourceType: 'obligation', resource: '" + powerFunction.norm.name + "', state:'suspension'"
        PFObligationResumed:
            con = "resourceType: 'obligation', resource: '" + powerFunction.norm.name + "', state:'suspension'"
        PFObligationDischarged:
            con = "resourceType: 'obligation', resource: '" + powerFunction.norm.name + "', state:'discharge'"
        PFObligationTerminated:
            con = "resourceType: 'obligation', resource: '" + powerFunction.norm.name + "', state:'
     unsuccessfultermination'"
        PFObligationTriggered:
            con = "resourceType: 'obligation', resource: '" + powerFunction.norm.name + "', state:'create'"
        PFContractSuspended:
            con = "resourceType: 'contract', resource: 'contract', state:'suspension'"
        PFContractResumed:
            con = "resourceType: 'contract', resource: 'contract', state:'suspension'"
        PFContractTerminated:
            con = "resourceType: 'contract', resource: 'contract', state:'unsuccessfultermination'"
    }

    return con
}
```

Listing A.4: New Xtend function `compilePowerCondition()` for generating stateCondition structures for power consequents.

```
def void compileEventsFile(IFileSystemAccess2 fsa, Model model) {
//... omitted code

const EventListeners = {
        «FOR obligation : obligationTriggerEvents.keySet»
          createObligation_«obligation.name»(contract) {
            if («generatePropositionString(obligation.trigger)») { «"\n"+generatePropositionAssignString(obligation.
     trigger)»
              if (contract.obligations.«obligation.name» == null || contract.obligations.«obligation.name».isFinished())
      {
                const isNewInstance =  contract.obligations.«obligation.name» != null && contract.obligations.«
     obligation.name».isFinished()
                contract.«obligation.name»Situation = new LegalSituation();
              «IF !(obligation.consequent instanceof PAtomPredicateTrueLiteral)»
                «generateLegalpositionCondition(obligation.consequent,"contract."+obligation.name+"Situation.
     addConsequentOf(")»
                «ENDIF»
                «IF !(obligation.antecedent instanceof PAtomPredicateTrueLiteral)»
                 «generateLegalpositionCondition(obligation.antecedent,"contract."+obligation.name+"Situation.
     addAntecedentOf(")»
                «ENDIF»
                 contract.obligations.«obligation.name» = new Obligation('«obligation.name»', «
     generateDotExpressionString(obligation.creditor, 'contract')», «generateDotExpressionString(obligation.debtor, '
     contract')», contract, contract.«obligation.name»Situation)
                 «getSpecifiedControllerObligation(obligation)»
                 «getSpecifiedRulesCondObligation(obligation, model)»
                if («obligation.antecedent instanceof PAtomPredicateTrueLiteral ? "true" : "!isNewInstance "» ) { «"\n"+
     generatePropositionAssignString(obligation.antecedent)»
                  contract.obligations.«obligation.name».trigerredUnconditional()
                  if (!isNewInstance && «generatePropositionString(obligation.consequent)») { «"\n"+
     generatePropositionAssignString(obligation.consequent)»
                    contract.obligations.«obligation.name».fulfilled()
                  }
                } else {
                  contract.obligations.«obligation.name».trigerredConditional()
                }
              }
            }
          },
        «ENDFOR»
// ... omitted code
 «FOR power : powerTriggerEvents.keySet»
          createPower_«power.name»(contract) {
            const effects = { powerCreated: false }
            if («generatePropositionString(power.trigger)») { «"\n"+generatePropositionAssignString(power.trigger)»
              if (contract.powers.«power.name» == null || contract.powers.«power.name».isFinished()){
                const isNewInstance =  contract.powers.«power.name» != null && contract.powers.«power.name».isFinished()
                contract.«power.name»Situation = new LegalSituation();
                          «IF !(power.antecedent instanceof PAtomPredicateTrueLiteral)»
                «generateLegalpositionCondition(power.antecedent,"contract."+power.name+"Situation.addAntecedentOf(")»
                          «ENDIF»
                             this.«power.name»Situation.addConsequentOf({_type: 'stateCondition',«compilePowerCondition
     (power.consequent)»})
                contract.powers.«power.name» = new Power('«power.name»', «generateDotExpressionString(power.creditor, '
     contract')», «generateDotExpressionString(power.debtor, 'contract')», contract, contract.«power.name»Situation)
```

```
                effects.powerCreated = true
                effects.powerName = '«power.name»'
                «getSpecifiedControllerPower(power)»
                «getSpecifiedRulesCondPower(power, model)»
                if («power.antecedent instanceof PAtomPredicateTrueLiteral ? "true" : "!isNewInstance && "+
    generatePropositionString(power.antecedent) » ) { «"\n"+generatePropositionAssignString(power.antecedent)»
                  contract.powers.«power.name».trigerredUnconditional()
                } else {
                  contract.powers.«power.name».trigerredConditional()
                }
              }
            }
            return effects
          },
        «ENDFOR»
//... omitted code
}
```

Listing A.5: Xtend source code of part of the improved `compileEventsFile()` method.

Appendix B

Generation of Smart Contract Transactions – Run-Time

This Appendix provides details on the remaining runtime transactions (#2 and #3) introduced in Section 9.5.

B.1 Transactions for Generating Notifications

For notifications related to contract termination, the notification mechanism is updated because this is a special case: terminating a contract can change the state of multiple powers, obligations, and surviving Obligation. After updating the contract state accordingly, a notification event is generated. Listing B.1 shows the improved Xtend transformation to change the contract state, powers, obligations, and surviving Obligation, with generated notifications.

```
1  async p_«powerName»_«stateMethod»_contract(ctx, contractId) {
2       const cid = new ClientIdentity(ctx.stub);
3       let roleObj;
4  // ... Omitted code
5                 let controllers = contract.powers.«powerName»._controller
6
7         if(!contract.accessPolicy.hasPermesstion('grant','read', contract.powers.«powerName», roleObj, contract.«
       powerName».getController(controllers.length - 1)) ||
8                   !contract.accessPolicy.isValid(new Rule('grant','read', contract.powers.«powerName», roleObj, contract
       .«powerName».getController(controllers.length - 1))) ){
9                     throw new Error('access denied...')
10                  }
11        // Contract state notification
12        const statM="«stateMethod»"
13        // Notify
14        controllers = contract._controller
15        let MSG= "Contract "+Contract._name+" is "+  stateM+', '+ contract.id;
16        contract.notified.message.push({name: contract._name, message: MSG, roles:contract.accessPolicy.permissionValid(
       contract,contract._controller,contract.getController(controllers.length - 1), contract) , time: new Date().
       toISOString()})
17
18        for (let index in contract.obligations) {
19          const obligation = contract.obligations[index]
20          «IF stateMethod.equals("suspended")»
21          obligation._suspendedByContractSuspension = true
22          obligation.suspended()
23          controllers = obligation._controller
24          let MSG= "Obligation "+obligation.name+" is suspended By Contract Suspension, "+obligation.contract.id;
25          contract.notified.message.push({name: obligation.name, message: MSG, roles:contract.accessPolicy.
       permissionValid(obligation.name,[obligation.creditor,obligation.debtor],obligation.getController(controllers.length
        - 1), contract) , time: new Date().toISOString()})
26          «ELSEIF stateMethod.equals("resumed")»
```

```
27            if (obligation._suspendedByContractSuspension === true){
28              obligation.resumed()
29              controllers = obligation._controller
30              let MSG= "Obligation "+obligation.name+" is resumed By Contract resume, "+obligation.contract.id;
31              contract.notified.message.push({name: obligation.name, message: MSG, roles:contract.accessPolicy.
        permissionValid(obligation.name,[obligation.creditor,obligation.debtor],obligation.getController(controllers.length
         - 1), contract) , time: new Date().toISOString()})
32
33            }
34            «ELSEIF stateMethod.equals("terminated")»
35            obligation.terminated({emitEvent: false})
36            controllers = obligation._controller
37            let MSG= "Obligation "+obligation.name+" is terminated By Contract termination, "+obligation.contract.id;
38            contract.notified.message.push({name: obligation.name, message: MSG, roles:contract.accessPolicy.
        permissionValid(obligation.name,[obligation.creditor,obligation.debtor],obligation.getController(controllers.length
         - 1), contract) , time: new Date().toISOString()})
39
40            «ENDIF»
41          }
42          for (let index in contract.survivingObligations) {
43            const obligation = contract.survivingObligations[index]
44            «IF stateMethod.equals("suspended")»
45            obligation._suspendedByContractSuspension = true
46            obligation.suspended()
47            controllers = obligation._controller
48            let MSG= "survivingObligations "+obligation.name+" is suspended By Contract Suspension, "+obligation.contract.
        id;
49            contract.notified.message.push({name: obligation.name, message: MSG, roles:contract.accessPolicy.
        permissionValid(obligation.name,[obligation.creditor,obligation.debtor],obligation.getController(controllers.length
         - 1), contract) , time: new Date().toISOString()})
50
51            «ELSEIF stateMethod.equals("resumed")»
52            if (obligation._suspendedByContractSuspension === true){
53              obligation.resumed()
54              controllers = obligation._controller
55              let MSG= "survivingObligation "+obligation.name+" is resumed By Contract resume, "+obligation.contract.id;
56              contract.notified.message.push({name: obligation.name, message: MSG, roles:contract.accessPolicy.
        permissionValid(obligation.name,[obligation.creditor,obligation.debtor],obligation.getController(controllers.length
         - 1), contract) , time: new Date().toISOString()})
57
58            }
59            «ELSEIF stateMethod.equals("terminated")»
60            obligation.terminated()
61            controllers = obligation._controller
62            let MSG= "survivingObligation "+obligation.name+" is terminated By Contract termination, "+obligation.contract
        .id;
63            contract.notified.message.push({name: obligation.name, message: MSG, roles:contract.accessPolicy.
        permissionValid(obligation.name,[obligation.creditor,obligation.debtor],obligation.getController(controllers.length
         - 1), contract) , time: new Date().toISOString()})
64
65            «ENDIF»
66          }
67          for (let index in contract.powers) {
68            const power = contract.powers[index]
69            if (index === '«powerName»') {
70              continue;
71            }
72            «IF stateMethod.equals("suspended")»
73            power._suspendedByContractSuspension = true
74            power.suspended()
75            controllers = power._controller
76             let MSG= "Power "+power.name+" is suspended By Contract Suspension, "+power.contract.id;
77             contract.notified.message.push({name: power.name, message: MSG, roles:contract.accessPolicy.permissionValid(
        power.name,[power.creditor,power.debtor],power.getController(controllers.length - 1), contract) , time: new Date().
        toISOString()})
78
79            «ELSEIF stateMethod.equals("resumed")»
80            if (power._suspendedByContractSuspension === true){
81              power.resumed()
82              controllers = power._controller
83             let MSG= "Power "+power.name+" is resumed By Contract resume, "+power.contract.id;
84             contract.notified.message.push({name: power.name, message: MSG, roles:contract.accessPolicy.permissionValid(
        power.name,[power.creditor,power.debtor],power.getController(controllers.length - 1), contract) , time: new Date().
        toISOString()})
85
86            }
87            «ELSEIF stateMethod.equals("terminated")»
88            power.terminated()
89              controllers = power._controller
90               let MSG= "Power "+power.name+" is terminated By Contract termination, "+power.contract.id;
91               contract.notified.message.push({name: power.name, message: MSG, roles:contract.accessPolicy.permissionValid
        (power.name,[power.creditor,power.debtor],power.getController(controllers.length - 1), contract) , time: new Date()
        .toISOString()})
92
93            «ENDIF»
94          }
95          if (contract.«stateMethod»() && contract.powers.«powerName».exerted()) {
96          // Notification
97            for (const message of contract.notified.message) {
98                  this.trigger_notification(ctx, message)
99                }
100
```

```
          await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
          return {successful: true}
        } else {
          return {successful: false}
        }
      } else {
        return {successful: false}
      }
    }'''
```

Listing B.1: Improved Xtend source code for a contract termination transaction that generates notifications.

B.2 Embedding a Two-Layer Security Mechanism in Transactions

Generating Two Security Layers for Triggering a `DataTransfer`: These transactions are invoked when an IoT sensor generates a data transfer for the smart contract. A transaction for IoT sensors is used to change the corresponding contract state in the ledger. Xtend is used to generate two security layers for these transactions. In the first layer, the transaction verifies that the API listener invoking the transaction is known and holds a valid certificate. In the second layer, the transaction checks that this identity has the appropriate credentials, i.e., it is authorized as the regulator, who acts as the controller of the SymboleoAC policy. Only when the API listener is both known and authorized will the IoT data transfer be accepted and the state of the contract be updated in the ledger.

```
async trigger_«variable.name»(ctx, args) {
          const cid = new ClientIdentity(ctx.stub);
          let roleObj;
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
            const contractState = await ctx.stub.getState(contractId)
            if (contractState == null) {
         return {successful: false}
          }
          const contract = deserialize(contractState.toString())

          const oldMessagesList = []
          oldMessagesList.push(contract.notified.message.slice())
          this.initialize(contract)
          if (contract.isInEffect() «survivEvent(variable.name)» ){
            // First security layer
            try{
                   roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
                   cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

                        if(roleObj === null ){
                         throw new Error('Unauthorized: Unknown access');
                    }

                }catch(err){
                    console.log('access control error: ', err)
                    return { successful: false, message: err.message }
                }// End of first layer
                // Second layer
                let controllers = contract.«variable.name»._controller
            if(!contract.accessPolicy.hasPermesstion('grant','read', contract.«variable.name»,roleObj, contract.«
    variable.name».getController(controllers.length - 1)) ||
                !contract.accessPolicy.isValid(new Rule('grant','read', contract.«variable.name», roleObj, contract.«
    variable.name».getController(controllers.length - 1)) ){
                  throw new Error('access denied...')
                }
            contract.«variable.name».happen(event)
            Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent, InternalEventType.
    contractEvent.Happened, contract.«variable.name»))
```

```
            // Notification
                  for (const message of contract.notified.message) {
                  if (!oldMessagesList[0].includes(message)) {
                    this.trigger_notification(ctx, message)
                  }
                }

            await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
            return {successful: true}
          } else {
            return {successful: false}
          }
        }
```

Listing B.2: Xtend source code for a transaction to trigger a data transfer, with two security layers.

Generating Two Security Layers for Violating an Obligation: These transactions are invoked to record violations of obligations. The condition is first evaluated, and if satisfied, the state of the obligation is changed to *violated*. Two security layers are added to control who is allowed to change the state of such obligation, specifically, only the performer of that obligation. First, the transaction verifies that the performer holds a valid certificate. Second, it checks that the performer has the appropriate permission through a call to `hasPermission()`, and that no constraints override or deny this permission through a call to `isValid()`.

```
async trigger_«variable.name»(ctx, args) {
          const cid = new ClientIdentity(ctx.stub);
          let roleObj;
          const inputs = JSON.parse(args);
          const contractId = inputs.contractId;
          const event = inputs.event;
            const contractState = await ctx.stub.getState(contractId)
            if (contractState == null) {
         return {successful: false}
          }
          const contract = deserialize(contractState.toString())
          // Notification
          const oldMessagesList = []
          oldMessagesList.push(contract.notified.message.slice())
          this.initialize(contract)
          if (contract.isInEffect() «survivEvent(variable.name)» ){
            // First security layer
            try{
                  roleObj = contract.authenticate(cid.getAttributeValue('HF.role'), cid.getAttributeValue('HF.name'),
                  cid.getAttributeValue('organization'), cid.getAttributeValue('department'),contract)

                        if(roleObj === null ){
                         throw new Error('Unauthorized: Unknown access');
                    }

                }catch(err){
                    console.log('access control error: ', err)
                    return { successful: false, message: err.message }
                }// End of first layer
                // Second layer
                let controllers = contract.«variable.name»._controller
          if(!contract.accessPolicy.hasPermesstion('grant','read', contract.«variable.name»,roleObj, contract.«
    variable.name».getController(controllers.length - 1)) ||
                !contract.accessPolicy.isValid(new Rule('grant','read', contract.«variable.name», roleObj, contract.«
    variable.name».getController(controllers.length - 1)) ){
                  throw new Error('access denied...')
                } // End of second layer
          contract.«variable.name».happen(event)
          Events.emitEvent(contract, new InternalEvent(InternalEventSource.contractEvent, InternalEventType.
    contractEvent.Happened, contract.«variable.name»))
           // Notification
                  for (const message of contract.notified.message) {
                  if (!oldMessagesList[0].includes(message)) {
                    this.trigger_notification(ctx, message)
                  }
                }

          await ctx.stub.putState(contractId, Buffer.from(serialize(contract)))
          return {successful: true}
```

```
47            } else {
48              return {successful: false}
49            }
50          }
```

Listing B.3: Xtend source code for a transaction for violating an obligation, with two security layers.

Appendix C

Meat Sale Contract (JavaScript)

This appendix presents the complete `MeatSale.js` file for the Meat Sale contract evaluated in Section 10.2. The generated JavaScript class includes all declarations defined in the SymboleoAC specification as contract properties with the assigned controller and performer, and the contract parameters are reflected in the constructor of the generated class. In addition, all obligations and powers are instantiated with the assigned controller. This file also contains the generated access control rules, as well as the legal situations (antecedents and consequents) of unconditional obligations and powers. The other JavaScript classes generated for the Meat Sale contract are available at https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/tree/main/MeatSale.

```
class MeatSale extends SymboleoContract {
    constructor(buyerP, sellerP, transportCoP, assessorP, regulatorP, storageP, shipperP, adminP, barcodeP, qnt, qlt,
     amt, curr, payDueDate, delAdd, effDate, delDueDateDays, interestRate) {
      super("MeatSale")
      this._name = "MeatSale"
      this.buyerP = buyerP
      this.sellerP = sellerP
      this.transportCoP = transportCoP
      this.assessorP = assessorP
      this.regulatorP = regulatorP
      this.storageP = storageP
      this.shipperP = shipperP
      this.adminP = adminP
      this.barcodeP = barcodeP
      this.qnt = qnt
      this.qlt = qlt
      this.amt = amt
      this.curr = curr
      this.payDueDate = payDueDate
      this.delAdd = delAdd
      this.effDate = effDate
      this.delDueDateDays = delDueDateDays
      this.interestRate = interestRate

      this.obligations = {};
      this.survivingObligations = {};
      this.powers = {};
      //notification
      this.notified = new Notified ('notified')
      // assign varaibles of the contract
          this.seller = new Seller("seller")

      this.seller.name._value = this.sellerP.name
      this.seller.returnAddress._value = this.sellerP.returnAddress
      this.seller.org._value = this.sellerP.org
      this.seller.dept._value = this.sellerP.dept
      this.seller.addController(this.seller)
      this.addRole(this.seller)
      this.buyer = new Buyer("buyer")

      this.buyer.name._value = this.buyerP.name
      this.buyer.warehouse._value = this.buyerP.warehouse
      this.buyer.org._value = this.buyerP.org
      this.buyer.dept._value = this.buyerP.dept
```

```
        this.buyer.addController(this.buyer)
        this.addRole(this.buyer)
        this.transportCo = new TransportCo("transportCo")

        this.transportCo.name._value = this.transportCoP.name
        this.transportCo.org._value = this.transportCoP.org
        this.transportCo.dept._value = this.transportCoP.dept
        this.transportCo.addController(this.transportCo)
        this.addRole(this.transportCo)
        this.assessor = new Assessor("assessor")

        this.assessor.name._value = this.assessorP.name
        this.assessor.org._value = this.assessorP.org
        this.assessor.dept._value = this.assessorP.dept
        this.assessor.addController(this.assessor)
        this.addRole(this.assessor)
        this.regulator = new Regulator("regulator")

        this.regulator.name._value = this.regulatorP.name
        this.regulator.org._value = this.regulatorP.org
        this.regulator.dept._value = this.regulatorP.dept
        this.regulator.addController(this.regulator)
        this.addRole(this.regulator)
        this.storage = new Storage("storage")

        this.storage.name._value = this.storageP.name
        this.storage.address._value = this.storageP.address
        this.storage.org._value = this.storageP.org
        this.storage.dept._value = this.storageP.dept
        this.storage.addController(this.storage)
        this.addRole(this.storage)
        this.shipper = new Shipper("shipper")

        this.shipper.name._value = this.shipperP.name
        this.shipper.org._value = this.shipperP.org
        this.shipper.dept._value = this.shipperP.dept
        this.shipper.addController(this.shipper)
        this.addRole(this.shipper)
        this.admin = new Admin("admin")

        this.admin.name._value = this.adminP.name
        this.admin.org._value = this.adminP.org
        this.admin.dept._value = this.adminP.dept
        this.admin.addController(this.admin)
        this.addRole(this.admin)
        this.goods = new Meat("goods")

        this.goods.quantity._value = this.qnt
        this.goods.quality._value = this.qlt
        this.goods.barcode._value = this.barcodeP
        this.goods.owner = this.seller
        this.goods.addController(this.seller)
        this.delivered = new Delivered("delivered")

        this.delivered.deliveryAddress._value = this.delAdd
        this.delivered.delDueDate._value = Utils.addTime(this.effDate, this.delDueDateDays, "days")
        this.delivered.addPerformer(this.transportCo)
        this.delivered.addController(this.seller)
        this.paidLate = new PaidLate("paidLate")

        this.paidLate.amount._value = (1 + this.interestRate / 100) * this.amt
        this.paidLate.currency._value = this.curr
        this.paidLate.from._value = this.buyer
        this.paidLate.to._value = this.seller
        this.paidLate.addPerformer(this.buyer)
        this.paidLate.addController(this.buyer)
        this.paid = new Paid("paid")

        this.paid.amount._value = this.amt
        this.paid.currency._value = this.curr
        this.paid.from._value = this.buyer
        this.paid.to._value = this.seller
        this.paid.payDueDate._value = this.payDueDate
        this.paid.addPerformer(this.buyer)
        this.paid.addController(this.buyer)
        this.temperature = new Alert("temperature")

        this.temperature.condition._value = "value > 2"
        this.temperature.window._value = 10
        this.temperature.count._value = 1
        this.temperature.addController(this.seller)
        this.temperature.addPerformer(this.regulator)
        this.humidity = new Alert("humidity")

        this.humidity.condition._value = "value < 85 OR value > 90"
        this.humidity.window._value = 10
        this.humidity.count._value = 1
        this.humidity.addController(this.seller)
        this.humidity.addPerformer(this.regulator)
        this.passwordNotification = new PasswordNotification("passwordNotification")

        this.passwordNotification.addPerformer(this.transportCo)
```

```
136         this.passwordNotification.addController(this.transportCo)
137         this.inspectedQuality = new InspectedQuality("inspectedQuality")
138 
139         this.inspectedQuality.addPerformer(this.assessor)
140         this.inspectedQuality.addController(this.assessor)
141         this.unLoaded = new UnLoaded("unLoaded")
142 
143         this.unLoaded.addPerformer(this.assessor)
144         this.unLoaded.addController(this.assessor)
145         this.accessPolicy = new ACPolicy([this.seller])
146         this.addController(this.seller);
147         this.addController(this.buyer);
148 
149 
150         // create instance of triggered obligations
151         this.deliverySituation = new LegalSituation();
152 
153         this.deliverySituation.addConsequentOf({_type: 'eventCondition', resource:"delivered", resourceType:"Delivered"}
            )
154         this.deliverySituation.addConsequentOf({ leftSide:'contract.delivered.deliveryAddress._value', op:'===',
          rightSide: 'contract.buyer.warehouse._value', _type: 'Condition'})
155 
156         this.deliverySituation.addConsequentOf({_type: 'eventCondition', resource:"temperature", resourceType:"Alert"} )
157 
158         this.deliverySituation.addConsequentOf({_type: 'eventCondition', resource:"humidity", resourceType:"Alert"} )
159         this.obligations.delivery = new Obligation('delivery', this.buyer, this.seller, this, this.deliverySituation)
160         this.paymentSituation = new LegalSituation();
161 
162         this.paymentSituation.addConsequentOf({_type: 'eventCondition', resource:"paid", resourceType:"Paid"} )
163 
164         this.paymentSituation.addAntecedentOf({_type: 'eventCondition', resource:"unLoaded", resourceType:"UnLoaded"} )
165         this.obligations.payment = new Obligation('payment', this.seller, this.buyer, this, this.paymentSituation)
166 
167 
168         //Rules
169         this.accessPolicy.addRulee("grant", "read", this.goods.quantity, this.buyer, this.seller)
170         this.accessPolicy.addRulee("grant", "read", this.obligations.delivery, this.assessor, this.seller)
171         this.accessPolicy.addRulee("grant", "read", this.inspectedQuality, this.transportCo, this.assessor)
172         this.accessPolicy.addRulee("grant", "read", this.inspectedQuality, this.seller, this.assessor)
173         this.accessPolicy.addRulee("grant", "write", this.inspectedQuality, this.assessor, this.seller)
174         this.accessPolicy.addRulee("revoke", "read", this.goods.quality, this.buyer, this.seller)
175         this.accessPolicy.addRulee("grant", "read", this.temperature.value, this.buyer, this.seller)
176     }
177 }
178 
179     module.exports.MeatSale = MeatSale
```

Listing C.1: MeatSale class in JavaScript, with access control rules at the end.

Appendix D

Vaccine Procurement Contract (JavaScript)

This appendix presents the `VaccineProcurementC.js` file for the COVID-19 Vaccine Procurement contract evaluated in Section 10.3, with the generated access control rules at the end. The other JavaScript classes generated for the Vaccine Procurement contract are available at https://github.com/Smart-Contract-Modelling-uOttawa/SymboleoAC2SC/tree/main/VaccineProcurementC.

```
class VaccineProcurementC extends SymboleoContract {
    constructor(pfizerP, mcdcP, regulatorP, adminP, fdaP, worldcourierP, approval, unitPrice, minQuantity, maxQuantity
     ) {
      super("VaccineProcurementC")
      this._name = "VaccineProcurementC"
      this.pfizerP = pfizerP
      this.mcdcP = mcdcP
      this.regulatorP = regulatorP
      this.adminP = adminP
      this.fdaP = fdaP
      this.worldcourierP = worldcourierP
      this.approval = approval
      this.unitPrice = unitPrice
      this.minQuantity = minQuantity
      this.maxQuantity = maxQuantity

      this.obligations = {};
      this.survivingObligations = {};
      this.powers = {};
      //notification
      this.notified = new Notified ('notified')
      // assign varaibles of the contract
          this.regulator = new Regulator("regulator")

    this.regulator.name._value = this.regulatorP.name
    this.regulator.org._value = this.regulatorP.org
    this.regulator.dept._value = this.regulatorP.dept
      this.regulator.addController(this.regulator)
      this.addRole(this.regulator)
      this.admin = new Admin("admin")

    this.admin.name._value = this.adminP.name
    this.admin.org._value = this.adminP.org
    this.admin.dept._value = this.adminP.dept
      this.admin.addController(this.admin)
      this.addRole(this.admin)
      this.pfizer = new Manufacturer("pfizer")

    this.pfizer.name._value = this.pfizerP.name
    this.pfizer.org._value = this.pfizerP.org
    this.pfizer.dept._value = this.pfizerP.dept
      this.pfizer.addController(this.pfizer)
      this.addRole(this.pfizer)
      this.mcdc = new Government("mcdc")

    this.mcdc.name._value = this.mcdcP.name
    this.mcdc.org._value = this.mcdcP.org
    this.mcdc.dept._value = this.mcdcP.dept
```

```
      this.mcdc.addController(this.mcdc)
      this.addRole(this.mcdc)
      this.fda = new FDA("fda")

    this.fda.name._value = this.fdaP.name
    this.fda.org._value = this.fdaP.org
    this.fda.dept._value = this.fdaP.dept
      this.fda.addController(this.fda)
      this.addRole(this.fda)
      this.worldcourier = new WorldCourier("worldcourier")

    this.worldcourier.name._value = this.worldcourierP.name
      this.worldcourier.org._value = this.worldcourierP.org
      this.worldcourier.dept._value = this.worldcourierP.dept
      this.worldcourier.addController(this.worldcourier)
      this.addRole(this.worldcourier)
      this.requested = new Requested("requested")

      this.requested.addPerformer(this.mcdc)
    this.requested.addController(this.mcdc)
      this.leadtimeINform = new LeadtimeInformedNegotiated("leadtimeINform")

      this.leadtimeINform.addPerformer(this.pfizer)
      this.leadtimeINform.addController(this.pfizer)
      this.notifiedOD = new NotifiedOfDelivery("notifiedOD")

      this.notifiedOD.addPerformer(this.pfizer)
      this.notifiedOD.addController(this.pfizer)
      this.delivered = new Delivered("delivered")

      this.delivered.addPerformer(this.worldcourier)
      this.delivered.addController(this.pfizer)
      this.invoiced = new Invoiced("invoiced")

      this.invoiced.addPerformer(this.mcdc)
      this.invoiced.addController(this.mcdc)
      this.paid = new Paid("paid")

      this.paid.addPerformer(this.mcdc)
      this.paid.addController(this.mcdc)
      this.confirmed = new Confirmed("confirmed")

      this.confirmed.addPerformer(this.mcdc)
      this.confirmed.addController(this.mcdc)
      this.lawStopWork = new StopWork("lawStopWork")

      this.lawStopWork.addPerformer(this.mcdc)
      this.lawStopWork.addController(this.mcdc)
      this.regulationStopWork = new ThirdPartyStopWork("regulationStopWork")

      this.regulationStopWork.addPerformer(this.regulator)
      this.regulationStopWork.addController(this.regulator)
      this.judicialStopWork = new ThirdPartyStopWork("judicialStopWork")

      this.judicialStopWork.addPerformer(this.regulator)
      this.judicialStopWork.addController(this.regulator)
      this.adminStopWork = new ThirdPartyStopWork("adminStopWork")

      this.adminStopWork.addPerformer(this.regulator)
      this.adminStopWork.addController(this.regulator)
      this.govStopWork = new StopWork("govStopWork")

      this.govStopWork.addPerformer(this.mcdc)
      this.govStopWork.addController(this.mcdc)
      this.vaccineDose = new VaccineDose("vaccineDose")

      this.vaccineDose.price._value = this.unitPrice
      this.vaccineDose.FDAapproval._value = this.approval
      this.vaccineDose.owner = this.pfizer
      this.vaccineDose.addController(this.pfizer)
      this.agreedFromG = new Agreed("agreedFromG")

      this.agreedFromG.addPerformer(this.mcdc)
      this.agreedFromG.addController(this.mcdc)
      this.outsideRisk = new Risk("outsideRisk")

      this.outsideRisk.addPerformer(this.pfizer)
      this.outsideRisk.addController(this.pfizer)
      this.remain = new Remain("remain")

      this.remain.value._value = this.maxQuantity
      this.remain.owner = this.mcdc
      this.remain.addController(this.mcdc)
      this.paidAmount = new PaidAmount("paidAmount")

      this.paidAmount.value._value = 0
      this.paidAmount.owner = this.mcdc
      this.paidAmount.addController(this.mcdc)
      this.withdrewApproval = new WithdrewApproval("withdrewApproval")

      this.withdrewApproval.addPerformer(this.fda)
      this.withdrewApproval.addController(this.fda)
```

```
        this.mcdcTerminateAgreement = new TerminateAgreementG("mcdcTerminateAgreement")

        this.mcdcTerminateAgreement.addPerformer(this.mcdc)
        this.mcdcTerminateAgreement.addController(this.mcdc)
        this.pfizerTerminateAgreement = new TerminateAgreementM("pfizerTerminateAgreement")

        this.pfizerTerminateAgreement.addPerformer(this.pfizer)
        this.pfizerTerminateAgreement.addController(this.pfizer)
        this.temperature = new Alert("temperature")

        this.temperature.condition._value = "value > -80"
        this.temperature.window._value = 10
        this.temperature.count._value = 5
        this.temperature.addController(this.pfizer)
        this.temperature.addPerformer(this.regulator)
        this.humidity = new Alert("humidity")

        this.humidity.condition._value = "value > 70"
        this.humidity.window._value = 15
        this.humidity.count._value = 3
        this.humidity.addController(this.pfizer)
        this.humidity.addPerformer(this.regulator)
        this.shock = new Alert("shock")

        this.shock.condition._value = "value > 2.5"
        this.shock.window._value = 5
        this.shock.count._value = 1
        this.shock.addController(this.pfizer)
        this.shock.addPerformer(this.regulator)
        this.lightExposure = new Alert("lightExposure")

      this.lightExposure.condition._value = "value > 0"
      this.lightExposure.window._value = 1
      this.lightExposure.count._value = 1
      this.lightExposure.addController(this.pfizer)
        this.lightExposure.addPerformer(this.regulator)
        this.sealOpen = new Alert("sealOpen")

        this.sealOpen.condition._value = "value = 1"
        this.sealOpen.window._value = 1
        this.sealOpen.count._value = 1
        this.sealOpen.addController(this.pfizer)
        this.sealOpen.addPerformer(this.regulator)
        this.accessPolicy = new ACPolicy([this.regulator])
        this.addController(this.pfizer);
        this.addController(this.mcdc);

        // create instance of triggered obligations
        this.oRequestVaccineDosageSituation = new LegalSituation();

        this.oRequestVaccineDosageSituation.addConsequentOf({ leftSide:'contract.remain.value._value', op:'<',
       rightSide: 'contract.minQuantity', _type: 'Condition'})

        this.oRequestVaccineDosageSituation.addConsequentOf( {_type: 'stateCondition', resourceType:"obligation",
       resource: "oAgreedOnRequest", state:"fulfillment"} )

        this.oRequestVaccineDosageSituation.addConsequentOf( {_type: 'stateCondition', resourceType:"obligation",
       resource: "oDeliver", state:"fulfillment"} )

        this.oRequestVaccineDosageSituation.addConsequentOf( {_type: 'stateCondition', resourceType:"obligation",
       resource: "oAssign", state:"fulfillment"} )

        this.oRequestVaccineDosageSituation.addConsequentOf({_type: 'eventCondition', resource:"mcdcTerminateAgreement",
        resourceType:"TerminateAgreementG"} )

        this.oRequestVaccineDosageSituation.addConsequentOf({_type: 'eventCondition', resource:"pfizerTerminateAgreement
       ", resourceType:"TerminateAgreementM"} )

        this.obligations.oRequestVaccineDosage = new Obligation('oRequestVaccineDosage', this.pfizer, this.mcdc, this,
       this.oRequestVaccineDosageSituation)

        this.survivingObligations.oWithdrewApproval = new Obligation('oWithdrewApproval', this.mcdc, this.pfizer, this,
       null, true)


      this.accessPolicy.addRulee("grant", "read", this.vaccineDose, this.mcdc, this.pfizer)
      this.accessPolicy.addRulee("grant", "read", this.vaccineDose.FDAapproval, this.worldcourier, this.pfizer)
      this.accessPolicy.addRulee("grant", "read", this.delivered, this.mcdc, this.pfizer)
      this.accessPolicy.addRulee("revoke", "write", this.temperature.value, this.pfizer, this.regulator)
      this.accessPolicy.addRulee("revoke", "write", this.humidity.value, this.pfizer, this.regulator)
      this.accessPolicy.addRulee("revoke", "write", this.shock.value, this.pfizer, this.regulator)
      this.accessPolicy.addRulee("revoke", "write", this.lightExposure.value, this.pfizer, this.regulator)
      this.accessPolicy.addRulee("revoke", "write", this.sealOpen.value, this.pfizer, this.regulator)
      this.accessPolicy.addRulee("grant", "read", this.withdrewApproval, this.pfizer, this.fda)
    }
}

    module.exports.VaccineProcurementC = VaccineProcurementC
```

Listing D.1: VaccineProcurementC class in JavaScript, with access control rules at the end.

Appendix E

Multiple Instances of Multiple Contracts with Shared Parties

This appendix provides three screenshots with additional and successful output results for the experiment conducted in Section 10.5. These screenshots illustrate end-to-end execution logs for the experiment involving multi-instances of multiple contracts:

1. Secure sensor publishing (Figure E.1);
2. CEP detection and alert generation (Figure E.2); and
3. Smart contract enforcement and instance-specific notification delivery (Figure E.3).

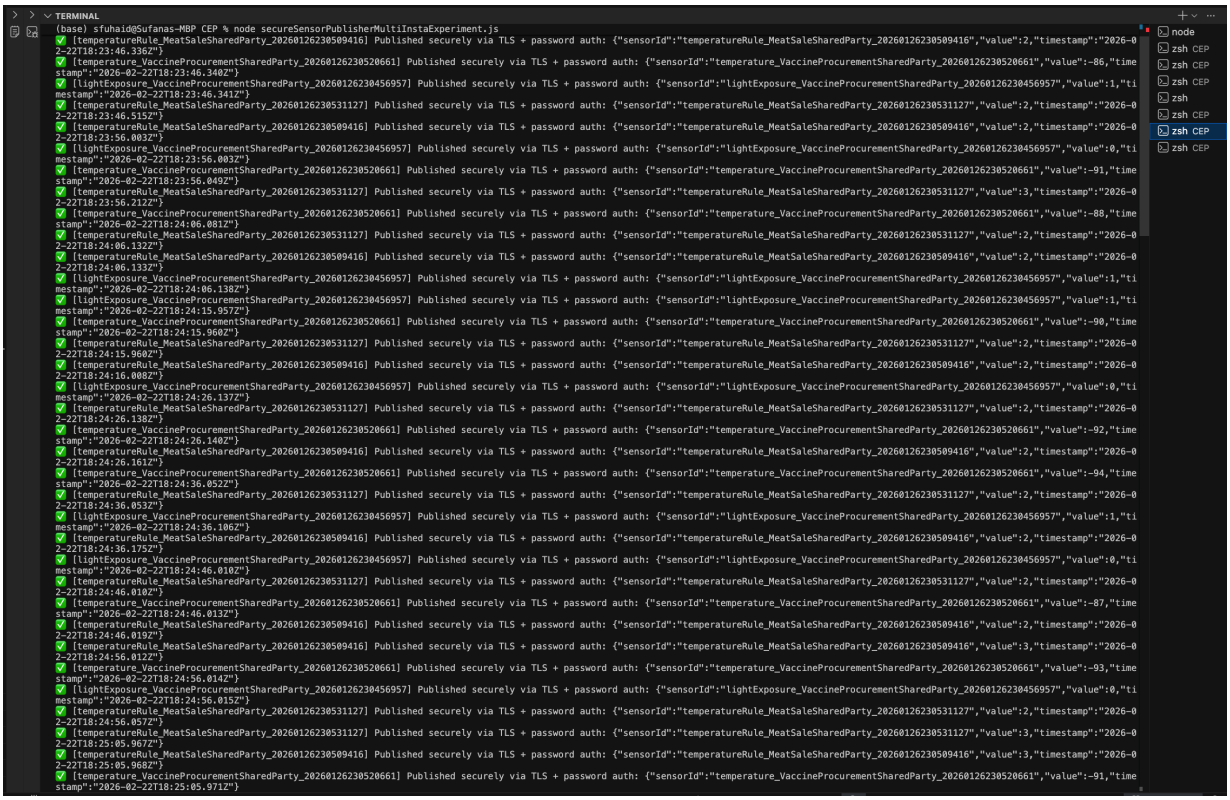


Figure E.1: Virtual sensors publish instance-specific readings to RabbitMQ using TLS and sensor credentials.

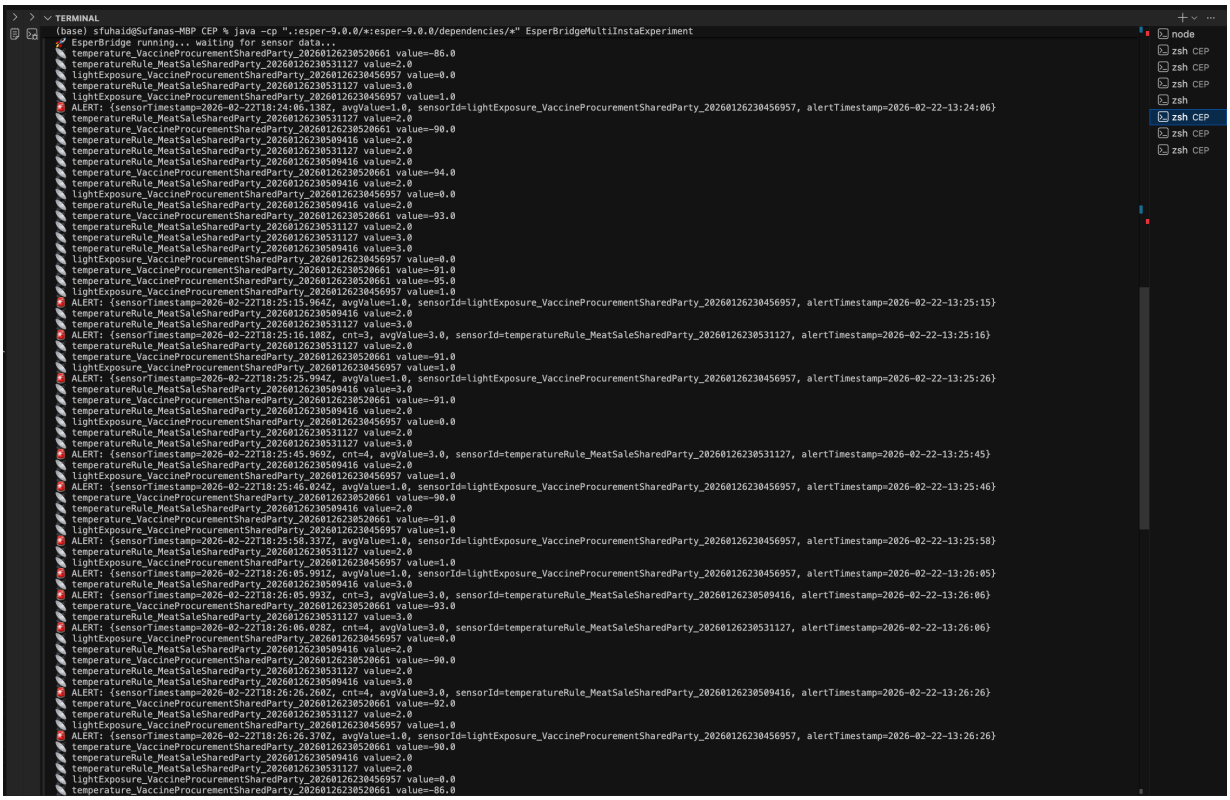


Figure E.2: Esper CEP evaluates the per-instance rules and emits alerts when violations are detected.

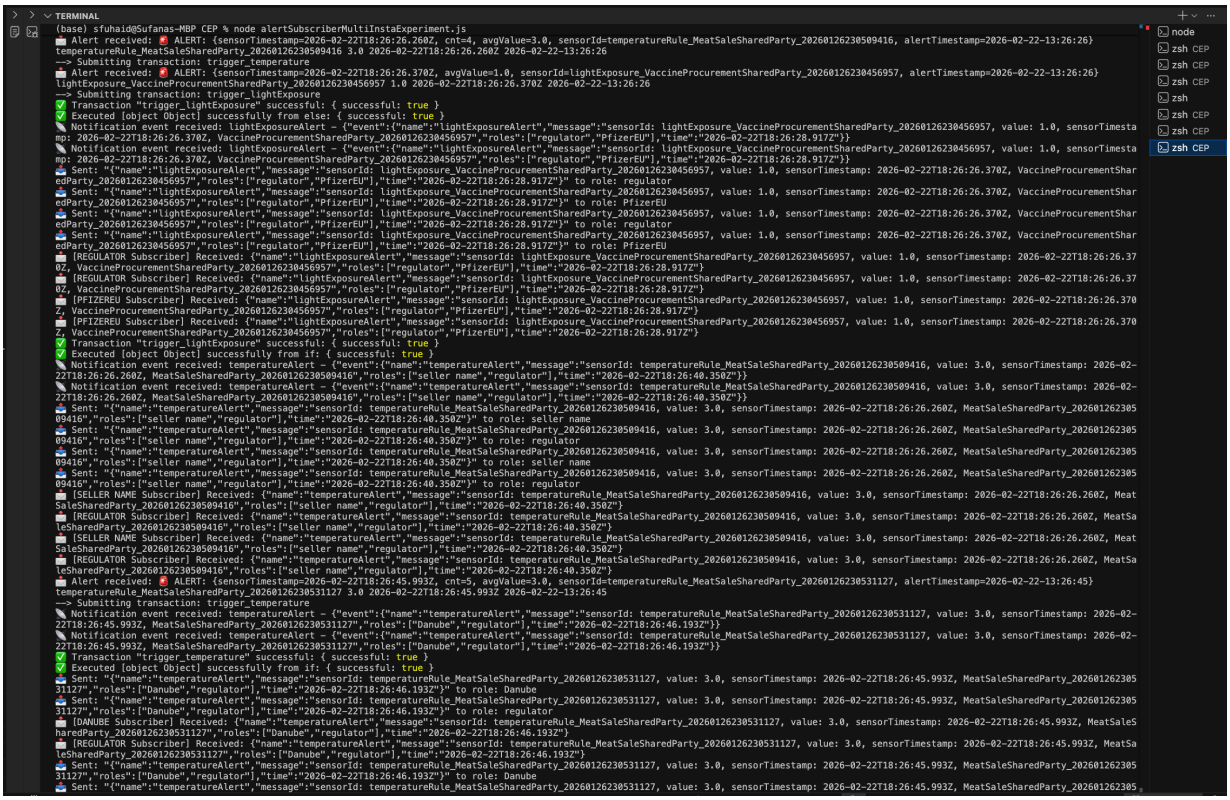


Figure E.3: Alert subscriber consumes CEP alerts and invokes the correct SymboleoAC transaction per instance, and notifications are delivered to authorized subscribers only.