UNIVERSITÀ DEGLI STUDI DI CAMERINO

SCHOOL OF ADVANCED STUDIES

DOCTOR OF PHILOSOPHY IN :

Blockchain And Distributed Ledger Technology

CURRICULUM :

Methodologies, technologies and tools

CYCLE **XXXVIII**

Analysis and verification of smart contracts with behavioural types

PhD Candidate:
Gerardin Elvis Konjoh Selabi

Supervisor:
Dr. Maurizio Murgia

Co-supervisor:
Prof. Emilio Tuosto

Coordinator of the PhD Programme:
Prof. Flavio Corradini

Date of award:
June 23, 2026

Imprint

Published by the University of Camerino, Camerino, Italy.

Colophon

La borsa di dottorato cofinanziata con risorse dell'Unione europea-NextGeneration EU Piano Nazionale di Ripresa e Resilienza Missione 4 - Componente 1 - Riforma 4.1 Riforma dei Dottorati - Investimento 4.1 Borse Generici Ricerca PNRR - J11J22002120006

DECLARATION

I herewith declare that I have produced this paper under the supervision of Prof. Maurizio Murgia and Prof. Emilio Tuosto.
I declare that I have read and understood the University of Camerino's regulations on plagiarism and academic integrity.
I declare that I have not used any material or ideas from any other source without proper citation.

Gerardin Elvis Konjoh Selabi
June 23, 2026

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the School of Advanced Studies of University of Camerino.

Riassunto

Gli smart contract distribuiti su piattaforme blockchain sono immutabili una volta deployati, rendendo la correttezza e la sicurezza preoccupazioni critiche che hanno portato a perdite finanziarie sostanziali dovute a vulnerabilità. Una proporzione significativa di queste vulnerabilità deriva da codice scritto manualmente piuttosto che dall'infrastruttura blockchain o dalle primitive crittografiche. Questa osservazione motiva un cambio di paradigma dallo sviluppo manuale del codice ad approcci model-driven che generano smart contract semanticamente corretti da specifiche formali.

Questa tesi presenta EDAM (Enhanced Data-Aware Machines), un modello comportamentale per specificare smart contract che bilancia espressività e trattabilità. Il framework estende le tradizionali macchine a stati finiti data-aware [3] con caratteristiche essenziali per un'ampia gamma di applicazioni di smart contract: controllo degli accessi dinamico basato sui ruoli che consente l'assegnazione e la revoca dei ruoli a runtime, gestione dei partecipanti che supporta partecipanti illimitati e variabili, e modellazione esplicita delle interazioni inter-contratto attraverso tentativi di chiamata con gestione di successo e fallimento. La semantica formale di EDAM è fondata su tecniche consolidate della teoria dei tipi comportamentali, dei calcoli di processo e della teoria delle macchine a stati finiti, consentendo un ragionamento rigoroso sul comportamento dei contratti.

La tesi contribuisce con una toolchain completa che integra modellazione, generazione di codice, generazione di test e validazione in una metodologia unificata. Sviluppiamo un motore di generazione di codice che produce automaticamente smart contract Solidity da specifiche EDAM. Il codice generato implementa fedelmente il modello formale, garantendo che il comportamento stabilito a livello di modello sia preservato nel codice eseguibile. Il processo di generazione del codice utilizza una rappresentazione intermedia in JavaScript Object Notation (JSON), che consente una generazione di codice indipendente dalla piattaforma con estensioni in corso per supportare ulteriori piattaforme blockchain come Aptos.

Presentiamo una metodologia automatizzata di generazione di test che produce suite di test eseguibili da specifiche EDAM. L'approccio combina la generazione simbolica di tracce utilizzando la semantica formale implementata in OCaml con l'esplorazione randomizzata della rete di Finite State Machine (FSM), consentendo la derivazione di tracce concrete attraverso l'assegnazione casuale di valori e la risoluzione di vincoli Satisfiability Modulo Theories (SMT). Questa metodologia esplora sistematicamente lo spazio degli stati per generare tracce che esercitano transizioni, guardie e vincoli sui ruoli, producendo suite di test eseguibili per framework di testing standard come Hardhat. Il processo è completamente automatizzato e può essere integrato nel flusso di sviluppo.

La nostra valutazione dimostra l'espressività e la praticità dell'approccio attraverso un benchmark diversificato di smart contract, inclusi contratti dal repository Azure [148], contratti token standard (Ethereum Request for Comments 20 (Token Standard) (ERC20)), protocolli Decentralized Finance (DeFi) (Automated Market Makers (AMMs)) e sistemi multi-coordinatore. La valutazione dimostra l'espressività attraverso la modellazione di caratteristiche essenziali, mostrando che l'approccio è in grado di modellare un'ampia gamma di funzionalità di smart contract.

La metodologia di validazione impiega un approccio multi-faccettato che combina l'analisi della copertura del codice, il mutation testing per validare la correttezza del codice generato e l'efficacia delle suite di test, e la cross-validazione applicando i test generati ad altre implementazioni consolidate. Questo approccio di cross-validazione mostra che le nostre suite di test generate sono applicabili per validare implementazioni esistenti di smart contract, fornendo evidenza della qualità e della correttezza sia del codice generato sia della metodologia di testing. I risultati indicano empiricamente che il nostro approccio model-driven produce contratti e suite di test che preservano la struttura e la semantica del modello formale e possono essere applicati per validare implementazioni esistenti di smart contract.

A differenza degli approcci esistenti che affrontano fasi isolate del ciclo di vita dello sviluppo, EDAM fornisce una toolchain integrata che garantisce coerenza tra specifiche, codice generato e suite di test. Il framework dimostra che i tipi comportamentali forniscono una solida base per la modellazione e la verifica di smart contract, consentendo lo sviluppo di framework unificati che integrano modellazione, generazione di codice, generazione di test e validazione. Sebbene la nostra implementazione sia rivolta alle piattaforme blockchain, la metodologia è indipendente dalla piattaforma e può generalizzarsi ad altre architetture orientate ai servizi e distribuite. I risultati mostrano che gli approcci model-driven possono produrre smart contract di alta qualità e suite di test complete, contribuendo al progresso delle pratiche di sviluppo sicuro di smart contract.

Parole chiave. smart contract, tipi comportamentali, metodi formali, sviluppo model-driven, generazione di codice, generazione di test, verifica, blockchain

Abstract

Smart contracts deployed on blockchain platforms are immutable once deployed, making correctness and security critical concerns that have led to substantial financial losses due to vulnerabilities. A significant proportion of these vulnerabilities stem from human-written code rather than blockchain infrastructure or cryptographic primitives. This observation motivates a paradigm shift from manual code development to model-driven approaches that generate semantically correct smart contracts from formal specifications.

This thesis presents EDAM (Enhanced Data-Aware Machines), a behavioural model for specifying smart contracts that balances expressiveness with tractability. The framework extends traditional data-aware finite state machines [3] with essential features for a wide range of smart contract applications: dynamic role-based access control enabling runtime role assignment and revocation, participant management supporting unbounded and varying participants, and explicit modelling of inter-contract interactions through call tries with success and failure handling. The formal semantics of EDAM are grounded in established techniques from behavioural type theory, process calculi, and finite state machine theory, enabling rigorous reasoning about contract behaviour.

The thesis contributes a comprehensive toolchain that integrates modelling, code generation, test generation, and validation in a unified methodology. We develop a code generation engine that automatically produces Solidity smart contracts from EDAM specifications. The generated code faithfully implements the formal model, ensuring that the behaviour established at the model level is preserved in the executable code. The code generation process uses an intermediate JavaScript Object Notation (JSON) representation, which enables platform-agnostic code generation with ongoing extensions to support additional blockchain platforms such as Aptos.

We present an automated test generation methodology that produces executable test suites from EDAM specifications. The approach combines symbolic trace generation using the formal semantics implemented in OCaml with randomized exploration of the Finite State Machine (FSM) network, enabling concrete trace derivation through random value assignment and Satisfiability Modulo Theories (SMT) constraint solving. This methodology systematically explores the state space to generate traces that exercise transitions, guards, and role constraints, producing executable test suites for standard testing frameworks such as Hardhat. The process is fully automated and can be integrated into the development workflow.

Our evaluation demonstrates the expressiveness and practicality of the approach through a diverse benchmark of smart contracts, including contracts from the Azure repository [148], standard token contracts (Ethereum Request for Comments 20 (Token Standard) (ERC20)), Decentralized Finance (DeFi) protocols (Automated Market Makers (AMMs)), and multi-coordinator systems. The evaluation demonstrates expressiveness through the modelling of essential features, showing that the approach is able to model a wide range of smart contract features.

The validation methodology employs a multi-faceted approach that combines code coverage analysis, mutation testing to validate the correctness of the generated code and the effectiveness of the test suites, and cross-validation by applying the generated tests to other established implementations. This cross-validation approach shows that our generated test suites are applicable to validate existing smart contract implementations, providing evidence of the quality and correctness of both the generated code and the testing methodology. The results empirically indicate that our model-driven approach produces contracts and test suites that preserve the structure and semantics of the formal model and can be applied to validate existing smart contract implementations.

Unlike existing approaches that address isolated phases of the development lifecycle, EDAM provides an integrated toolchain that ensures consistency between specifications,

generated code, and test suites. The framework shows that behavioural types provide a solid foundation for smart contract modelling and verification, enabling the development of unified frameworks that integrate modelling, code generation, test generation, and validation. Although our implementation targets blockchain platforms, the methodology is platform-agnostic and may generalise to other service-oriented and distributed architectures. The results show that model-driven approaches can produce high-quality smart contracts and comprehensive test suites, contributing to the advancement of secure smart contract development practices.

Keywords. smart contracts, behavioural types, formal methods, model-driven development, code generation, test generation, verification, blockchain

To my beloved wife and kids for your enormous sacrifices, patience and support.
To my parents and siblings for your endless support and encouragement.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to God, whose grace and guidance have been the foundation of this journey. Without His blessings, this achievement would not have been possible.
I am profoundly grateful to my supervisor, Dr. Maurizio Murgia, for his unwavering support and guidance throughout this research. Your harsh yet sincere feedback has been instrumental in my growth as a researcher. Your commitment to excellence and your willingness to challenge my ideas have pushed me to achieve more than I thought possible.
I owe a special debt of gratitude to my co-supervisor, Prof. Emilio Tuosto, who has always been there to support me. Seeing you working has amazed me, and I have learned a great deal from you. The discipline to carry out this long journey came massively from you. Even beyond academia, you were always there, and for that, I am very grateful.
I would like to thank the Gran Sasso Science Institute (GSSI) for all the support, both financial and academic. Your resources and environment have made me feel at home in a place where I would like to foster and continue my research career.
I am grateful to the University of Camerino and the PNRR (Piano Nazionale di Ripresa e Resilienza) program, which gave us this opportunity to explore the beautiful world of blockchain. For the funding and the platform to pursue this research, I am deeply thankful.
I would like to express my appreciation to Prof. Antonio Ravara for his guidance. The time spent with you has been invaluable, and your insights have significantly contributed to this work.
I am grateful to Prof. Alberto Lluch Lafuente for your kindness and great support during my period abroad. Your mentorship and encouragement have been essential to my development.
I am grateful to other professors who have contributed to my academic journey: Prof. Andrey Rivkin, Prof. Flavio Corradini, and Prof. Massimo Bartoletti. Your teachings and guidance have been invaluable and I am very grateful for your support.
To my colleagues of cycle 38, we went through this all together. This journey could not have been completed without you. Your companionship, shared struggles, and mutual support have made this experience memorable and meaningful.
I would like to thank my colleagues at GSSI for creating a stimulating and supportive research environment. Your collaboration and friendship have enriched my academic experience.
To my friends in L'Aquila, without whom life would not have been the same. Your friendship, support, and the moments we shared have made my time in Italy truly special.
To my friends in Cameroon, thank you for your constant encouragement and for being a source of strength throughout this journey, even from afar.
To my family, words cannot express my gratitude. Your unconditional love, support, and sacrifices have been the driving force behind this achievement. This thesis is as much yours as it is mine.
To my Crypto Education Community and to all those whom I have not mentioned, please know that you are not forgotten. Each of you has contributed to this achievement in your own way, and I am very grateful. This work is the result of the collective support, encouragement, and inspiration from all the wonderful people in my life.

Contents

List of Figures

List of Tables

List of Acronyms

ABI	Application Binary Interface
AMM	Automated Market Maker
AMMs	Automated Market Makers
API	Application Programming Interface
ASM	Abstract State Machine
BDD	Binary Decision Diagram
BFT	Byzantine Fault Tolerance
BIP	Behaviour, Interaction, Priority
BPMN	Business Process Model and Notation
CTL	Computation Tree Logic
CTL*	Extended Computation Tree Logic
DAG	Directed Acyclic Graph
DAML	Digital Asset Modeling Language
DAO	Decentralized Autonomous Organization
DeFi	Decentralized Finance
DoS	Denial of Service
DSL	Domain-Specific Language
DSML	Domain-Specific Modeling Language
ECDSA	Elliptic Curve Digital Signature Algorithm
EDAM	Enhanced Data-Aware Machines
ERC	Ethereum Request for Comments
ERC20	Ethereum Request for Comments 20 (Token Standard)
ERC721	Ethereum Request for Comments 721 (Non-Fungible Token Standard)
EVM	Ethereum Virtual Machine
FSM	Finite State Machine
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
LTL	Linear Temporal Logic
LTS	Labelled Transition System
MRO	Method Resolution Order
NFT	Non-Fungible Token
PBFT	Practical Byzantine Fault Tolerance
PoS	Proof of Stake
PoW	Proof of Work
RBAC	Role-Based Access Control
SC	Smart Contract
SCs	Smart Contracts
SMT	Satisfiability Modulo Theories
UML	Unified Modeling Language
UTXO	Unspent Transaction Output
VM	Virtual Machine

Chapter I: Introduction

I.1 Motivation

The advent of blockchain technology has revolutionised the way we conceive of decentralised systems and digital transactions. Smart contracts [188], as self-executing programs deployed on blockchain platforms, have emerged as a fundamental building block for decentralised applications, which enable the automated execution of contractual agreements without the need for trusted intermediaries. The immutable nature of blockchain deployments, combined with the substantial financial and operational stakes involved in many smart contract applications, makes the correctness and security of these programs a critical concern.

Unlike traditional software systems where bugs can be patched through updates, smart contracts deployed on public blockchains remain unalterable once deployed. This immutability, while providing security guarantees against unauthorized modifications, also means that any errors or vulnerabilities in the contract code become permanent and potentially exploitable. The financial consequences of smart contract failures have been demonstrated by numerous high-profile incidents [20, 104, 18, 207, 208], resulting in losses amounting to hundreds of millions of dollars. These incidents underscore the urgent need for rigorous verification methodologies that can ensure the correctness of smart contracts before deployment.

A fundamental observation underpins this thesis: a significant proportion of vulnerabilities observed in deployed smart contracts do not originate from the blockchain infrastructure itself, nor from cryptographic primitives, but from a much more fundamental source—human-written code [20]. Manual programming is inherently error-prone, especially in adversarial, irreversible, and financially sensitive environments such as blockchains. This observation motivates a paradigm shift in how we approach smart contract development.

Instead of directly implementing smart contracts in low-level programming languages, we should formally model their behaviour, verify the model against well-defined correctness, and generate the executable smart contract code from this verified model. This model-driven approach offers several structural advantages that address the fundamental challenges of smart contract verification.

To understand how formal models work, we must first introduce the fundamental concepts they employ. A *state* represents a particular configuration of the smart contract at a given moment, characterised by the values of its variables and the current stage of execution. A *transition* is a change from one state to another, triggered by an operation or event. *Guards* are conditions that must be satisfied for a transition to occur; they act as preconditions that ensure operations only execute when appropriate. *Roles* represent different permission levels or responsibilities assigned to participants, such as owner, buyer, or administrator. *Role assignment* determines which participants can perform which operations based on their assigned roles. These concepts form the building blocks of state machine-based models, where contracts are represented as collections of states connected by transitions that are guarded by conditions and restricted by role constraints.

Reasoning is performed at the semantic level (states, transitions, roles, invariants), not at the syntactic level of programming languages. This abstraction enables developers and verification tools to focus on the essential behaviour of the contract rather than implementation details. Formal models capture the *what* and *why* of contract behaviour, while code generation handles the *how*.

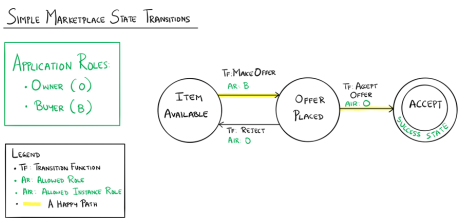
Common issues in smart contract specifications include *deadlocks*, where the system reaches a state from which no further transitions are possible; *invalid states*, where the contract enters configurations that violate business logic or safety properties; *missing guards*, where transitions

lack necessary preconditions to prevent unauthorized or unsafe operations; and *role inconsistencies*, where access control rules conflict or permit unintended interactions. These issues can be identified before code generation through analysis of the model's structure and semantics, enabling early detection and correction of design flaws.

The complexity of smart contract behaviour further compounds the verification challenge. Smart contracts are inherently stateful systems where the same function call can produce different results depending on the current state, the identity of the caller, and the roles assigned to participants. Access control mechanisms, temporal constraints, and complex state transitions create intricate behavioural patterns that are difficult to reason about using traditional software engineering practices [104]. Moreover, the interaction between multiple contracts in decentralised applications introduces additional complexity, requiring specification techniques that can reason about distributed system behaviour.

Formal methods offer a promising approach to address these challenges by providing mathematical foundations for specifying and verifying system behaviour. However, the gap between formal specifications and executable code remains a significant barrier to the widespread adoption of formal verification in smart contract development [152, 7, 82, 124, 30, 8]. Model-driven development approaches, which generate code from formal models, can bridge this gap by ensuring that the generated code faithfully implements the formal specification. This thesis explores how behavioural type systems, specifically the EDAM (Enhanced Data-Aware Machines) framework, can be used to model, generate, and verify smart contracts in a unified framework.

The design of EDAM is inspired by the Azure initiative of Microsoft [148], which advocates the use of Finite State Machines (FSMs) to specify the coordination of smart contracts. This idea, however, is not formalised; in fact, Azure's FSMs are informal sketches aiming to capture the correct executions of smart contracts. For instance, the FSM for a simple marketplace scenario (borrowed from [147]) specifies roles (Owner and Buyer) played by participants, as illustrated in Fig. I.1.



The sketch declares the roles (Owner and Buyer) played by participants.
In the initial state `Item Available` the buyer is allowed to make an offer, moving the protocol to the `Offer Placed` state where two options are possible: the owner either accepts the offer (making the protocol reach the success state `Accept`) or rejects the offer (moving back the protocol to `Item Available`).
The labels of the transitions specify which role executes with operations to make the protocol progress.

Figure I.1: Azure FSM for the simple marketplace scenario

Such an FSM informally specifies a protocol coordinating the participants enacting the roles owner and buyer, from a global standpoint; we call such specification a *coordination protocol*. A coordination protocol can be regarded as a global view—in the sense of choreographies—where the state of the protocol determines which operations are enabled. This resembles the execution model of monitors, where coordination protocols encapsulate a state that—through an Application Programming Interface (API)—concurrent processes can have exclusive access to. The API is basically a set of operations guarded by conditions set to maintain an invariant on the encapsulated state. The key differences between coordination protocols and monitors is that in the former participants are distributed and do not share memory, the invocation of an operation whose guard is not valid in the current state is simply ignored without preempting the caller, and therefore processes do not have to be awakened.

We aim to refine the approach of Azure so to enable algorithmic verification of behaviours of data-aware coordination protocols. In fact, as for monitors, the interplay among the operations that modify the state and the guards in the API can lead to unexpected behaviours when informal specifications are used. The sketch of Azure's approach does not clarify if a participant can play more roles simultaneously; for instance, it is not clear if an owner must be a different instance than

buyers. The labels distinguish roles and instances, assuming that there can be many instances of a same role, but the scope and quantification of roles is not clear. For instance, a requirement might specify that transitions between states can continue until the owner is satisfied with the offer made, but this does not clarify if, after a rejection, the new offer can be made by a new buyer or it must be the original one. Moreover, the sketch specifies neither the conditions enabling operations in a given state nor how operations change the state of the contract's variables; should the price of the item remain unchanged when the owner invokes the `Reject` operation? These ambiguities in informal specifications can lead to implementation errors and security vulnerabilities. The EDAM framework addresses these limitations by providing a formal, mathematically precise model that enables rigorous verification while preserving the intuitive state machine-based approach advocated by Azure [148].

I.2 Problem Statement

The development of smart contracts presents several interconnected challenges that existing approaches address only partially. These challenges span the entire development lifecycle, from initial specification to final deployment, and require integrated solutions that address multiple concerns simultaneously.

There is a fundamental tension between the expressiveness of modelling formalisms and their amenability to automated analysis. While highly expressive models can capture complex behavioural patterns, they often become computationally intractable for automated processing. Conversely, simpler models that are amenable to automated analysis may fail to capture essential aspects of smart contract behaviour, such as role-based access control, temporal constraints, and inter-contract interactions [1]. This trade-off manifests in several ways. Temporal logic specifications [56] can express rich behavioural properties but may require expensive model checking procedures. Process calculi [149] provide elegant compositional semantics but may lack direct support for blockchain-specific features. State machine models [140, 139] offer intuitive representations but may require extensions to capture complex access control patterns. The challenge is to develop a modelling framework that is sufficiently expressive to capture the essential features of smart contracts while remaining tractable for automated code and test generation.

The gap between formal models and executable code creates an implementation gap: even when a formal model accurately captures the intended behaviour, there is no guarantee that the manually written code correctly implements the model. This gap is particularly problematic in smart contract development, where a single implementation error can have catastrophic consequences. Manual translation from formal specifications to code can introduce implementation errors. Model-driven code generation can address this issue by automatically generating code from formal models, but existing approaches. The challenge is to develop code generation techniques that produce correct code while preserving the intended behaviour of the model.

When validation is performed through testing, comprehensive test suites that can exercise all relevant behavioural scenarios are needed. Manual test generation is time-consuming and error-prone, while random test generation often fails to achieve adequate coverage of complex state spaces. Automated test generation from formal models can address this challenge. The challenge is compounded by the need to generate tests that cover both successful execution paths and failure yet realistic scenarios. Test generation must produce test suites that systematically explore many scenarios, ensuring that the contract behaves as expected.

The validation of smart contract correctness requires metrics beyond simple code coverage. Mutation testing [115, 26, 28], which systematically introduces faults into code to assess test suite quality, provides a more rigorous measure of test effectiveness. However, the application of mutation testing to smart contracts is still at an early stage [204, 142, 24, 159], and has been introduced as a service for the development of smart contracts [27] but there is a need for comprehensive validation frameworks that combine multiple testing and analysis techniques. Code coverage metrics alone are insufficient: a test suite may achieve 100% statement coverage while missing critical behavioural scenarios [153]. This limitation arises because statement coverage only indicates which lines of code were executed, not whether they were tested with appropriate inputs, boundary conditions, or

error scenarios. For instance, a test that executes a function with valid inputs achieves coverage but may miss vulnerabilities that only manifest with edge cases or malicious inputs. Mutation testing [115] provides a more rigorous assessment by measuring the test suite's ability to detect faults, systematically evaluating whether tests can distinguish correct code from intentionally introduced errors. Moreover, mutation testing for smart contracts requires domain-specific mutation operators that reflect common vulnerabilities and programming errors in smart contract development.

Finally, there is a lack of unified frameworks that integrate modelling, code generation, test generation, and validation in a coherent methodology. Existing tools and approaches often address only one aspect of the development challenge, requiring developers to integrate multiple tools and methodologies manually. This fragmentation increases the complexity of the development process and reduces its accessibility to practitioners. For example, a developer might use one tool for modelling, another for code generation, a third for test generation, and a fourth for mutation testing. Each tool may have different input formats, different assumptions about the model, and different output formats. This fragmentation creates barriers to adoption and increases the risk of errors in the integration process.

In summary, this thesis addresses the following problems:

i) developing a modelling framework that balances expressiveness with tractability for automated code and test generation;
ii) bridging the implementation gap between formal models and executable code through reliable code generation;
iii) automatically generating comprehensive test suites that systematically explore the behavioural space of smart contracts;
iv) creating a unified framework that integrates modelling, code generation, test generation, and validation in a single coherent methodology.

I.3 Thesis Focus and Contributions

This thesis addresses the challenges outlined above by developing EDAM, a comprehensive framework for modelling, code generation, test generation, and validation of smart contracts using behavioural types. The framework provides a formal model that balances expressiveness with tractability, supporting essential smart contract features including state-dependent behaviour, role-based access control, dynamic participant management, and inter-contract interactions. The formal semantics enable rigorous reasoning about contract behaviour while remaining tractable for automated analysis and code generation.

The framework integrates four key components. First, a code generation engine automatically produces Solidity smart contracts from EDAM specifications, faithfully implementing the formal model and preserving its structure and semantics. The generator handles complex features including role management, state transitions, guard conditions, and inter-contract interactions, producing both correct and maintainable code.

Second, a test generation methodology produces executable test suites directly from EDAM specifications. The approach combines symbolic trace generation using the formal semantics, randomized exploration of the FSM network, and concrete trace derivation through random value assignment and Satisfiability Modulo Theories (SMT) constraint solving. This methodology systematically explores the state space, capturing both successful and failed execution scenarios to achieve high coverage of the contract's behavioural space.

Third, a multi-faceted validation approach combines code coverage analysis, mutation testing with domain-specific operators for smart contracts, and cross-validation with established implementations. This provides quantitative metrics for assessing the quality of generated code and test suites.

Fourth, we evaluate the framework on a diverse benchmark including standard token contracts (Ethereum Request for Comments 20 (Token Standard) (ERC20)), Decentralized Finance (DeFi) protocols (Automated Market Makers (AMMs)), contracts from the Azure repository, and complex multi-contract systems. The evaluation demonstrates that the framework handles a wide range of

smart contract features and produces code and test suites achieving high coverage and mutation scores. A detailed case study involving a complex multi-contract system demonstrates the scalability and practical applicability to real-world scenarios.

The thesis demonstrates that behavioural type systems provide a solid foundation for smart contract development, enabling unified frameworks that integrate modelling, code generation, test generation, and validation. The research methodology combines theoretical development with practical implementation and empirical evaluation, using established techniques from behavioural type theory [111], process calculi [149], compiler construction [4], symbolic execution [121], constraint solving [29], and software testing [153].

How safe is this model-driven approach in practice? The honest answer is: only time and large-scale adoption can fully validate it. Safety is not a claim; it is an empirical and cumulative property that emerges through usage, scrutiny, and iteration. However, early results are encouraging. The framework has been applied to a diverse range of smart contracts, from simple token contracts to complex multi-contract systems, demonstrating its practical applicability. The model-driven approach provides structural advantages that reduce the likelihood of errors: abstraction enables reasoning at the semantic level, early verification catches errors before code generation, and semantically correct code generation eliminates the verification gap. The future of secure smart contracts will not be written line by line. It will be modeled, verified, and generated. This thesis contributes to that future by providing a comprehensive framework that integrates modelling, verification, code generation, and validation in a unified methodology.

I.4 Thesis Outline

This thesis is organised as follows:

Chapter II Background presents a state of the art in smart contract development, covering formal methods, model-driven development, and automated testing techniques. The chapter establishes the theoretical foundations for our work and positions it within the broader context of smart contract research. It reviews existing approaches to modelling, code generation, and validation, identifies gaps in current methodologies, and motivates the need for integrated frameworks that unify these aspects in a single methodology.

Chapter III Modelling smart contracts using EDAM presents the EDAM framework, including its formal syntax, semantics, and modelling capabilities. The chapter explains how EDAM can be used to specify smart contract behaviour, including state-dependent transitions, role-based access control, participant management, and inter-contract interactions. The chapter also discusses the expressiveness of the framework and its relationship to other modelling approaches. The formal semantics are presented using established techniques from process calculi and behavioural type theory, providing a rigorous foundation for code generation and test generation.

Chapter IV Code and test generation from EDAM describes the automated code generation process that transforms EDAM specifications into Solidity smart contracts. The chapter also presents the test generation methodology, which produces comprehensive test suites from EDAM specifications. The test generation process includes symbolic trace generation, concrete trace derivation, and test case generation, with detailed explanations of each stage. The chapter discusses the correctness guarantees provided by the code generation process and the coverage properties of the generated test suites.

Chapter V Validation of smart contract behaviour presents the validation framework, including code coverage analysis, mutation testing, and cross-validation with established implementations. The chapter reports comprehensive experimental results on a diverse benchmark of smart contracts, demonstrating the effectiveness of our approach across different contract types and complexity levels. The evaluation includes both quantitative metrics and qualitative analysis, providing insights into the strengths and limitations of the approach.

Chapter VI Conclusions & Future works summarises the contributions of the thesis, discusses the implications of the results, and outlines directions for future research. The chapter reflects on the achievements and limitations of the current work and identifies opportunities for further

development. It discusses the practical applicability of the framework, the challenges encountered in its development, and the lessons learned from the evaluation.

List of Publications and Forthcoming Work

Published

- **TRAC: A Tool for Data-Aware Coordination (with application to smart contracts)** [3]
  João Afonso, Elvis Konjoh Selabi, Maurizio Murgia, António Ravara, Emilio Tuosto *COORDINATION 2024, Springer-Verlag, 2023*
  Best Artefact Award - COORDINATION 2024
- **Enabling Citizen Sustainable Behaviors in Urban Mobility through Blockchain and Tokenization** [143]
  Silvio Meneguzzo, Elvis Gerardin Konjoh Selabi, Alfredo Favenza, Valentina Gatteschi, Claudio Schifanella, *CRC Press, pp. 245-279, 2023*, *Blockchain and tokenization in urban mobility*
- **Automatic Code and Test Generation of Smart Contracts from Coordination Models**
  Elvis Konjoh Selabi, Maurizio Murgia, António Ravara, Emilio Tuosto
  Accepted for publication in ECOOP 2026
  Artifact Evaluation: Successful with Functional and Reusability badges (ECOOP 2026 artifact evaluation) [123]

Forthcoming Publications

- Ongoing work with Alberto Lluch Lafuente and Andrey Rivkin on model checking of Enhanced Data-Aware Machines (EDAM).
  Status: Ongoing

Chapter II: Background

II.1 Introduction

The concept of smart contracts originates from the seminal work of Nick Szabo in the 1990s [188, 189], who defined them as computer protocols intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. The practical realisation began with Bitcoin [156], which demonstrated the viability of decentralised, trustless systems, followed by Ethereum [44], which introduced a blockchain platform specifically designed to support programmable smart contracts through the Ethereum Virtual Machine (EVM). The EVM's introduction of Turing-completeness (the ability to express any computable function, enabling arbitrary program logic) to blockchain programming created new challenges for verification, as noted by Hirai [108], who provided one of the first formal specifications of the EVM for interactive theorem provers.

This chapter provides an analysis of the state of the art in smart contract modelling, verification, and code and test generation. We begin by establishing the foundational context: blockchains and smart contracts, their execution environment, and the challenges they pose for verification. We then introduce the theoretical foundations from programming language theory, with particular emphasis on behavioural type theory—the core theoretical framework that underlies our approach. Behavioural type systems extend traditional static type systems by capturing not just the structure of data, but the dynamic behaviour of programs, particularly how components interact through communication protocols. This theoretical foundation provides the mathematical basis for specifying and verifying smart contract behaviour.

We examine coordination models based on FSMs and their extensions, showing how they instantiate behavioural type theory in practice. The chapter reviews model-driven development frameworks that generate executable smart contract code from formal specifications, including abstract model specifications, choreography-based modelling, typestate-based languages, and integrated modelling and testing approaches. We then survey verification and testing techniques, and conclude by identifying the gaps in current approaches that motivate the EDAM framework.

II.2 Blockchains and Smart Contracts

II.2.1 Blockchain

A blockchain is a distributed ledger technology that maintains a continuously growing list of records, called blocks, which are linked and secured using cryptography [156, 205]. The fundamental innovation of blockchain lies in its ability to create a tamper-evident, append-only data structure that enables trustless coordination among mutually distrusting parties. Unlike traditional centralized databases where a single authority controls data integrity, blockchains distribute trust across a network of independent nodes, each maintaining a complete copy of the ledger and independently verifying all transactions. This architectural paradigm shift eliminates single points of failure and creates systems that are resistant to censorship and manipulation, even when individual participants may be malicious or compromised.

The blockchain's security derives from cryptographic primitives that provide authentication, integrity, and privacy guarantees. Cryptographic hash functions are one-way mathematical functions that produce a fixed-size output (hash) from arbitrary-sized input data. The critical property of cryptographic hash functions is computational infeasibility: given a hash value, it is computationally infeasible to determine the original input, and any modification to the input produces a completely

different hash. This property enables the creation of a cryptographic chain where each block contains the hash of the previous block, creating an immutable sequence where any modification to historical data would be immediately detectable through hash mismatches. Digital signatures provide authentication and non-repudiation, enabling users to prove transaction authorization without revealing private keys. Most blockchains use elliptic curve cryptography, specifically the Elliptic Curve Digital Signature Algorithm (ECDSA), where each user generates a key pair: a private key (randomly chosen secret) and a public key (derived from the private key through elliptic curve multiplication). The private key signs transactions, while the public key verifies signatures. Bitcoin and Ethereum use the secp256k1 elliptic curve, chosen for its efficiency and security properties. Merkle trees [144] are a way to cryptographically commit to a list of values using hash functions, enabling efficient verification of data integrity and membership proofs. A Merkle tree is a binary tree where leaf nodes contain data (or hashes of data), and each internal node contains the hash of its children. The root node, called the Merkle root, cryptographically commits to all data in the tree: any modification to any leaf node propagates upward through parent nodes, ultimately changing the root hash. This structure enables logarithmic-space proofs of membership: proving that a transaction belongs to a block requires only $\log(n)$ hashes for a block with n transactions, rather than the entire block contents.

Consensus Mechanisms

Consensus are protocols that enable distributed nodes to agree on the state of the blockchain without relying on a trusted central authority. Consensus mechanisms operate in peer-to-peer networks where nodes (computers running blockchain software) participate in consensus and create new blocks. While different node types exist (full nodes that maintain complete blockchain copies, light nodes that store only headers, and archive nodes that maintain historical state), the focus here is on nodes that actively participate in consensus. These nodes validate transactions, execute consensus algorithms, and create blocks containing batches of transactions. Each block includes a state root (derived from Merkle trees). The challenge lies in achieving agreement despite network delays, node failures, and potentially malicious participants. Different consensus algorithms trade off security, performance, and decentralization properties (this is also known as the blockchain trilemma). a) Security: the ability to resist attacks and maintain the integrity of the blockchain b) Performance: the ability to handle a large number of transactions per second c) Decentralization: the ability to distribute the power of the blockchain to many independent nodes. Here we mention a few of the most popular consensus mechanisms.

Proof of Work (PoW) [156], pioneered by Bitcoin, requires nodes (miners) to solve computationally intensive cryptographic puzzles to create valid blocks. The puzzle involves finding a nonce value such that when combined with the block header and hashed, the result is below a target threshold. Since hash functions are unpredictable, miners must try many nonce values through brute-force search, consuming significant computational resources. The difficulty target adjusts periodically to maintain a constant block production rate (approximately every 10 minutes in Bitcoin), ensuring network stability.

Proof of Work (PoW) provides security through economic incentives: attacking the network requires acquiring majority computational power (51% attack), which is economically infeasible for large networks. However, PoW consumes enormous energy: Bitcoin's annual energy consumption rivals that of medium-sized countries. This environmental concern has motivated alternative consensus mechanisms.

Proof of Stake (PoS) [122] selects block validators based on their economic stake (cryptocurrency holdings) rather than computational work. Validators lock cryptocurrency as collateral (staking), and the probability of being selected to create a block is proportional to their stake. Proof of Stake (PoS) eliminates energy-intensive mining while maintaining security through economic penalties: validators who validate invalid blocks risk losing their staked funds (slashing). Ethereum's transition to PoS (the Merge) reduced energy consumption by over 99%, demonstrating the environmental benefits of stake-based consensus.

PoS variants include delegated PoS (where stakeholders vote for delegates who validate blocks),

liquid staking (where staked tokens remain transferable), and sharded PoS (where validators are assigned to specific shards to improve scalability). Each variant addresses different trade-offs between decentralization, security, and performance.

Hybrid consensus mechanisms combine multiple approaches to leverage the strengths of different consensus algorithms while mitigating their weaknesses. For example, Hyperledger Fabric [14] uses a permissioned blockchain architecture where ordering nodes use crash fault-tolerant consensus (such as Raft or Kafka) for transaction ordering, while execution nodes validate transactions independently, combining ordering consensus with execution validation. Tendermint [125] (used in Cosmos) combines PoS validator selection with Byzantine fault-tolerant consensus, where validators are selected based on stake but consensus within the validator set uses a BFT algorithm that provides immediate finality. Polkadot uses a hybrid approach combining nominated proof-of-stake (NPoS) for validator selection with the GRANDPA finality gadget that provides deterministic finality through a BFT-style consensus mechanism [206]. These hybrids attempt to combine the security properties of different mechanisms while mitigating their individual weaknesses, enabling systems that provide both decentralization (through PoS or PoW) and fast finality (through Byzantine Fault Tolerance (BFT) mechanisms).

One of the most popular consensus algorithms for permissioned networks is *Practical Byzantine Fault Tolerance (PBFT)* [47], a consensus algorithm designed for permissioned networks where participants are known and authenticated. PBFT can tolerate up to f Byzantine (arbitrarily malicious) faults among $3f + 1$ total nodes, achieving consensus in three phases: pre-prepare, prepare, and commit. Unlike PoW and PoS, Practical Byzantine Fault Tolerance (PBFT) provides finality (immediate irreversibility) rather than probabilistic finality, making it suitable for enterprise applications requiring guaranteed transaction ordering. Such algorithms are used for instance in the Hyperledger Fabric, Cosmos and Polkadot blockchains.

Block Structure and Composition

The structure of a block depends fundamentally on the blockchain platform, reflecting different architectural choices and design goals. Bitcoin [156] employs a Unspent Transaction Output (UTXO) model where transactions consume previous outputs and create new ones, with blocks containing transactions that reference specific UTXO s. EVM-compatible blockchains [205], such as Ethereum, use an account-based model where blocks contain transactions that modify account balances and contract state, with each account having a persistent address and state. Directed Acyclic Graph (DAG)-based systems [113], such as IOTA's Tangle [170], fundamentally differ by not using blocks at all: transactions directly reference previous transactions, forming a DAG structure that enables parallel processing of transactions. Given the focus of this work on smart contract specification and verification, the following discussion concentrates on EVM-compatible blockchains, particularly Ethereum, as they represent the dominant platform for smart contract development and execution [193, 129, 218].

A fundamental concept in EVM-compatible blockchains is *gas*, a unit that measures the computational cost of executing operations [205]. Gas serves as a metering mechanism that quantifies the resources consumed by each operation (opcode) in the virtual machine, including arithmetic operations, storage reads and writes, cryptographic operations, and inter-contract calls. Each EVM opcode has a fixed gas cost assigned based on its computational complexity: simple operations like addition consume 3 gas, while expensive operations like storage writes consume 20,000 gas, and cryptographic operations like SHA-3 hashing consume 30-60 gas plus additional costs based on input size. The total gas consumed by a transaction is the sum of all operation costs executed during its execution. Gas limits prevent denial-of-service attacks by bounding the computational resources any transaction can consume, while gas prices (paid in the native cryptocurrency) determine transaction priority and compensate validators for executing transactions. Transactions that exceed their gas limit are reverted, ensuring that execution always terminates and network resources remain bounded.

In EVM-compatible blockchains, each block contains several components that work together to ensure integrity, ordering, and verifiability (see Figure II.1). The *block header* contains metadata

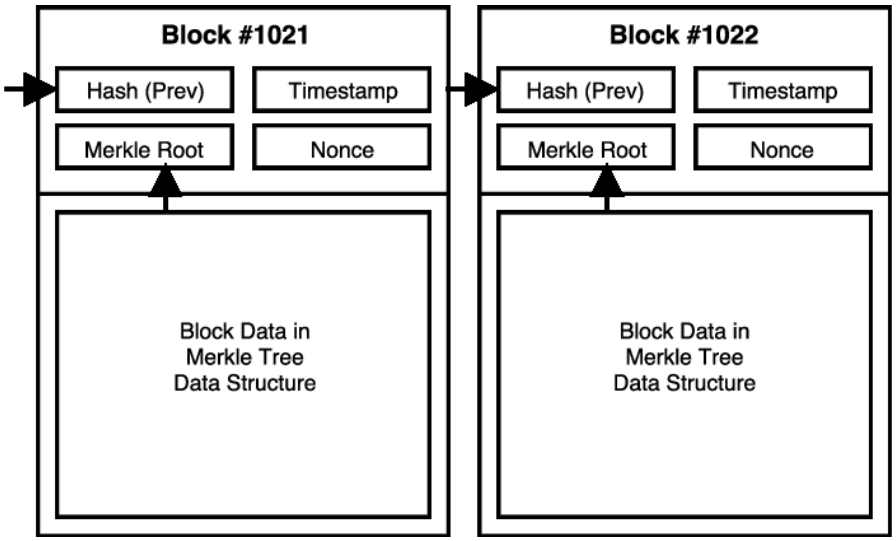


Figure II.1: Structure of a blockchain block [88]

that enables block validation and chain linking. The header typically includes: (i) the *previous block hash*, a 256-bit hash of the previous block's header that creates the cryptographic link in the chain; (ii) the *transactions root*, a Merkle root hash; (iii) the *state root*, a Merkle Patricia Trie root that commits to the complete blockchain state after processing all transactions in the block; (iv) the *receipts root*, a Merkle root of transaction receipts containing execution logs and gas consumption; (v) the *timestamp*, indicating when the block was created, used for difficulty adjustment and preventing timestamp manipulation attacks; (vi) the *nonce*, a random value used in Proof of Work consensus to find a valid block hash (or validator signature in Proof of Stake systems); and (vii) the *difficulty target* or *base fee*, specifying the required hash value threshold for block validity or the base gas price for transactions. Additional fields may include version numbers, block height, gas limit, gas used, and platform-specific metadata.

The *block body* contains the actual transaction data. Transactions are grouped into a Merkle tree structure, where leaf nodes represent individual transactions and internal nodes contain hashes of their children. The root of this tree (the transactions root) is included in the block header, creating a compact cryptographic commitment to all transactions. This structure enables efficient proofs of transaction inclusion: to prove that a specific transaction belongs to a block, one only needs to provide the transaction itself along with the hashes of sibling nodes along the path to the root, rather than the entire block contents. This property, known as *Merkle proofs* [144], enables light clients to verify transaction inclusion without storing the full blockchain.

Block size limits in EVM-compatible blockchains are typically expressed in terms of gas limits rather than byte size. Ethereum does not enforce a strict byte limit but uses gas limits (currently around 30 million gas per block) to constrain block complexity and execution time. This approach balances throughput, decentralization (larger gas limits require more computational resources to process, potentially excluding smaller nodes), and network propagation time (larger blocks take longer to transmit across the network, increasing the risk of forks). In contrast, Bitcoin limits blocks to approximately 1MB (expanded to 4MB with SegWit [131]), focusing on transaction count rather than computational complexity.

Transactions encode state transitions such as value transfers, contract deployments, or function calls [205]. An *address* is a 160-bit (20-byte) identifier representing an account. Two account types exist: *Externally Owned Accounts* (EOAs), controlled by private keys with addresses derived from the last 20 bytes of the Keccak-256 hash of the public key; and *Contract Accounts*, with addresses deterministically computed during deployment as the last 20 bytes of the Keccak-256 hash of the deployer's address and nonce. Addresses are represented as hexadecimal strings prefixed with "0x" (e.g., `0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb`). Transaction fields include [205]: (i) *from* and *to* addresses; (ii) *value*, the native cryptocurrency amount; (iii) *data*, containing function

call parameters or contract bytecode; (iv) *gas limit* and *gas price*; (v) *nonce*, a sequence number preventing replay attacks; and (vi) *signature*, cryptographic proof of authorization.

The transaction lifecycle proceeds through: creation and cryptographic signing; broadcasting to the peer-to-peer network via gossip protocol [156]; validation by nodes (signature verification, balance checks, nonce sequencing); entry into the *mempool* (a local data structure maintained by each node containing transactions that are candidates for inclusion in the next block); selection by block producers prioritizing higher gas prices; and deterministic execution upon block inclusion, updating global state consistently across the network [205].

Security Properties and Attack Vectors

Blockchain security relies on cryptographic assumptions and economic incentives. The fundamental security property is *immutability*: once transactions are confirmed in sufficiently deep blocks, reversing them requires controlling majority consensus power, which becomes economically infeasible as block depth increases. This property enables trustless value transfer: users can accept payments with confidence that they cannot be reversed after confirmation.

Double-spending attacks [156] attempt to spend the same cryptocurrency twice by creating conflicting transactions. PoW prevents this through the longest chain rule: nodes accept the chain with most accumulated work, making it computationally infeasible for attackers to create longer alternative chains unless they control majority hash power. PoS prevents double-spending through slashing penalties: validators who sign conflicting blocks lose their staked funds, making attacks economically irrational.

Selfish mining attacks [79] exploit the longest chain rule in PoW systems to gain disproportionate rewards without controlling majority hash power. A selfish miner with hash power $\alpha < 0.5$ strategically withholds newly mined blocks instead of immediately broadcasting them, creating a private chain. When honest miners discover a block, the selfish miner releases blocks from their private chain to create a fork, forcing honest miners to abandon their work and switch to the selfish miner's chain. This strategy wastes honest miners' computational resources while allowing the selfish miner to capture a larger share of block rewards than their hash power would normally entitle them to. The attack becomes profitable when $\alpha > \frac{1}{3}$, meaning a miner with only 33% hash power can gain more than their fair share of rewards. Selfish mining demonstrates that the honest mining strategy (immediately broadcasting blocks) is not always optimal, challenging the assumption that miners always act in the network's best interest. Countermeasures include detecting and penalizing block withholding, implementing uncle block rewards (as in Ethereum) to reduce the advantage of withholding, and using alternative consensus mechanisms that are less susceptible to strategic manipulation.

51% attacks [175] occur when a single entity controls majority consensus power (PoW hash power or PoS stake). Such attackers can exclude transactions, reverse recent transactions, and prevent other miners/validators from creating blocks. However, these attacks are temporary: once the attacker's power drops below 50%, the honest chain will outpace the attacker's chain. Additionally, 51% attacks are expensive and visible, as they require acquiring majority resources and produce obvious chain reorganizations.

Sybil attacks [73] involve creating many fake identities to gain disproportionate influence. PoW naturally resists Sybil attacks: creating identities is free, but acquiring hash power is expensive. PoS systems may be more vulnerable if stake distribution is concentrated, though many PoS designs include mechanisms to prevent single entities from controlling excessive stake.

Eclipse attacks [105] isolate target nodes by controlling all their peer connections, feeding them false blockchain data. Nodes defend against eclipse attacks by maintaining diverse peer connections, using authenticated peer discovery, and validating all received data independently rather than trusting peers.

The security of blockchain networks ultimately depends on the assumption that honest participants control majority resources. It is commonly believed that this assumption holds for mature networks where consensus power is distributed among many independent entities, but may be violated in small networks or during early bootstrapping phases when a few entities may control

significant resources.

II.2.2 Smart Contracts

Smart contracts are self-executing programs deployed on blockchain platforms that automatically execute predefined actions when specific conditions are met [188]. The concept was first proposed by Nick Szabo in 1996, envisioning contracts encoded as computer protocols that execute automatically without requiring human intervention or trusted intermediaries. Unlike traditional contracts that rely on legal systems and intermediaries for enforcement [199], smart contracts encode the terms of an agreement directly into executable code, which runs deterministically on the blockchain network. This fundamental shift from legal enforcement to cryptographic and algorithmic enforcement enables trustless coordination among mutually distrusting parties, eliminating the need for centralized authorities or trusted third parties.

The practical realization of smart contracts began with Bitcoin's scripting language [156], which represents the first deployed implementation of smart contracts on a blockchain. Bitcoin scripts are indeed smart contracts, albeit intentionally limited in their expressiveness. The Bitcoin scripting language (Script) enables conditional transfers and multi-signature transactions, allowing users to specify conditions under which bitcoins can be spent. Script supports operations such as signature verification, time locks, hash preimage verification, and simple arithmetic, enabling use cases such as escrow services, time-locked transactions, and multi-signature wallets. Many extensions to the Bitcoin scripting language have been proposed, such as covenants [154], which allow transactions to restrict how the value they transfer is used in the future. Script is not Turing-complete (it lacks loops), which prevents denial-of-service attacks but limits expressiveness.

Ethereum's introduction of a Turing-complete virtual machine [44] marked a paradigm shift, enabling arbitrary program logic to be executed on the blockchain. This Turing-completeness, combined with persistent state storage and inter-contract communication capabilities, transformed blockchains from simple value transfer systems into general-purpose distributed computing platforms capable of hosting complex decentralized applications. The key difference between Bitcoin scripts and Ethereum smart contracts lies in their computational model: Bitcoin scripts are stateless and execute only during transaction validation, while Ethereum contracts maintain persistent state and can execute complex logic across multiple transactions. This fundamental difference enables Ethereum contracts to implement sophisticated protocols such as decentralized exchanges, lending platforms, and governance mechanisms, which would be impossible with Bitcoin's limited scripting capabilities.

Smart contracts undergo a lifecycle from development to execution. Developers write source code in high-level languages, which is compiled to bytecode understood by the virtual machine. Deployment occurs through a special transaction with null recipient address, containing the contract's bytecode in the data field. The deployment process executes a constructor function (if present) to initialize state variables. Once deployed, contract bytecode becomes immutable, ensuring predictable and verifiable behaviour. Interaction with deployed contracts occurs through transactions specifying the contract address and function call data. Function calls use Application Binary Interface (ABI) encoding to specify which function to invoke and its parameters. The virtual machine loads the contract's bytecode, decodes the function call, executes the corresponding logic, and updates contract state. Contracts can receive native cryptocurrency transfers alongside function calls. Fallback functions or receive functions execute when contracts receive value without a matching function signature, enabling contracts to act as wallets or payment recipients.

Smart contracts execute within isolated virtual machine environments. Isolation prevents contracts from accessing each other's storage or execution context directly; contracts interact only through well-defined interfaces: function calls, value transfers, and event emissions. Virtual machines operate as stack-based machines, where operations consume values from a stack and push results back. The execution environment provides three storage types: (i) the *stack*, used for temporary values during computation; (ii) *memory*, temporary storage cleared between transactions, cheap but ephemeral; and (iii) *storage*, persistent key-value storage that survives across transactions, expensive but permanent. State variables declared in contracts are stored on the blockchain through

storage mechanisms implemented as key-value stores, where keys and values are 256-bit words. Contract state variables map to storage slots through compiler-determined layouts, with each variable occupying one or more slots depending on its type. Contracts emit *events*, structured logs written to the blockchain that serve multiple purposes: enabling communication with off-chain applications, facilitating efficient querying of contract activity without scanning all transactions, and creating audit trails of operations. Events are indexed, allowing efficient filtering and searching by event type and parameter values. Unlike storage, events are append-only and immutable once emitted. Events enable asynchronous communication between contracts and off-chain systems [155], though they are not reliable communication mechanisms: there is no guarantee that events will be observed, and contracts cannot directly respond to events from other contracts.

Smart contracts interact through synchronous function calls, enabling composition of complex decentralized applications from simpler components. This composability allows developers to build on existing contracts, creating abstraction layers and reusing tested code. Inter-contract calls transfer control to the called contract, which executes and returns a result to the caller. The calling contract waits for the result before continuing, enabling contracts to query each other's state and coordinate operations. Synchronous calls create a call stack: if contract A calls contract B, which calls contract C, execution proceeds sequentially through the call chain. The call mechanism introduces security considerations. When contract A calls contract B, control transfers to B's code, which may be malicious or buggy. Contract B can call back into contract A (a *reentrancy attack*), potentially exploiting vulnerabilities if A's state has not been updated before the external call. To mitigate reentrancy, contracts must follow the checks-effects-interactions pattern: perform all validation checks, update internal state, and only then make external calls. Inter-contract calls require the caller to provide sufficient gas for the callee's execution; if the callee runs out of gas, execution reverts, potentially causing the caller's transaction to fail.

Smart contract execution exhibits strong atomicity guarantees: either all operations in a transaction succeed, or the entire transaction reverts as if it never occurred. This all-or-nothing semantics ensures the blockchain state never enters inconsistent intermediate states, even if execution fails partway through. Atomicity applies to the entire transaction, including all inter-contract calls: if any contract in a call chain reverts (explicitly or due to an error), the entire transaction reverts, and all state changes roll back. This enables complex multi-contract operations to be treated as single atomic units. Reverts can occur due to explicit revert statements, assertion failures, out-of-gas conditions, or invalid operations. When a revert occurs, execution halts immediately, and the virtual machine discards all state changes made during the transaction. The transaction sender still pays for gas consumed up to the point of failure, preventing denial-of-service attacks where malicious contracts consume resources without completing execution.

Security Properties and Challenges

Smart contracts operate in an adversarial environment where attackers actively seek to exploit vulnerabilities for financial gain [133, 120]. The immutable nature of deployed contracts means that vulnerabilities cannot be patched, making pre-deployment security critical. The combination of financial incentives, immutability, and public code creates a unique security landscape where vulnerabilities can have severe consequences.

Several classes of vulnerabilities are common in smart contracts. *Reentrancy* vulnerabilities occur when contracts make external calls before updating state, allowing attackers to recursively call back into the contract and exploit intermediate states. *Integer overflow/underflow* vulnerabilities arise when arithmetic operations exceed type bounds, potentially causing incorrect calculations. *Access control* vulnerabilities occur when contracts fail to properly restrict function access, allowing unauthorized users to perform privileged operations.

Front-running attacks [216, 45, 211] exploit the public nature of pending transactions: attackers observe transactions in the mempool and submit their own transactions with higher gas prices to execute first, potentially profiting from information asymmetry [198, 217, 136, 215]. *Timestamp dependence* vulnerabilities occur when contracts rely on block timestamps for critical logic, which miners can manipulate within small bounds.

Denial of service attacks [181] can target contracts that iterate over unbounded data structures or perform expensive computations. If gas limits are insufficient or contracts fail to bound iteration, attackers can cause transactions to fail or consume excessive resources. Samreen and Alalfi [177] propose a smartscan approach to detect denial of service vulnerabilities in Ethereum smart contracts.

The security of smart contracts depends on multiple factors: correct implementation of business logic, proper handling of edge cases, secure interaction with external contracts, and appropriate access control mechanisms. Formal verification, comprehensive testing, and security audits are essential for ensuring contract security, though they cannot guarantee absolute correctness [70].

Use Cases and Applications

Smart contracts enable a wide range of applications that benefit from trustless, automated execution. *DeFi* protocols use smart contracts to implement financial instruments without traditional intermediaries: lending platforms, decentralized exchanges, stablecoins, and derivatives markets all rely on smart contract automation to manage assets, enforce rules, and coordinate participants.

Non-Fungible Token (NFT) s use smart contracts to represent unique digital assets, enabling ownership, transfer, and provenance tracking without centralized registries. Non-Fungible Token (NFT) contracts implement standards for metadata, royalties, and transfer mechanisms, creating interoperable ecosystems of digital collectibles and assets.

Governance mechanisms use smart contracts to implement decentralized decision-making processes. Token holders vote on proposals, with votes recorded on-chain and outcomes automatically executed. This enables Decentralized Autonomous Organization (DAO) s to coordinate without central authorities, though governance attacks and voter apathy remain challenges.

Supply chain applications use smart contracts to track goods through complex supply chains, recording transfers and verifying authenticity. Contracts can enforce business rules, trigger payments upon delivery, and provide transparent audit trails.

Identity and credentials systems use smart contracts to manage decentralized identity, enabling users to control their credentials without relying on centralized identity providers. Formal verification of Internet of Things (IoT) access control via blockchain-enhanced smart contracts is explored in [99]. Contracts can verify credentials, manage attestations, and enable selective disclosure of identity information.

The versatility of smart contracts stems from their ability to encode arbitrary logic while maintaining trustless execution guarantees. However, this versatility also introduces complexity: contracts implementing complex protocols must carefully manage state, handle edge cases, and coordinate with other contracts, increasing the potential for vulnerabilities.

II.2.3 Solidity

Solidity is the most widely used high-level programming language for developing smart contracts on Ethereum and other EVM-compatible blockchains [16]. Developed by the Ethereum Foundation, Solidity was designed specifically for writing smart contracts, combining features from JavaScript, Python, and C++ to create a statically-typed, contract-oriented language. Since its introduction in 2014, Solidity has evolved through multiple major versions, with each version introducing new features, security improvements, and breaking changes that require careful migration strategies for existing contracts. The language's design philosophy emphasizes expressiveness and developer productivity. Solidity's static type system catches many errors at compile time, while its contract-oriented paradigm encourages encapsulation and modular design. However, the language must balance expressiveness with security, as vulnerabilities in smart contracts can have severe financial consequences due to their immutable nature and direct access to digital assets [20]. Solidity programs are organised around *contracts*, which are similar to classes in object-oriented languages but with blockchain-specific semantics. Contracts define state variables, functions, events, and modifiers that collectively implement the contract's behaviour. The compilation process transforms Solidity source code into EVM bytecode [205] through several stages: lexical analysis, parsing, semantic analysis, optimisation, and code generation. The Solidity compiler (solc) performs extensive static analysis during compilation, including type checking, control flow analysis, and gas optimisation. However,

compiler versions can produce different bytecode for the same source code due to optimisation improvements or bug fixes, making version management critical for reproducible builds. The compiler also generates metadata files containing the ABI, which describes function signatures and data structures for external interaction.

The `pragma` directive specifies compiler version constraints (semantic versioning). Solidity's type system includes primitives (uint256/int256 are gas-efficient for the EVM's 256-bit word), composite types (arrays, mappings, structs, enums), and reference types (`storage`, `memory`, `calldata`) that control data location—storage is expensive, memory is temporary. Visibility modifiers (`public`, `private`, `internal`, `external`) and state mutability (`view`, `pure`) enable compile-time optimisations. Multiple inheritance uses C3 linearization Method Resolution Order (MRO); modifiers inject reusable logic (e.g. the `Ownable` pattern). Libraries deploy once and are reused across contracts, executing in the caller's context.

Error Handling and Revert Mechanisms

Solidity provides several mechanisms for error handling and transaction reversion. The `require` statement checks conditions and reverts with a custom error message if the condition fails. The `assert` statement is used for internal consistency checks and consumes all remaining gas on failure (unlike require, which refunds unused gas). The `revert` statement unconditionally reverts execution, optionally with an error message or custom error type.

```
function withdraw(uint256 amount) public {
    require(balances[msg.sender] >= amount + 100, "Insufficient balance");
    balances[msg.sender] -= amount;
    (bool success, ) = msg.sender.call{value: amount}("");
    assert(balances[msg.sender] >= 100);
}
```

`assert(balances[msg.sender] >= 100);` requires the balance of the sender to be always greater than 100 after the withdrawal. This mechanism can be used to enforce invariants in the contract. Custom errors, introduced in Solidity 0.8.4, provide gas-efficient error reporting by encoding error signatures rather than error messages. Custom errors enable more informative error handling while reducing gas costs compared to string-based error messages.

Inter-Contract Calls and Interfaces

Solidity provides multiple mechanisms for contracts to interact with each other, each with different semantics and security implications. *Direct function calls* use the contract's interface to invoke functions, while *low-level calls* provide more control but require manual ABI encoding and decoding.

Interfaces define function signatures without implementation, enabling contracts to interact with each other through well-defined contracts. Interfaces enable type checking and provide clear contracts for inter-contract communication.

```
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
}
contract TokenHolder {
    IERC20 public token;
    function deposit(uint256 amount) external { token.transfer(address(this),
        amount); }
}
```

Interfaces enable contracts to interact with any contract implementing the interface, providing flexibility and enabling protocol composability. The `external` keyword in interfaces indicates that functions are called from outside the contract.

Call types include `call`, `delegatecall`, and `staticcall`. The `call` operation transfers control to another contract, executing its code in its own context. The `delegatecall` operation executes another contract's code in the calling contract's context, preserving storage and state. The `staticcall` operation is like call but prevents state modifications, enabling safe view operations.

```
otherContract.someFunction(param);  // Direct call
(bool ok, ) = addr.call(abi.encodeWithSignature("fn(uint256)", p));  // Low-level
```

```
(bool ok, ) = addr.delegatecall(abi.encodeWithSignature("fn(uint256)", p));  // Delegatecall
```

Low-level calls provide fine-grained control but require manual ABI encoding and careful error handling. They enable interaction with contracts that don't have Solidity interfaces or contracts with unknown function signatures, though they sacrifice type safety and require careful validation.

Fallback and Receive Functions

Fallback functions are a fundamental mechanism in Solidity that enable contracts to handle calls that do not match any defined function signature, as well as to receive Ether (Ethereum's native cryptocurrency) transfers [205]. Solidity provides two special functions for this purpose: the `receive` function and the `fallback` function, both introduced in Solidity 0.6.0 to provide clearer semantics for value transfers and function calls.

The `receive` function is executed when a contract receives Ether via a plain transfer (e.g., `send`, `transfer`, or a plain `call` with value but no data). This function must be declared as `external payable` and cannot have parameters or return values. The `fallback` function is executed when a contract receives a call that does not match any function signature, or when a call includes data but no matching function exists. The fallback function can be declared as `external payable` (to receive Ether) or `external` (non-payable), and it can access `msg.data` to inspect the call data.

```
contract Wallet {
    mapping(address => uint256) public balances;
    receive() external payable { balances[msg.sender] += msg.value; }
    fallback() external payable { revert("Function not found"); }
}
```

The execution order is: (i) if the call has data and matches a function signature, that function executes; (ii) if the call has no data but includes value, the `receive` function executes (if present); (iii) if the call has data but no matching function, the `fallback` function executes (if present); (iv) otherwise, the call reverts.

Fallback functions are essential for several use cases. *Proxy patterns* use fallback functions to forward calls to implementation contracts, enabling upgradability while preserving the proxy's address. *Wallet contracts* use receive functions to accept Ether deposits without requiring specific function calls. *Multi-sig wallets* and *payment channels* rely on fallback functions to handle arbitrary calls or value transfers.

```
contract SimpleProxy {
    address implementation;
    fallback() external payable {
        assembly {
            calldatacopy(0, 0, calldatasize())
            let r := delegatecall(gas(), sload(0), 0, calldatasize(), 0, 0)
            if iszero(r) { revert(0, returndatasize()) }
            return(0, returndatasize())
        }
    }
    receive() external payable {}
}
```

Listing 1 shows a minimal upgradeable proxy that stores an implementation address and forwards all unmatched calls to it via `delegatecall` in the `fallback` function, executing the implementation's code while preserving the proxy's storage. The `receive` function allows the proxy to accept plain Ether transfers, which can then be handled or forwarded by the implementation logic.

Security considerations for fallback functions include ensuring that contracts can handle unexpected Ether transfers, preventing reentrancy attacks when fallback functions interact with external contracts, and avoiding gas limit vulnerabilities when fallback functions perform expensive operations. Contracts should explicitly define receive and fallback functions rather than relying on implicit behaviour, as the absence of these functions causes calls to revert, which may be unintended for contracts designed to receive Ether.

In Solidity versions prior to 0.6.0, a single unnamed function served both purposes, leading to ambiguity about when it would execute. The separation into `receive` and `fallback` functions

provides clearer semantics and better type safety, enabling developers to explicitly handle value transfers versus unmatched function calls.

Security Vulnerabilities and Common Pitfalls

Solidity's design and the EVM's execution model introduce several classes of security vulnerabilities that have led to substantial financial losses [20, 138, 100, 172, 219]. Understanding these vulnerabilities is essential for writing secure contracts.

Reentrancy attacks occur when external calls allow malicious contracts to recursively call back into the original contract before state updates complete. This vulnerability famously led to the DAO hack, where an attacker drained millions of dollars by recursively calling a withdrawal function.

```
// Vulnerable: state update after external call
function withdraw() public {
    uint256 amt = balances[msg.sender];
    msg.sender.call{value: amt}("");  // Reentrancy point
    balances[msg.sender] = 0;  // Too late!
}
// Secure: checks-effects-interactions
function withdraw() public {
    uint256 amt = balances[msg.sender];
    balances[msg.sender] = 0;  // State first
    msg.sender.call{value: amt}("");
}
```

The secure version follows the *checks-effects-interactions* pattern: validate inputs, update state, then make external calls. Reentrancy guards can also prevent recursive calls, though they must be used carefully to avoid blocking legitimate patterns.

Integer overflow and underflow vulnerabilities arise when arithmetic operations exceed type bounds. Solidity 0.8.0+ includes built-in overflow checks that automatically revert on arithmetic errors, but older versions require explicit checks using libraries like SafeMath.

Access control vulnerabilities occur when functions lack proper authorization checks, allowing unauthorized users to execute privileged operations. These vulnerabilities are among the most critical in smart contracts, as they can enable attackers to drain funds, modify critical state, or bypass security mechanisms. Common patterns include missing access control modifiers, incorrect role-based checks, flawed ownership transfer mechanisms, and insufficient validation of caller permissions.

```
// Vulnerable: no access control
function changeOwner(address newOwner) public { owner = newOwner; }

// Secure: modifier enforces authorization
modifier onlyOwner() { require(msg.sender == owner, "Not authorized"); _; }
function changeOwner(address newOwner) public onlyOwner { owner = newOwner; }
```

Access control vulnerabilities can also arise from incorrect role-based access Role-Based Access Control (RBAC) implementations. Contracts using role-based systems must carefully validate role assignments, ensure role revocation works correctly, and prevent role escalation attacks. Additionally, multi-signature wallets and governance contracts require careful implementation to prevent unauthorized actions while maintaining flexibility for legitimate operations.

Unchecked external calls can lead to silent failures when called contracts revert. The return value of external calls must be checked, as failed calls do not automatically propagate reverts in all calling contexts.

```
someContract.someFunction();  // Vulnerable: unchecked return
(bool ok, ) = addr.call(abi.encodeWithSignature("someFunction()"));
require(ok, "Call failed");  // Secure
```

Timestamp dependence vulnerabilities occur when contracts rely on block timestamps for critical logic. Miners can manipulate timestamps within a small window (typically ±15 seconds), making timestamp-based logic unreliable for security-critical operations.

Delegatecall vulnerabilities arise from the `delegatecall` operation, which executes code from another contract in the calling contract's context. Delegatecall preserves the calling contract's

storage and state, enabling proxy patterns but also creating risks if the delegatecalled contract is malicious or buggy.

```
// Vulnerable: anyone can delegatecall arbitrary implementation
function delegateCall(bytes memory data) public {
    implementation.delegatecall(data);  // No access control
}
// MaliciousContract.owner at slot 0 = VulnerableProxy.owner
function attack() public { owner = msg.sender; }  // Steals ownership
```

The vulnerability in the example above demonstrates how delegatecall executes code in the calling contract's storage context. When 'MaliciousContract.attack()' is delegatecalled, it modifies storage slot 0 (the 'owner' variable), which corresponds to 'VulnerableProxy.owner', effectively transferring ownership to the attacker. This occurs because delegatecall preserves the storage layout of the calling contract, making storage slot collisions a critical concern in proxy patterns.

Storage layout collisions occur in proxy patterns where storage variable layouts must remain consistent across upgrades. Changing storage layouts can corrupt data or create security vulnerabilities, requiring careful versioning and migration strategies.

Gas limit vulnerabilities arise when loops or recursive calls consume excessive gas, potentially causing transactions to fail or enabling denial-of-service attacks. Contracts must bound iteration and avoid unbounded loops that could exhaust gas limits.

Development Tools and Ecosystem

The Solidity ecosystem includes numerous development tools that facilitate contract development, testing, and deployment. The Remix IDE [85] provides a browser-based development environment with integrated compilation, debugging, and deployment capabilities, enabling developers to write, test, and deploy contracts directly from a web browser without local setup. Hardhat [86] and Truffle [185] are popular development frameworks that provide comprehensive testing infrastructure, deployment scripts, and integration with development networks, offering features such as automated testing, network management, and contract interaction utilities.

Static analysis tools like Slither [81], Mythril [71], Securify [194], SmartCheck [191], and ZEUS [119] identify common vulnerabilities and security issues through automated code scanning. Slither performs static analysis on Solidity source code to detect vulnerabilities, while Mythril uses symbolic execution on EVM bytecode to find security flaws. Securify employs security patterns and data-flow analysis to identify potential exploits. Formal verification tools like Certora [49] and K framework [176] enable mathematical proofs of contract correctness through specification-based verification. Certora provides automated formal verification with user-defined specifications, while the K framework offers a rewriting-based semantic framework for defining and verifying programming languages; it has been applied to Ethereum in KEVM [107], a complete formal semantics of the EVM. Chinen et al. [52] introduced RA, which statically analyzes contracts for re-entrancy vulnerabilities using symbolic execution and SMT-based equivalence checking. These tools complement manual code review and testing, though they cannot guarantee absolute security due to the complexity of smart contract interactions and the inherent limitations of automated analysis.

The immutable nature of deployed contracts means that vulnerabilities cannot be patched after deployment, making comprehensive testing, security audits, and formal analysis essential before deployment. To address these verification challenges, we turn to behavioural type theory, which provides a rigorous foundation for specifying and verifying distributed system behaviour. This theoretical framework enables us to reason about smart contracts not just as programs, but as behavioural protocols that coordinate multiple participants through well-defined interaction patterns.

II.3 Theoretical Foundations for Formal Modeling

Having established the context of smart contracts and their verification challenges, we now introduce the theoretical foundations that enable formal modelling and analysis. This section covers the fundamental concepts from programming language theory, with particular emphasis on behavioural

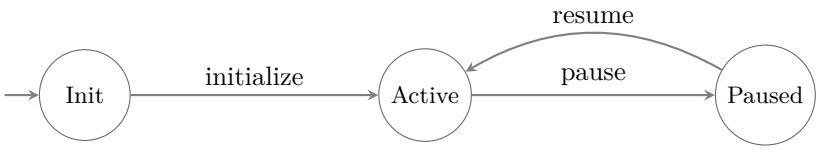
type theory, which provides the theoretical basis for our framework. We begin with the basic structures (Labelled Transition Systems (LTSs) and operational semantics) that underlie formal modelling, then introduce behavioural types and typestates, which extend these foundations to capture the dynamic behaviour of distributed systems.

II.3.1 Labelled Transition Systems

A LTS is a mathematical model used to describe the behaviour of concurrent and reactive systems [149]. LTSs are fundamental to process calculi and formal verification, providing a framework for modelling and analysing system behaviour. An LTS consists of a set of states, a set of labels (representing actions or events), and a transition relation that specifies how the system moves from one state to another when certain actions occur. Formally, an LTS is a triple $(S, A, \rightarrow)$ where S is the set of states, A is the set of action labels, and $\rightarrow \subseteq S \times A \times S$ is the transition relation. A transition $(s, a, s') \in \rightarrow$ is typically written as $s \xrightarrow{a} s'$, indicating that the system can move from state s to state s' by performing action a. LTSs are closely related to other formal models such as Petri nets [167, 187], which model concurrent systems using places, transitions, and tokens.

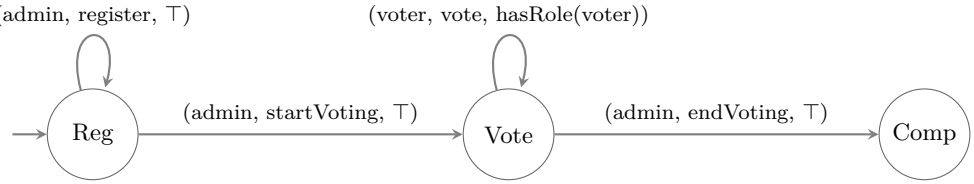
An LTS typically includes an initial state $s_0 \in S$ from which execution begins. The behaviour of the system is captured by the set of all possible execution traces, which are sequences of states and actions starting from the initial state. A *trace* is a finite or infinite sequence $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots$ where each step represents a valid transition. A state s is *reachable* if there exists a trace from s_0 to s. In a labelled transition system, a *deadlock* is a state with no possible outgoing transitions in the relevant transition relation, meaning the system cannot make progress from that state. These concepts are in important for analysing smart contracts, where deadlocks represent situations where the contract becomes unresponsive, and reachability analysis helps identify which states can be reached during execution.

Consider a simple example of an LTS modelling a token contract with three states: *Init*, *Active*, and *Paused*. The transitions are shown in the following transition diagram:



Starting from *Init*, the system can transition to *Active* via the *initialize* action. From *Active*, it can move to *Paused*, and from *Paused* back to *Active*. This simple LTS captures the essential state machine behaviour of a pausable token contract.

LTSs can model more complex scenarios by enriching labels with additional information. For instance, transitions can be labelled with tuples (p, op, ϕ) where p is the participant performing the action, op is the operation name, and ϕ is a guard condition that must be satisfied. Consider a voting contract where only participants with the *Voter* role can cast votes, shown in the following transition diagram, where state *Reg* represents the registration state, state *Vote* represents the voting state, and state *Comp* represents the completed state.



Here, $\top$ represents a condition that is always true, while hasRole(Voter) is a guard that checks role membership. The labels encode both the actor and the conditions under which transitions are enabled, enabling precise modelling of access control policies.

LTSs can be extended with additional structure to capture more complex behaviour. For instance, *Kripke structures* [56] augment TSs with atomic propositions that label states with

properties, enabling temporal logic verification. *Timed automata* [11] extend LTSs with clocks and timing constraints, allowing modelling of real-time systems. *Data-aware* LTSs extend the basic model with variables and data-dependent transitions, where guards and assignments can reference and modify variable values. For smart contracts, which operate in a discrete, event-driven environment with persistent state, data-aware LTSs provide the expressiveness needed to capture both control flow and data manipulation.

Bisimulation relations provide a way to compare LTSs for behavioural equivalence. Strong bisimulation, the first such notion, was introduced by Park [165]; several variants have since been developed [149, 109, 106]. Two states s_1 and s_2 are strongly bisimilar if they can mimic each other's behaviour: for every transition from s_1 to s_1', there exists a corresponding transition from s_2 to s_2' with the same label leading to bisimilar states, and vice versa. Bisimulation can be used to prove that two contracts or contract implementations exhibit equivalent behaviour, which is crucial for ensuring that code generation preserves the semantics of formal models.

LTSs provide a natural way to model the behaviour of smart contracts, where states represent the contract's configuration (variable values, current state in a state machine), and transitions represent operations or function calls that modify the contract's state. The labels on transitions can encode rich information, including the participant performing the action, the parameters passed, and the conditions that must be satisfied for the transition to occur. This makes LTSs particularly suitable for modelling role-based access control, where transitions are only enabled when executed by authorized participants. The formal foundation provided by LTSs enables rigorous analysis of contract behaviour, including verification of safety properties (nothing bad happens) and liveness properties (something good eventually happens).

II.3.2 Operational Semantics

Operational semantics is a formal method for defining the meaning of programming languages by describing how programs execute step by step [145]. Unlike denotational semantics, which assigns mathematical meanings to program constructs, operational semantics focuses on the computational process itself, specifying how a program transforms from one configuration to another during execution. This approach is particularly well-suited for reasoning about program behaviour, as it directly models the execution process that occurs in practice [149].

There are two main approaches to operational semantics: *small-step* and *big-step* semantics. Small-step semantics (also called structural operational semantics [145]) describes the execution as a sequence of individual computation steps, where each step represents a single atomic operation. This approach is particularly useful for modelling intermediate states and reasoning about partial computations, concurrency, and non-terminating programs. Big-step semantics (also called natural semantics) describes the execution as a single step from an initial configuration to a final result, abstracting away intermediate states. This approach is more suitable when only the final outcome matters, and is often more concise for defining the semantics of expressions and simple statements.

Operational semantics is typically specified using *inference rules* [145], which define how to derive transitions between configurations. These rules have the form of logical implications: if certain premises hold, then a conclusion about a transition can be drawn. Inference rules can be *axioms* (base cases that always hold) or *derived rules* (rules that depend on other rules). This rule-based approach provides a modular and compositional way to define semantics, where complex constructs are built from simpler ones [149].

Inference rules are typically written in the following format:

$$\frac{\text{premise}_1 \quad \text{premise}_2 \quad \dots \quad \text{premise}_n}{\text{conclusion}}$$

The horizontal line separates premises (above) from the conclusion (below). If there are no premises, the rule is an axiom and is written without the horizontal line.

Consider a simple example of operational semantics for a token contract's transfer operation. A configuration $\langle q, \rho, \sigma \rangle$ consists of the current state q, role assignment function ρ, and environment mapping σ (mapping variables to values). The transfer operation moves tokens from one account

to another, but only if the sender has sufficient balance and the contract is in the *Active* state. The operational semantics rule for transfer can be written as:

$$\frac{\sigma(\text{sender}) \geq \text{amount} \quad q = \mathit{Active}}{\langle q, \rho, \sigma \rangle \xrightarrow{\text{transfer(caller,to,amount)}} \langle q, \rho, \sigma[\text{sender} \mapsto \sigma(\text{sender}) - \text{amount}, \text{to} \mapsto \sigma(\text{to}) + \text{amount}] \rangle}$$

This rule states that a transfer transition is enabled when: (i) the sender's balance is sufficient, and (ii) the contract is in the *Active* state. When these conditions hold, the transition updates the environment by subtracting amount from the sender's balance and adding it to the recipient's balance, while preserving the state and role assignment.

As another example, consider a pause operation that requires the Admin role. This operation transitions the contract from *Active* to *Paused* state, but only if the caller has administrative privileges:

$$\frac{\rho(\text{caller}, \text{Admin}) \quad q = \mathit{Active}}{\langle q, \rho, \sigma \rangle \xrightarrow{\text{pause(caller)}} \langle \text{Paused}, \rho, \sigma \rangle}$$

This rule demonstrates role-based access control: the pause operation is only enabled when the caller has the Admin role and the contract is in the Active state. The transition changes the state from *Active* to *Paused* while preserving the role assignment and environment.

A *derivation* shows how inference rules are applied step by step to derive a transition. Consider deriving the execution of a sequence of operations. Suppose we have configurations c_0, c_1, and c_2, and we want to show that c_0 can transition to c_2 through two steps:

$$\frac{\text{premises for op}_1}{c_0 \xrightarrow{\text{op}_1} c_1} \qquad \frac{\text{premises for op}_2}{c_1 \xrightarrow{\text{op}_2} c_2}$$

If both rules can be applied (their premises hold), we can derive the composite transition $c_0 \xrightarrow{\text{op}_1;\text{op}_2} c_2$ through sequential composition. This derivation demonstrates how complex executions are built from simpler atomic steps.

For a concrete example, consider a simple state transition in our token contract. Starting from configuration $c_0 = \langle \text{Uninitialized}, \rho_0, \sigma_0 \rangle$, we can derive the initialization transition:

$$\frac{\rho_0(\text{admin}, \text{Owner})}{\langle \text{Uninitialized}, \rho_0, \sigma_0 \rangle \xrightarrow{\text{initialize(admin)}} \langle \text{Active}, \rho_0, \sigma_0[\text{initialized} \mapsto \text{true}] \rangle}$$

This derivation shows that if the admin has the Owner role, the contract can transition from Uninitialized to Active, updating the environment to mark the contract as initialized. The derivation tree provides a formal proof that this transition is valid according to the operational semantics.

For smart contracts, operational semantics provides a rigorous way to define how contract execution proceeds [117]. Each function call or operation can be modeled as a transition rule that specifies: the preconditions that must hold (guards, role constraints), the state changes that occur (variable updates, state transitions), and any side effects (external calls, events). This formal specification enables precise reasoning about contract behaviour, allowing us to prove properties such as safety (the contract never reaches an invalid state) and liveness (the contract can always make progress under certain conditions). The operational semantics also provides a foundation for automated analysis tools, such as model checkers [56] and symbolic execution engines, which can systematically explore the state space defined by the semantics.

The combination of LTSs and operational semantics provides a powerful framework for modelling smart contracts. The LTS structure captures the state space and possible transitions, while operational semantics defines the precise rules governing how these transitions occur, including the conditions under which they are enabled and the effects they produce. Together, they enable both static analysis (reasoning about all possible behaviours) and dynamic analysis (exploring specific execution paths), making them essential tools for understanding and verifying smart contract behaviour. These foundational structures form the basis upon which behavioural type systems are built, as we will see in the following subsection.

II.3.3 Behavioural Types and Typestates

Behavioural type systems extend traditional static type systems by capturing not just the structure of data, but the dynamic behaviour of programs—particularly how components interact through communication protocols and how behaviour evolves with system state. This perspective is fundamental to modelling smart contracts, which are inherently stateful systems where the same function call may have different effects depending on the current state and the roles of participants.

Session Types and Communication Protocols

Session types, introduced by Honda et al. [111], extend traditional type systems to capture communication protocols between distributed components [91]. Unlike static types that describe the structure of data, session types describe the *sequence* and *structure* of messages exchanged between processes, enabling static verification that communication follows a prescribed protocol. This capability is crucial for distributed systems where communication errors can lead to deadlocks, protocol violations, or security vulnerabilities [111].

A *session* represents a single conversation among processes over communication channels [111]. Each endpoint of a channel has a *session type* that specifies the sequence of messages it will send or receive. Session types are complementary: if one endpoint has type S, its partner must have a type compatible with the *dual* $\overline{S}$ (the exact dual in the basic case; a subtype thereof in systems with subtyping), ensuring that every send has a corresponding receive and vice versa [111].

Basic operations in session types include: *send* (!), which sends a value of a given type; *receive* (?), which receives a value; *selection* ($\oplus$), which represents an internal choice where the process decides which branch to take; and *branching* (&), which represents an external choice where the process offers multiple options and waits for the other party to choose. Prefixing (action before continuation) is denoted by the dot (.), and recursion enables protocols that repeat or have cycles.

A session type specifies the behaviour of one endpoint in a communication channel. For example, a session type $S = !\text{int}.?\text{bool}.S$ describes a protocol where the process sends an integer, receives a boolean, and then repeats. The notation !int means "send an integer", ?bool means "receive a boolean", and the dot (.) before the continuation S denotes prefixing. The recursive definition S at the end indicates that the protocol repeats indefinitely.

Consider a simple vending machine protocol as an example. The protocol involves two participants: a *Customer* and a *VendingMachine*. The customer initiates the interaction by selecting a product, then inserts payment. The machine verifies the payment and either dispenses the product or returns the payment if insufficient funds are provided. The complete protocol can be specified as a single session type:

For the Customer endpoint:

$$C = !\text{select}.!\text{payment}.\& \begin{cases} ?\text{product}.C \\ ?\text{refund}.C \end{cases}$$

For the VendingMachine endpoint (dual of Customer):

$$V = ?\text{select}.?\text{payment}. \oplus \begin{cases} !\text{product}.V \\ !\text{refund}.V \end{cases}$$

The protocol ensures that: (i) the customer always selects before paying, and (ii) after receiving payment, the machine directly responds with either product delivery or refund. The duality principle is correctly applied: the machine uses $\oplus$ (internal choice) to decide whether to dispense or refund based on payment verification, while the customer uses & (external choice) to wait for and handle whichever outcome the machine chooses. The customer must be prepared to handle either outcome, but the machine controls which branch is taken. The outcome is implicit in whether product or refund is sent—no separate outcome message is needed. The protocol is recursive (indicated by C and V at the end), allowing the interaction to repeat for multiple transactions.

This vending machine example demonstrates how session types capture the temporal ordering of interactions: the customer cannot pay before selecting, and the machine cannot dispense before

receiving payment. The protocol also shows how duality works: the machine uses $\oplus$ (internal choice) to make an internal decision (based on payment verification) to either send product or refund, while the customer uses & (external choice) to wait for and accept whichever outcome the machine chooses. The $\oplus$ operator represents the machine's control over which branch to take, while & represents the customer's obligation to handle both possibilities, ensuring that all possible outcomes are handled correctly.

Duality is fundamental to session types. Given a session type S, its dual $\overline{S}$ is obtained by: (i) swapping sends (!) and receives (?), (ii) swapping internal choices ($\oplus$) with external choices (&), and (iii) recursively applying duality to sub-terms. The special type *end* represents a terminated session where no further communication occurs. Formally, the duality rules are:

$$\begin{aligned}
\overline{!T.S} &= ?T.\overline{S} \\
\overline{?T.S} &= !T.\overline{S} \\
\overline{S_1 \oplus S_2} &= \overline{S_1} \& \overline{S_2} \\
\overline{S_1 \& S_2} &= \overline{S_1} \oplus \overline{S_2} \\
\overline{\text{end}} &= \text{end}
\end{aligned}$$

The last rule states that *end* is self-dual, meaning a terminated session has no further communication obligations for either endpoint.

Two endpoints with dual session types are guaranteed to communicate without deadlock or protocol violations. If process P has session type S and process Q has session type $\overline{S}$, then P and Q can safely communicate over a shared channel. The duality ensures that every send has a matching receive, and every internal choice has a matching external choice.

Subtyping in session types [90, 53] enables safe protocol specialization. A session type S_1 is a subtype of S_2 ($S_1 \leq S_2$) if S_1 can be used wherever S_2 is expected. Subtyping for session types follows contravariant rules: a process that sends more options (larger $\oplus$ choice) or receives fewer options (smaller & choice) is more general and can be substituted for a more specific process. This enables protocol evolution where new options can be added without breaking existing implementations.

Session delegation enables transferring a session channel to another process, enabling dynamic protocol composition and higher-order communication patterns.

Multiparty session types extend this concept to multiple participants, enabling the specification of global communication protocols that coordinate multiple processes [112]. A *global type* specifies the overall communication pattern among all participants, while *local types* specify each participant's view of the protocol. The *projection* operation [69, 25] extracts local types from global types, ensuring that each participant's behaviour is consistent with the global protocol. This framework has been extended and applied in various contexts, including flexible multiparty session protocols [92, 93].

Consider a three-party escrow protocol involving a Buyer (B), Seller (S), and EscrowAgent (E). The global protocol specifies: (i) the Buyer sends the payment to the EscrowAgent, (ii) the EscrowAgent notifies the Seller that the payment has been received, (iii) the Seller sends a shipment confirmation to the EscrowAgent, (iv) the EscrowAgent sends the product to the Buyer, and (v) the EscrowAgent releases the payment to the Seller. The global type can be written as:

$$\begin{aligned}
G = B &\to E : \mathsf{Payment}; \\
E &\to S : \mathsf{PaymentReceived}; \\
S &\to E : \mathsf{ShipmentConfirmation}; \\
E &\to B : \mathsf{Product}; \\
E &\to S : \mathsf{ReleasePayment}; \\
&\mathsf{end}
\end{aligned}$$

Projecting this global type onto each participant yields the following local session types.

Projection for the Buyer

$$G \upharpoonright B = E!\mathsf{Payment};\ E?\mathsf{Product};\ \mathsf{end}$$

Projection for the Seller

$$G \upharpoonright S = E?\mathsf{PaymentReceived};\ E!\mathsf{ShipmentConfirmation};\ E?\mathsf{ReleasePayment};\ \mathsf{end}$$

Projection for the EscrowAgent

$$G \upharpoonright E = B?\mathsf{Payment};\ S!\mathsf{PaymentReceived};\ S?\mathsf{ShipmentConfirmation};\ B!\mathsf{Product};\ S!\mathsf{ReleasePayment};\ \mathsf{end}$$

This projection ensures that each participant follows their local protocol, which collectively implements the global protocol. Multiparty session types are particularly relevant for smart contracts, which often involve multiple parties (buyers, sellers, escrow agents, arbitrators) coordinating through well-defined interaction patterns.

Type checking for session types [111] verifies that processes use channels according to their session types. The algorithm tracks the current state of each session, ensuring that sends and receives match, choices are handled correctly, and recursive definitions are well-formed. Type checking is decidable for session types, enabling automated verification of communication protocols [91]. The type checker maintains a session environment that maps channel names to their session types, and updates this environment as sessions progress through their protocols.

Session types are grounded in process calculi, mathematical formalisms for modelling concurrent and distributed systems. Milner's π-calculus [149] provides the foundational theory, where processes communicate by exchanging messages over named channels. The π-calculus and its extensions with session primitives enable type systems that statically verify communication safety, ensuring that processes follow prescribed interaction protocols.

The π-calculus provides primitives for channel creation, message sending, and message receiving. Session types add structure to channels by associating each channel with a protocol that specifies the sequence of messages. This enables static verification that processes use channels correctly, preventing protocol violations and deadlocks.

Session types have been applied to various domains beyond process calculi, including choreography automata and software components [196], distributed systems [111], and choreography-based modelling [94, 46]. While session types provide a theoretical foundation for modelling communication protocols, their direct application to smart contracts is less common: FSM-based approaches are more commonly used for smart contract specification [57, 3, 2, 140, 139, 141, 64, 80, 186, 179]. In the context of smart contracts, session types can conceptually model the interaction patterns between contracts and their users, ensuring that function calls follow prescribed sequences and that all participants handle all possible outcomes correctly. This provides a formal foundation for verifying that smart contracts implement their intended communication protocols correctly.

Typestate Systems, Linear Types, and State Machines

Behavioural type theory encompasses multiple complementary perspectives on specifying and verifying program behaviour. Beyond session types, three particularly relevant approaches for smart

contracts are typestate systems, linear types, and their unified representation through finite state machines.

Typestate [184] represents a perspective on behavioural typing where objects have behavioural types that evolve as the system state changes. In typestate systems, the type of an object depends not only on its static structure, but on the current *state* of execution. An object may have type T_1 in one state and type T_2 in another, where the transition between states is governed by method calls that satisfy certain preconditions. This perspective naturally connects to finite state machines: the states of an FSM correspond to different typestates, and transitions correspond to operations that change the object's behavioural type. Garcia et al. [89] provide a comprehensive foundation for typestate-oriented programming, showing how typestate systems can be formalized and how they enable static verification of state-dependent behaviour. In a typestate system, the available operations in each state define the object's behavioural interface at that point in execution, ensuring that operations can only be called when appropriate.

Smart contracts are inherently stateful systems where the same function call may have different effects depending on the current state. Typestate systems provide a natural framework for capturing this: a contract in state q_1 has a different behavioural type (available operations) than the same contract in state q_2. This makes typestate a natural fit for modelling smart contract behaviour, as demonstrated by languages such as Obsidian [57] and Stipula [64], which embed typestate checking directly into the contract programming language.

Building upon the state-dependent nature of typestate systems, linear type systems provide another dimension of behavioural typing, focusing on resource management and ownership [203]. In a linear type system, resources must be used exactly once, preventing resource leaks and ensuring proper cleanup. This is particularly relevant for smart contracts managing valuable assets: linear types can ensure that tokens, ownership rights, or other assets are properly transferred rather than duplicated or lost. Ownership type systems [55] extend linear types to support flexible aliasing while maintaining safety guarantees. The key insight is that ownership can be transferred or temporarily shared, enabling more expressive programming while preserving resource safety invariants. Languages like Obsidian leverage linear types and ownership to control assets in smart contracts, ensuring that digital assets cannot be double-spent or accidentally destroyed [57].

For smart contracts, linear and ownership types can ensure that digital assets are properly managed: tokens cannot be double-spent, ownership cannot be duplicated, and assets cannot be accidentally destroyed. These guarantees are particularly critical in blockchain environments where errors cannot be corrected after deployment. The resource-aware programming discipline enforced by linear types aligns with the need for precise asset management in financial smart contracts.

While session types, typestate systems, and linear types are typically presented using different formalisms, they share a common underlying structure that can be captured using finite state machines. Session types can be viewed as FSMs where states represent protocol phases and transitions represent message exchanges. Typestate systems are explicitly state-based, with states corresponding to different behavioural types. Linear types can be understood as tracking resource state transitions, where each use of a resource moves it through a well-defined state space. This unified perspective enables a comprehensive framework for specifying behavioural properties across different type system paradigms.

Data-aware finite state machines [3] extend this foundation by enriching states with data fields and transitions with guards and assignments. This enhancement enables FSMs to capture not just the *structure* of behaviour (which operations are available), but also the *conditions* under which operations are enabled (guards) and how operations modify the system state (assignments). The connection between behavioural types and data-aware FSMs is fundamental to this thesis: EDAM can be understood as a behavioural type system where types are specified as extended finite state machines.

The behavioural type of a smart contract, as defined in EDAM, specifies:

- The states the contract can be in
- The transitions available in each state
- The guards that must be satisfied

- The role constraints
- The data transformations

This perspective positions EDAM within the broader landscape of behavioural type theory, showing how it extends and synthesizes concepts from session types, typestate systems, and linear types to provide a unified framework for specifying smart contract behaviour. The FSM structure provides an intuitive and visually accessible representation, while the underlying behavioural type theory ensures formal rigor and enables automated verification.

II.4 Smart Contract Modelling and Specification

Having established the theoretical foundations of behavioural type theory and its application to smart contract specification, we now examine how these concepts are instantiated in practice through various modelling and specification approaches. These approaches translate theoretical principles into concrete frameworks for describing smart contract behaviour, enabling developers to specify contracts formally before implementation and verification.

Model-driven development applies behavioural type theory to practical code generation. By treating models as behavioural type specifications, we can generate code that is guaranteed to implement the specified protocol, preserving the safety properties established at the type level. This section reviews the landscape of smart contract modelling approaches, organised by their underlying methodology: FSM and automata-based approaches, choreography and global specification languages, and language-based type systems.

The literature on models of coordination is vast. We restrict our comparison to tool-supported approaches within three categories: FSM-based models, formal language models, and domain-specific languages for smart contracts. We compare EDAM with other coordination models as well as with approaches specific to smart contracts.

II.4.1 FSM and Automata-based Approaches

The idea of decorating finite state machines with constraints on data is not new [23, 59, 128, 127], and it has been applied to other behavioural models like Petri nets [67, 137]. Coordination models of distributed systems based on extensions of FSMs with fragments of first-order logic have appeared in the literature. Notably, data-aware versions of Behaviour, Interaction, Priority (BIP) and REO have been studied in [72, 171]. As in EDAM, data can be accessed and modified as part of an interaction in both BIP and REO. A difference with our model is that interactions can involve more participants and updates are local to the participants of interactions. This also applies to recent models based on asserted communicating finite-state machines [169, 13].

These coordination models share fundamental concerns with behavioural type theory: both aim to specify and verify the interaction patterns between components. BIP –used for component-based systems in [36]– and REO coordination language can be understood as instances of behavioural typing, where the coordination protocol specifies the behavioural contract that components must follow. Data-aware extensions of these models enrich behavioural specifications with data constraints, paralleling the extension of session types to data-aware session types, and of typestate systems to systems with rich state representations. EDAM synthesizes these developments by providing a coordination model that is explicitly grounded in behavioural type theory, enabling specification of smart contracts as behavioural types (FSMs) that capture communication protocols, state-dependent behaviour, and resource management within a unified formal model.

The TRAC tool, developed by Afonso et al. [3], extends this approach by providing support for data-aware coordination in multiparty distributed systems. Their work demonstrates how finite-state machines with transition labels resembling Hoare triples can be used to specify and verify complex smart contract interactions. In this respect, the closest approach to ours is [3]; our model enhances it with dynamic role-based access control, improvements in participant-aware transitions, and cross-coordinator interactions through call tries. Such models lack either dynamic participants handling or cross-coordinator interaction. These limitations become critical in the

context of smart contracts, which require precise execution semantics, transactional guarantees, and verifiable inter-contract communication.

Protocol languages that advocate a programming style based on FSMs to specify smart contracts include FSolidM [140, 139, 141] , Symboleo [179], and SmartScribble [80]; FSM-based code generation is also explored with LLMs [132]. FSolidM relies on model checking Computation Tree Logic (CTL) formulae to verify safety and liveness properties (including deadlock-freedom). The automata have a global state, represented by contract, input, and output variables, and transitions are guarded by boolean conditions on these variables. The tool has been extended to feature code generation and interaction verification between multiple smart contracts [158]. This progress marked a substantial improvement in detecting common vulnerabilities such as re-entrancy attacks and fallback errors.

The interaction patterns programmable with SmartScribble [80] correspond to FSMs. The tool extracts Plutus code[1] from valid protocol descriptions, leaving to the developer the task to fill in the application logic. Automatic code generation greatly accelerates development time, and guarantees correct-by-construction code (in what concerns the interaction patterns). Participants are first-class citizens in EDAM while FSolidM encodes them with variables and SmartScribble identifies participants with roles which are fixed statically. Also, SmartScribble does not support assertions.

Mavridou and Laszka [140, 139] introduced a FSM based approach for designing secure Ethereum smart contracts. Their methodology leverages the inherent state-based nature of smart contracts to create formal models that can be systematically verified for security properties. The VeriSolid framework, developed by Mavridou et al. [141], extends this approach by providing a complete model-driven development environment that generates correct-by-design smart contracts. Interface automata have been used to formalise smart contracts [134]. In particular, BIP is used by [141] as an intermediate representation for the FSM to benefit from deadlock freedom checking capabilities of the BIP tool and then is converted to nuXmv [54] for verification. The nuXmv tool is a model checker for temporal logics, including CTL and Linear Temporal Logic (LTL). It is a state-of-the-art tool for model checking and verification of distributed systems.

Mavridou and Laszkaintroduced FSolidM [140] , a model-driven engineering framework that enables the specification of smart contracts using high-level models that are then automatically translated to Solidity code. FSolidM represents a significant advancement by providing a visual modelling environment that abstracts away the complexities of blockchain-specific programming constructs. The framework supports the modelling of contract states, transitions, and business logic through an intuitive graphical interface, making smart contract development accessible to domain experts who may not have deep blockchain programming expertise.

Jurgelaitis et al. [118] extended this approach by demonstrating how Unified Modeling Language (UML) state machines can be used to generate Solidity code, providing a bridge between established software engineering practices and smart contract development. Their work emphasises the importance of visual modelling languages in making smart contract development more accessible to traditional software engineers. Similarly, Tsiounis and Kontogiannis [195] explored goal-oriented modelling approaches for smart contract generation. Model-driven code generation [182] and LLM-assisted generation [168] were also used to generate smart contracts. They proposed a model-driven approach where extended goal models are used to denote the structure and inter-dependencies of smart contracts as well as stakeholder objectives. The authors introduced a Domain-Specific Language (DSL) for expressing these extended goal models and developed a transformation process that converts the Abstract Syntax Trees of DSL programs into Solidity smart contract source code. This approach ensures that the generated code conforms to the model specification, facilitating subsequent analysis, understanding, and maintenance of smart contract systems.

[1] Plutus is the programming language to develop smart contracts for the Cardano blockchain.

II.4.2 Choreography and Global Specifications

The concept of choreography-based modelling has emerged as a powerful paradigm for complex multi-party smart contract systems. Choreography automata are closely related to multiparty session types: both specify global interaction protocols that coordinate multiple participants. The key difference is that choreography automata focus on the visual representation and practical code generation, while session types emphasize the formal type theory and static verification. Our work bridges this gap by providing a framework that combines the intuitive modelling of choreographies with the rigorous foundations of behavioural type theory.

Barbanera et al. [25] introduced choreography automata as a formal model for specifying the behaviour of communicating systems, which can be applied to some smart contract specifications, but being not data-aware. Choreography automata [164] and their extension with assertions [92, 93] are global specifications for communicating systems behind Corinne [164] and CAScr [92]. Both these tools are designed to check well-formedness conditions different than ours (respectively those in [164] and in [92, 93]) and neither of them supports multiple instances of roles. Assertions in CAScr are not guards; they express rely-guarantee conditions between the sender and the receiver of interactions. In the same vein, CAT [32] is an automata-based tool for the verification of communication protocols. Based on *contract automata* [35, 33, 34], CAT is not data-aware and its contracts purely regard the communication interface of participants (which are also fixed).

These approaches enable the modelling of complex interactions between multiple contracts and participants, addressing one of the key challenges in decentralised application design. But the applicability to smart contracts can be limited.

Corradini et al. [60, 61] extended the choreography-based modelling approaches discussed above by demonstrating how Business Process Model and Notation (BPMN) [162] choreography diagrams can be translated into executable smart contracts. Their approach leverages the expressive power of business process modelling to capture complex multi-party interactions, making it particularly suitable for enterprise blockchain applications. The work shows how traditional business process modelling can be seamlessly integrated with blockchain execution, which enable business analysts to participate directly in smart contract design. Notably, [62] is an extension of [61] which allows multiple participants to play the same role. This line of work has fairly different goals than ours: its aim is to exploit blockchain immutability to record the execution in a secure way. Our approach instead concerns modelling and verifying distributed applications coordinated by a FSM, possibly implemented as a smart contract.

The integration of data-aware modelling has also become crucial for modern smart contract applications. Hahn et al. [101, 101] explored data-aware choreography modelling, which extends traditional process modelling to include explicit data flow specifications. This is particularly relevant for smart contracts that need to manage complex data transformations and cross-party data exchanges, such as those found in supply chain and financial applications.

II.4.3 Language-Based and Type-Based Approaches

The previous approaches take a top-down, model-driven perspective–they propose abstractions to rigorously define formal models of computations; only some of them support automatic generation of smart contract code. An alternative approach embeds behavioural type checking directly into the programming language itself, ensuring type safety at the language level rather than through code generation from models.

Typestate-based languages such as Obsidian [57] and Stipula [126, 64] are explicit instantiations of behavioural type theory for smart contracts. They embed the definition of the FSM in the contracts' programming language itself. Both are inspired by *typestates* [184] and their use in programming languages [89]: states are explicit entities with a defined API; invocations to an operation of the API of a state possibly update the variables of the program and yield to possibly another state. In this respect, the execution model of both languages is quite similar to EDAM. Obsidian uses typestates and linear types [203] to control assets (critical resources of smart contracts). Safe, yet flexible, aliasing is ensured with an ownership type system [55]. Two case studies show usability of Obsidian in real-world scenarios (parametric insurance and supply chain management).

Stipula focuses on legal contracts and provides a strict discipline to guarantee *liquidity*: no asset remains frozen forever. Laneve [126, 63] developed theoretical foundations for liquidity analysis in resource-aware programming, providing formal methods for reasoning about resource availability and transfer patterns in smart contracts.

EDAM takes a complementary approach: rather than embedding typestate into the contract language, it provides a modelling language that generates typestate-enforcing code. This separation of concerns enables verification at the model level before code generation, while still benefiting from the safety guarantees provided by typestate systems. The model-driven approach also supports platform-agnostic generation, allowing the same model to produce code for different blockchain platforms.

Other language-based approaches include Flint and Move [38], which incorporate safety features and resource management directly into the language design. In the same vein, the intermediate-level language Scilla [178] provides an automata-based model amenable to mechanised verification. An application of Event-B to smart contracts generating automatically Solidity code appeared in [180]. At the time of writing, there is no known report on the validation of the tool with benchmarks. The work by Singh et al. [180] on formal verification and code generation for Solidity smart contracts addresses the challenge of translating high-level requirements into executable code through intermediate formal models. The work by Braghin et al. [41, 40] models contracts as Abstract State Machine (ASM) s for formal verification, but they do not generate code and tests. This approach provides strong formal guarantees but lacks the practical code generation capabilities needed for real-world development.

Building upon the model-driven and language-based approaches discussed above, Hamdaqa et al. [103, 102] have explored Domain-Specific Modeling Language (DSML) s tailored specifically for smart contracts. They introduced iContractML, a comprehensive DSML that supports modelling and deployment across multiple blockchain platforms, including Ethereum, Hyperledger, and Azure Blockchain. This work addresses the challenge of platform fragmentation by providing a unified modelling approach that can generate platform-specific implementations from a single abstract specification. Frantz and Nowostawski [87] also explored automated generation of smart contracts from institutional specifications.

The interesting work by Al-Azzoni et al. [5] introduces a DSL for smart contracts, and supports both code generation in the Digital Asset Modeling Language (DAML) language and test generation. Such DSL allows to express complex RBAC policies, but lacks in data awareness. This represents a step towards integrated development environments, though it does not fully address the data-aware modelling requirements of modern smart contracts.

II.5 Verification and Testing Approaches

While modelling and specification approaches provide formal descriptions of smart contract behaviour, these specifications must be validated to ensure they correctly capture intended requirements and that implementations conform to their specifications. Verification and testing techniques bridge the gap between specifications and deployed contracts, providing mechanisms to detect errors, prove correctness properties, and validate contract behaviour before deployment. This section reviews the landscape of smart contract verification approaches, organised by their underlying methodology: model checking and static analysis, theorem proving and interactive verification, and automated test generation.

The verification of smart contracts has been addressed through multiple complementary approaches [135, 114], ranging from formal verification techniques [37] to dynamic testing methods. Each approach addresses different aspects of the verification challenge, from ensuring functional correctness to detecting security vulnerabilities. Verification in most cases starts with the construction of a model of the contract. Bai et al. [22] introduced the application of formal methods, specifically model checking, to the modelling and verification of smart contracts leveraged in [212, 157, 210] and SPIN [110]. Their work describes the use of an abstract model to formally model smart contract templates and verify key properties, aiming to reduce development errors and increase contract reliability.

II.5.1 Formal Verification Approaches

Model checking [56] exhaustively explores a system's state space to verify temporal properties, offering automated verification without manual proof construction. Clarke et al. [56] established the theoretical foundation; Holzmann [110] (SPIN) and Emerson [78] on temporal and modal logic provided groundwork for concurrent systems, directly relevant to blockchain contracts [83]. Smart contracts present state space explosion due to composition, unbounded data structures, and complex interaction patterns. Abstraction techniques focus on relevant behaviour while ignoring implementation details; statistical model checking [197] has also been applied. Reduction techniques eliminate redundant states or explore sufficient subsets to verify properties while maintaining soundness.

SolCMC [10] integrates model checking into the Solidity compiler, providing seamless verification without separate tooling or manual model construction. As part of the Ethereum Foundation toolchain, it features multiple reasoning engines and tracks language evolution, enabling automated analysis of non-trivial properties on source code. The nuXmv symbolic model checker [48] has been adapted for contract verification, using Binary Decision Diagram (BDD) s and SMT to handle large state spaces. It supports LTL and CTL for safety properties (nothing bad happens) and liveness properties (something good eventually happens), such as balance non-negativity or eventual processing of withdrawal requests. Predicate abstraction techniques [95] reduce complexity by abstracting concrete states to predicates over state variables, preserving expressible properties while enabling verification of contracts with complex data structures and arithmetic.

Static analysis examines code without execution through pattern matching, data flow, and control flow analysis [146]. Grech et al. [97] introduced Securify, combining static analysis with symbolic execution to detect reentrancy (external calls before state updates), access control gaps, and integer overflow. Securify may report false positives, and complex vulnerabilities may be missed. MadMax [97] addresses out-of-gas conditions, identifying unbounded loops, expensive operations in loops, and recursive calls that could exhaust gas or cause stack overflow. Mythril [71] and Manticore [151] integrate symbolic execution with dynamic analysis to explore complex paths across multiple contracts. Mythril uses SMT solvers to generate concrete inputs for path coverage; Manticore adds concolic testing and can generate test cases exercising specific vulnerabilities.

Permenev et al. [166] present VerX for safety verification of multi-contract systems, using predicate abstraction and CEGAR to prove access control, balance conservation, and reentrancy protection. The tool requires developers to specify properties in a dedicated language [31]; verification is decoupled from the modelling phase, so properties must be constructed manually and may omit aspects developers fail to anticipate. The Contractor toolkit [66] extracts data-aware FSM abstractions from Solidity code annotated with assertions, as benchmarked on the Azure repository [148, 96]. It automatically constructs sound models from annotated code, reducing model construction effort; it does not directly support multi-object protocols, and annotations must be added manually.

Static analysis faces challenges from language complexity (inheritance, modifiers, dynamic dispatch), cross-transaction state tracking, and multi-contract interactions where dependency implementations may be unknown. False positives and false negatives remain concerns. These tools complement each other and other verification techniques, though the landscape leaves room for approaches that integrate modelling, specification, and verification in a unified workflow.

Interactive theorem proving provides mathematical proofs that contracts satisfy specifications under all execution scenarios [9], offering the highest assurance for critical contracts. Proof assistants Isabelle/HOL and Coq are the main vehicles: Isabelle/HOL provides automation via Sledgehammer; Coq's dependent types embed specifications directly in types for compile-time verification.

Amani et al. [12] verified Ethereum bytecode directly in Isabelle/HOL by modelling the EVM instruction set as a state transition system. Each instruction transforms EVM state (stack, memory, storage, program counter). This low-level modelling supports verification independent of compiler optimisations and enables reasoning about functional correctness and gas consumption. Annenkov et al. [15] proposed a Coq framework for smart contracts, leveraging dependent types for access control, resource safety, and reentrancy protection. The FEther interpreter by Yang and Lei [213]

addresses the need for formal Solidity semantics: a definitional interpreter in Coq that models function calls, state variables, events, inheritance, modifiers, library calls, and gas consumption. It handles dynamic dispatch and modifier composition, providing a foundation for rigorous correctness proofs on Solidity programs.

Milo et al. [150] present ConCert, which allows contracts to be written in a subset of Coq so specifications and implementations share the same language. The framework supports property-based testing via QuickChick. ConCert is decoupled from the modelling phase and requires contracts in a specialized language rather than generating code from models. Sorensen [183] extends ConCert with a formal framework for contract upgrades.

Specification styles include invariants (proved by induction over execution traces) and Hoare triples [6] for precondition-postcondition specification of individual functions, enabling modular verification. Bräm et al. [43] developed a resource-aware methodology for smart contracts with unverified code and arbitrary reentrancy, implemented in 2vyper for Vyper[2] contracts. The methodology distinguishes trusted and untrusted code. Modular verification requires interfaces that specify dependency behaviour and assumptions about external contracts (reentrancy, error handling, resource management), balancing strength for verification with flexibility for implementations.

Practical adoption faces challenges [193, 192, 7]: proof effort is high; semantic gaps between formal models and execution can undermine guarantees; compiler optimisations and platform updates introduce discrepancies; scalability for large or multi-contract systems remains substantial. For critical contracts, the investment can be justified; automation and tooling continue to improve the barrier to entry.

II.5.2 Automated Test Generation

The approaches above focus on verification of properties against specifications or models; automated test generation complements them by systematically exploring execution paths and generating test cases that exercise contract behaviour without requiring formal specifications.

The generation of comprehensive test suites for smart contracts has become increasingly important as contract complexity grows. Manual test creation is time-consuming, error-prone, and often fails to achieve adequate coverage of the contract's behaviour space. Automated test generation techniques address these limitations by systematically exploring execution paths, generating test inputs that exercise different behaviours, and creating oracles that validate contract outputs. These techniques leverage various program analysis methods, from symbolic execution to search-based optimisation, to create test suites that systematically cover the contract's state space and execution paths.

Symbolic execution is a program analysis technique that executes programs with symbolic rather than concrete inputs, maintaining constraints on input values as execution proceeds. When execution branches based on symbolic conditions, both paths are explored, enabling systematic coverage of all feasible execution paths. Driessen et al. [74] developed AGSolT and SolAR [75], two automated test-case generation tools that leverage symbolic execution; other approaches include SMARTGIFT [190], TestSmart [84], and SynTest-Solidity [161] to create test cases covering various execution paths in Solidity smart contracts. AGSolT uses symbolic execution to explore contract execution paths, generating test cases that cover different branches and edge cases. The tool maintains path conditions (constraints on input values) as execution proceeds, and uses constraint solvers to generate concrete inputs that satisfy these conditions. This approach enables systematic exploration of the contract's behaviour space, generating tests that exercise paths that might be missed by manual testing. SolAR extends AGSolT with additional capabilities for analysing contract interactions and generating tests that exercise multi-contract scenarios. The tool can generate tests that involve multiple contracts, enabling verification of complex decentralized applications where contracts interact through function calls and events. Britikov et al. [42] developed SolTG, a CHC-based test case generator that exhaustively enumerates symbolic path constraints from the

[2]Vyper is a programming language for writing smart contracts on the Ethereum blockchain.

contract's constrained Horn clause (CHC) representation and uses SMT solvers to find concrete input values. SolTG generates test cases as sequences of function calls with concrete parameter values, enabling systematic exploration of execution scenarios and achieving high coverage for industrial-grade Solidity code. The effectiveness of symbolic execution depends on the ability to solve constraints efficiently. Complex constraints involving cryptographic operations, loops with many iterations, or unbounded data structures can make constraint solving intractable. Tools must balance exploration depth (covering more paths) with solving efficiency (generating tests in reasonable time).

Mutation testing evaluates test suite quality by measuring its ability to detect artificially introduced faults (mutants). A mutant is a slightly modified version of the program, created by applying a mutation operator (such as changing a comparison operator or modifying a constant). A test suite's mutation score is the percentage of mutants it detects; higher scores indicate better test quality. Barboni et al. [26] introduced SUMO, a mutation testing strategy specifically designed for Solidity smart contracts. Their approach systematically introduces faults into contract code to evaluate the effectiveness of existing test suites and identify areas requiring additional testing coverage. SUMO includes domain-specific mutation operators that reflect common smart contract vulnerabilities, such as reentrancy patterns, access control errors, and integer overflow/underflow. Mutation operators for smart contracts must reflect the unique characteristics of blockchain programming. Operators that modify gas-related operations, change access control checks, or alter reentrancy guards are more relevant than generic operators that modify arithmetic expressions. SUMO's operators are designed based on analysis of real-world vulnerabilities, ensuring that mutation testing evaluates tests against realistic faults. The computational cost of mutation testing can be prohibitive for large contracts, as each mutant must be executed against the entire test suite. Techniques such as mutant sampling (testing a subset of mutants) and mutant clustering (grouping equivalent mutants) can reduce this cost while maintaining assessment quality.

Search-based testing uses optimisation algorithms to search for test inputs that maximise coverage metrics or reveal faults. Genetic algorithms, simulated annealing, and particle swarm optimisation have been applied to test generation, treating test generation as an optimisation problem where the fitness function measures test quality (e.g., branch coverage, fault detection). Du et al. [76] developed a test case generation approach for Ethereum smart contracts based on data dependency analysis of state variables. Their approach uses genetic algorithms to optimize test inputs, with fitness functions that reward tests that exercise different data dependencies. By analysing how state variables depend on inputs and previous operations, the approach generates tests that systematically explore the contract's state space. The genetic algorithm maintains a population of test cases, evolving them through crossover and mutation operations to improve fitness. This enables discovery of test inputs that exercise complex interaction patterns between state variables, revealing vulnerabilities that depend on specific state configurations. Tools like SV-Gen [76] combine symbolic execution with genetic algorithms for test generation. The hybrid approach uses symbolic execution to identify interesting execution paths, then uses genetic algorithms to optimize inputs that exercise these paths. This combination leverages the systematic exploration of symbolic execution with the optimisation capabilities of search algorithms. Search-based approaches face challenges in defining effective fitness functions. Coverage metrics (such as branch coverage or statement coverage) may not correlate well with fault detection, leading to tests that achieve high coverage but miss critical bugs. Domain-specific fitness functions that incorporate knowledge about common vulnerabilities can improve effectiveness, but require expertise to design.

On-chain test generation leverages real-world contract interactions recorded on the blockchain to generate test cases. This approach extracts execution traces from deployed contracts, identifying common interaction patterns and generating tests that exercise these patterns. SolMigrator [214] uses on-chain behaviour extraction for test generation, demonstrating how real-world contract interactions can be leveraged to improve test coverage and effectiveness. The approach involves analysing transaction histories to identify common function call sequences, parameter value distributions, and interaction patterns between contracts. This analysis reveals how contracts are actually used in practice, enabling generation of tests that reflect realistic usage scenarios. Tests generated from on-chain data can reveal issues that occur in production but might be missed by tests generated

from specifications or code analysis. Zhang et al. [217] explored automated test generation for smart contracts via on-chain test case augmentation and migration. Their approach extracts test cases from one contract and adapts them for testing similar contracts, enabling reuse of test knowledge across contract families. This is particularly valuable for testing contracts that implement standard interfaces (such as ERC20 tokens), where tests from one implementation can be adapted for others. On-chain extraction faces challenges in handling the scale and noise of blockchain data. Transaction histories can be enormous, requiring efficient filtering and sampling techniques. Additionally, malicious or erroneous transactions can pollute the extracted patterns, requiring careful analysis to distinguish legitimate usage from attacks or bugs.

Model-based testing derives test cases directly from formal models of the system under test. Unlike random or search-based testing, model-based testing leverages the structure and semantics of the model to generate test suites that systematically explore the system's behaviour space. The fundamental principle is that a formal model captures the intended behaviour of the system, and test cases can be generated by traversing the model's state space and exploring different execution paths. The work by Boogaard et al. [39] on model-driven development for smart contracts demonstrates how formal models can be leveraged to generate systematic code that cover various execution scenarios and edge cases. Their framework transforms models into executable code, ensuring that code generation is directly tied to the behavioural specification. ModCon [130] uses user-specified models for test oracle definition, test generation guidance, and adequacy measurement. The tool enables developers to specify models that capture contract behaviour, including states, transitions, guards, and invariants. From these models, ModCon generates abstract test cases by exploring the model's state space, often using symbolic execution or constraint solving techniques. Abstract test cases are then concretized by solving constraints to obtain concrete input values. The generated tests are executed against the implementation, and results are compared with the model's expected behaviour. Model-based testing for smart contracts typically involves several stages: (1) constructing a formal model that captures the contract's behaviour, including states, transitions, guards, and invariants; (2) generating abstract test cases by exploring the model's state space, often using symbolic execution or constraint solving techniques; (3) concretizing abstract test cases by solving constraints to obtain concrete input values; (4) executing the concrete tests against the implementation and comparing the results with the model's expected behaviour. The effectiveness of model-based testing depends on the quality and completeness of the underlying model. A model that accurately captures the contract's behaviour will generate tests that effectively validate the implementation. Conversely, an incomplete or incorrect model will produce tests that miss important scenarios or validate incorrect behaviour. This highlights the importance of model validation and the need for systematic approaches to model construction and refinement [39].

Test oracle generation determines whether test executions produce correct results. For smart contracts, oracles must validate not just return values, but also state changes, event emissions, and gas consumption. Generating effective oracles is challenging, as contracts may have complex state spaces and multiple observable outputs. Property-based oracles specify general properties that should hold for all executions, rather than checking specific expected values. For example, a token contract oracle might verify that the sum of balances never exceeds total supply, or that transfers never create or destroy tokens. These properties can be derived from contract specifications or domain knowledge, enabling validation without requiring precise expected values. Model-based oracles compare contract behaviour against a formal model, checking that executions conform to the model's specified behaviour. This approach requires a formal model of the contract, but enables comprehensive validation that covers all observable aspects of execution. The model serves as a reference implementation, and tests verify that the actual contract matches the model's behaviour. The integration of automated test generation with formal models provides a powerful approach to smart contract verification, enabling systematic test generation that is directly tied to behavioural specifications rather than requiring manual test case construction.

II.5.3 Fuzzing and Dynamic Analysis

Dynamic analysis techniques have proven highly effective for identifying runtime vulnerabilities and unexpected behaviours in smart contracts. Unlike static analysis, which reasons about code without executing it, dynamic analysis executes contracts with concrete inputs and observes their runtime behaviour. This enables detection of vulnerabilities that depend on specific execution contexts, state configurations, or input values that may be difficult to reason about statically. Fuzzing, in particular, has emerged as a key strategy for uncovering subtle bugs that may not be detected through static analysis alone, leveraging automated input generation to systematically explore the contract's execution space and identify inputs that trigger vulnerabilities or unexpected behaviours.

Fuzzing techniques for smart contracts can be categorized into several approaches: blackbox fuzzing, which generates inputs without knowledge of the contract's internal structure; greybox fuzzing, which uses coverage feedback to guide input generation; and property-based fuzzing, which generates inputs to test specific invariants or properties. Each approach has different strengths: blackbox fuzzing is simple and requires no contract analysis, greybox fuzzing achieves better coverage by using execution feedback, and property-based fuzzing focuses on validating specific security properties. Jiang et al. [116] developed ContractFuzzer, which applies automated fuzzing techniques to smart contract testing, demonstrating how random and semi-random input generation can reveal vulnerabilities such as reentrancy, integer overflows, and exception disorders during contract execution. ContractFuzzer generates test inputs by analysing contract ABIs and creating function calls with randomly generated parameters. The tool monitors contract execution to detect common vulnerability patterns, such as reentrancy attacks (detected by monitoring external calls and state changes), integer overflow/underflow (detected by checking arithmetic operations), and exception disorders (detected by monitoring exception handling). The fuzzer maintains a test suite that evolves over time, with successful test cases (those that trigger vulnerabilities) being preserved and used to guide further exploration. GasFuzzer [19] is a blackbox fuzzer that fuzzes Ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. The tool uses a gas-greedy strategy to generate test inputs with different gas consumptions, and a gas-leveling strategy to generate test inputs with different gas allowances. The tool uses a gas coverage criterion to measure the coverage of the test inputs, and a gas-leveling strategy to generate test inputs with different gas allowances.

The Echidna tool by Grieco et al. [98] provides fast and effective fuzzing for Ethereum smart contracts, with particular emphasis on identifying gas-related vulnerabilities, assertion violations, and unexpected state transitions. Echidna's property-based testing approach allows developers to specify invariants that must hold throughout contract execution, enabling the detection of violations that could lead to security breaches. The tool uses a genetic algorithm to evolve test inputs, with fitness functions that reward inputs that violate specified properties or achieve high code coverage. Echidna supports Solidity-specific features such as events, modifiers, and inheritance, enabling comprehensive testing of complex contract structures. The tool can generate sequences of function calls that explore multi-transaction scenarios, enabling detection of vulnerabilities that only manifest through specific interaction patterns. Echidna's property language allows developers to express complex invariants, such as "the sum of all balances never exceeds total supply" or "only the owner can call administrative functions", enabling targeted testing of security-critical properties. The tool provides detailed reports showing the sequence of function calls that led to property violations, facilitating debugging and vulnerability remediation.

Harvey [209] introduces greybox fuzzing techniques tailored for smart contracts, combining coverage-guided fuzzing with domain-specific knowledge about smart contract vulnerabilities. Greybox fuzzing uses execution feedback (such as code coverage) to guide input generation, focusing exploration on unexplored code paths. Harvey extends this approach with smart contract-specific strategies, such as generating inputs that exercise different branches in access control checks, exploring state transitions that may lead to vulnerabilities, and creating sequences of function calls that trigger complex interaction patterns. The tool maintains coverage maps that track which code paths have been exercised, using this information to prioritize inputs that explore new paths. Harvey also incorporates domain knowledge about common smart contract vulnerabilities, such as

reentrancy patterns and integer overflow conditions, to generate inputs more likely to trigger these vulnerabilities. This combination of coverage guidance and domain knowledge enables Harvey to achieve high vulnerability detection rates while maintaining efficiency.

Sereum [173] combines dynamic taint analysis with runtime monitoring to detect reentrancy attacks as contracts execute on the Ethereum blockchain. Dynamic taint analysis tracks the flow of sensitive data (such as balances or state variables) through contract execution, identifying when tainted data influences control flow or external calls. Sereum monitors contract execution in real-time, tracking state changes and external calls to detect reentrancy patterns. When a contract makes an external call, Sereum checks whether the contract's state has been updated before the call, which would prevent reentrancy attacks. If state has not been updated, Sereum flags a potential vulnerability. The tool can monitor contracts deployed on the Ethereum mainnet, enabling detection of vulnerabilities in production contracts. Sereum's runtime monitoring approach complements static analysis by detecting vulnerabilities that manifest only under specific execution conditions, such as when contracts interact with malicious external contracts or when specific state configurations are reached.

Dynamic analysis techniques face several challenges when applied to smart contracts. The deterministic nature of blockchain execution means that fuzzing must account for blockchain state, including account balances, contract storage, and block properties (timestamp, number, etc.). Effective fuzzing requires maintaining realistic blockchain state configurations, as vulnerabilities may only manifest under specific state conditions. Gas limits constrain the depth and complexity of execution paths that can be explored, requiring fuzzers to balance exploration depth with gas efficiency. The stateful nature of smart contracts means that fuzzing must explore sequences of transactions rather than isolated function calls, significantly increasing the search space. Multi-contract interactions further complicate fuzzing, as vulnerabilities may depend on interactions between multiple contracts. Despite these challenges, dynamic analysis has proven highly effective for vulnerability detection, complementing static analysis by identifying vulnerabilities that depend on runtime conditions.

Recent surveys, such as those by Chen et al. [51] and Durieux et al. [77], provide comprehensive overviews of the landscape of dynamic analysis and fuzzing tools for smart contracts, highlighting the strengths and limitations of various approaches and underscoring the importance of combining multiple techniques for robust vulnerability detection. These surveys categorize tools by their analysis approach (fuzzing, symbolic execution, concolic testing), their target properties (functional correctness, security vulnerabilities, gas efficiency), and their integration with development workflows. The surveys identify common patterns in vulnerability detection, such as the importance of exploring multi-transaction scenarios, the need for realistic blockchain state modelling, and the value of combining static and dynamic analysis. They also highlight gaps in current tooling, such as limited support for multi-contract analysis, challenges in handling complex state spaces, and the need for better integration with development environments. The surveys emphasize that no single technique is sufficient for comprehensive vulnerability detection, and that combining multiple approaches (static analysis, dynamic analysis, formal verification) provides the best coverage of potential vulnerabilities.

II.5.4 Model-Based Testing

Model-based testing is a systematic approach to test generation that derives test cases directly from formal models of the system under test [17, 65]. Unlike random or search-based testing, model-based testing leverages the structure and semantics of the model to generate test suites that systematically explore the system's behaviour space. The fundamental principle is that a formal model captures the intended behaviour of the system, and test cases can be generated by traversing the model's state space and exploring different execution paths. This approach provides several advantages over code-based testing: tests are derived from specifications rather than implementations, enabling test generation before code is written; the model provides a clear specification of expected behaviour, enabling systematic coverage of the behavioural space; and the relationship between model elements and tests provides traceability, making it easier to understand what each test validates and to

update tests when specifications evolve.

The application of model-based testing to smart contracts has been explored through several approaches. Liu et al. [130] developed ModCon, a model-based testing platform that uses user-specified models for test oracle definition, test generation guidance, and adequacy measurement. ModCon accepts models specified as finite state machines with guards, assignments, and invariants. From these models, the tool generates abstract test cases by exploring the model's state space using graph traversal algorithms. Abstract test cases specify sequences of transitions and the conditions that must hold at each step, but do not specify concrete input values. These abstract test cases are then concretized by solving constraints to obtain concrete input values that satisfy the guards and enable the transitions. ModCon uses the model to define test oracles: for each test case, the model specifies the expected state after execution, enabling automatic validation of test results. The tool also uses the model to measure test adequacy, computing coverage metrics based on which states and transitions have been exercised by the test suite.

Boogaard et al. [39] presented a model-driven development for smart contracts, showing how models can be systematically transformed into executable tests. Their framework uses UML state machines as modelling notation, providing a visual representation that is accessible to developers familiar with standard modelling languages. The framework transforms UML models into executable test code, generating test suites that exercise all states and transitions in the model. The transformation process handles complex model features such as hierarchical states, concurrent regions, and history states, ensuring that generated tests accurately reflect the model's behaviour. The framework also generates test scaffolding code that sets up the execution environment, invokes contract functions, and validates results against model expectations.

The effectiveness of model-based testing depends on the quality and completeness of the underlying model. A model that accurately captures the contract's behaviour will generate tests that effectively validate the implementation. Conversely, an incomplete or incorrect model will produce tests that miss important scenarios or validate incorrect behaviour. Model completeness requires that all relevant states, transitions, and behaviours are represented. Model accuracy requires that the model correctly specifies the conditions under which transitions occur and how state is updated. Achieving model completeness and accuracy requires careful analysis of the contract's requirements and behaviour, often involving domain experts and multiple iterations of model refinement. Model validation techniques can help identify inconsistencies or incompleteness: model checking can verify that the model satisfies specified properties, simulation can help validate that the model's behaviour matches expectations, and comparison with reference implementations can identify discrepancies. The challenge of model construction is particularly acute for smart contracts, which may have complex state spaces, intricate interaction patterns, and subtle security requirements. However, the investment in model construction is often justified by the benefits: comprehensive test coverage, early detection of specification errors, and maintainable test suites that evolve with specifications.

II.6 Gap Analysis, Problem Statement, and Positioning of EDAM

The preceding sections have reviewed modelling, verification, and testing approaches for smart contracts. This section synthesises the identified limitations and states how EDAM addresses them.

Fragmentation of the development lifecycle.

Modelling approaches such as UML-based generators [118, 195], DSL s [102, 195, 87], and BPMN-based generators [62, 61] support code generation but typically lack integrated verification and test generation. Verification tools such as VerX [166] and ConCert [150] operate on manually written code and require developers to specify properties or write contracts in a specialised language; they are decoupled from the modelling phase. Fuzzers such as Echidna [98] automatically generate test inputs from property specifications but likewise operate on code rather than models. Works that model contracts as Abstract State Machines for formal verification [41, 40] do not generate code or tests. Approaches that derive tests from on-chain behaviour [214] or search-based exploration [76, 74] do

not tie test generation to model semantics. The result is a fragmented toolchain: models, code, and tests are produced by separate tools with no guarantee that tests cover the behaviour specified in the model or that verified properties are preserved in the implementation.

Static participant modelling.

Tools such as FSolidM and SmartScribble encode participants as variables or fixed role assignments [140, 80]. Choreography automata tools Corinne and CAScr do not support multiple instances of the same role [164, 92]. Real-world contracts often require dynamic participant management—roles assigned or revoked at runtime, multiple instances of a role participating simultaneously—as in governance or marketplace contracts. Without such support, users must use workarounds that complicate specifications and increase the risk of errors.

Limited cross-contract modelling.

Smart contracts frequently interact with multiple other contracts. Models based on single-contract FSMs, such as FSolidM [140] and VeriSolid [141], focus on individual contract behaviour. Extensions such as [158] address this gap by introducing deployment diagrams that specify possible interactions between contract types, thereby enabling modelling and verification of interacting contracts. However, inter-contract interactions remain outside the transition labels of the FSM introduced in [140], and many tools and frameworks still lack such support.

Validation beyond code coverage.

Code coverage indicates which code was executed but not whether tests adequately validate intended behaviour [153]. Mutation testing measures a test suite's ability to detect faults, but its application to smart contracts remains limited [142, 159]. Domain-specific mutation operators for smart contract vulnerabilities exist [26] but are not widely integrated with model-driven workflows. There is a need for validation frameworks that combine coverage, mutation testing with domain-specific operators, and cross-validation with established implementations, tightly integrated with modelling and code generation [50, 24].

Synthesis and positioning of EDAM.

Existing approaches address these concerns in isolation. Model-driven approaches provide visual modelling and code generation but often lack formal semantics grounded in behavioural type theory. Verification and testing tools offer strong analysis capabilities but operate on code, disconnected from modelling. Language-based approaches embed behavioural types in programming languages but require specialised languages rather than model-driven code generation.

EDAM addresses these gaps by providing a unified framework that:

- **Provides a unified behavioural model:** A single data-aware FSM integrates states (contract phases), transitions (operations), guards (preconditions), role constraints (participant access), and assignments (data transformations).
- **Enables model-driven code generation:** The model serves as the behavioural type specification; the generated code conforms to the model.
- **Supports automated test generation:** Tests are derived systematically from behavioural specifications via symbolic execution and trace simulation, tied to model semantics rather than manual test construction.
- **Provides integrated validation:** Coverage and mutation testing (domain-specific operators for smart contracts) assess the quality of generated code and tests; when modelling standards such as ERC20, cross-validation against reference implementations verifies behavioural conformity and specification issues.

Chapter III: Modelling smart contracts

III.1 A gentle introduction to the model

We aim to model distributed systems where components expose API s that can be invoked externally by programs, end users, or other components. Our motivation stems from the need to capture the stateful nature of distributed systems, with particular focus on blockchain protocols and smart contracts. In these contexts, function calls can trigger different behaviours depending on the current state of the system or on the caller. For instance, the same function may fail if called by an unauthorised user, or may produce different results depending on the current balance of an account, the phase of a protocol execution, or the roles assigned to participants. Consider a voting system implemented as a smart contract: a `vote` function may only be available during the voting period (state-dependent), and may only be callable by registered voters (caller-dependent). Similarly, a `withdraw` function in a banking contract may succeed or fail depending on whether sufficient funds are available (state-dependent) and whether the caller is the account owner (caller-dependent). These examples illustrate the need for a modelling framework that can express both temporal constraints (through states) and access control constraints (through participant identities and roles). We see such API s as structures with data fields and functions, similarly to classes in object oriented programming. We call API s *coordinators*, and users *participants*. Each coordinator is also a participant (this enables inter coordinator calls), and each participant has a distinct identity. We model coordinators as a sort of *FSMs*, where transitions are labelled with a function name, list of parameters and other information that we will introduce below. Intuitively, a transition in our model describes how a coordinator moves from one state to another in response to a function call, which is a common way to model the behaviour of distributed systems [91, 89, 160, 184].

Why State Matters. To understand the importance of states in our model, consider a simple marketplace scenario where buyers can make offers and sellers can accept or reject them. In a *stateless system* –where the coordinator has no concept of different phases or stages–all functions would be available at all times. This unrestricted availability makes reasoning about the system's behaviour difficult and error-prone.

First, without states, there is no way to enforce the correct sequence of operations. For instance, a buyer might attempt to finalise a purchase before making an offer, or a seller might try to reject an offer that was never made. Such invalid operations would need to be handled through complex error-checking logic embedded within each function, making the system harder to understand and maintain.

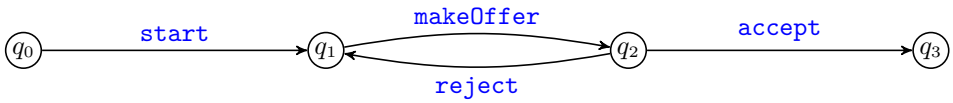
Second, the lack of explicit states obscures the protocol's intended flow. Developers and users cannot easily see at a glance what operations are valid at any given point. This opacity forces them to trace through function implementations to understand the system's behaviour, significantly increasing the cognitive load required to reason about the protocol.

By introducing states, we can ensure that certain functions are only available at specific stages of the protocol life-cycle. Each state represents a distinct phase of the coordinator's execution, and transitions between states occur only when valid operations are performed. This approach makes the protocol's structure explicit: the available functions in each state clearly indicate what actions are permitted, and the transitions show how the system progresses through its life-cycle.

It is worth noting that we can model stateless systems within our framework by using just one state where all functions remain available. However, our state-based approach provides significantly

more flexibility and expressiveness. By making the protocol's phases explicit and constraining function availability accordingly, we enable clearer reasoning about system behaviour and more intuitive understanding of the coordinator's intended operation.

The following example illustrates a simplified marketplace FSM where functions are only available in the states where they are allowed to be called.



In the above FSM, the marketplace coordinator begins in the initial state q_0, where only the `start` function is available to initialise the system. After initialisation, the coordinator transitions to state q_1, where function `makeOffer` is available to create an offer, moving the system to state q_2. In state q_2, two options are available: either `accept` the offer, which transitions to the final state q_3 representing a completed transaction, or `reject` the offer, which returns the system to state q_1 where new offers can be made. This state-based structure ensures that operations can only be performed in the correct sequence: offers must be made before they can be accepted or rejected, and the system cannot accept an offer that does not exist.

Role-based access control. While state-dependent function availability ensures that operations occur in the correct sequence, this alone is not sufficient to model a wide range of real-world scenarios. In many distributed systems, certain actions must be restricted to participants with specific roles or privileges. For instance, in the previous example with the marketplace coordinator, only the participant with the *Owner* role should be able to accept or reject an offer. Similarly, in a voting system, only registered voters should be permitted to cast votes during the voting period. These scenarios require that function availability depends not only on the current state, but also on the identity and permissions of the participant attempting to invoke the function.

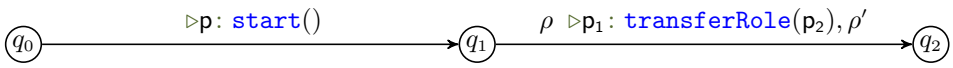
To address this need, we enrich our model with a *role-based access control mechanism*. In this approach, participants can be assigned *roles* that determine their permissions within each coordinator. It is important to emphasize that roles are *local* to each coordinator, not global to the entire system.

We model role-based constraints through *role assignments*, which are mappings that specify the relationship between participants and roles. Formally, a role assignment ρ is a function that takes a participant variable p and a role R, and returns a *role modality* that describes the participant's relationship to that role. There are three possible role modalities:

- **Has role** (■): The participant variable p must have role R for the transition to be enabled. This is written as $\rho(\mathsf{p}, R) = ■$. For instance, the `accept` function might require that the caller has the *Owner* role ($(\mathsf{p}, \mathit{Owner}) \mapsto ■$) when accepting an offer in the marketplace coordinator.
- **Does not have role** (□): The participant variable p must *not* have role R for the transition to be enabled. This is written as $\rho(\mathsf{p}, R) = □$.
- **Immaterial** (▣): It does not matter whether the participant variable p has role R or not. This is written as $\rho(\mathsf{p}, R) = ▣$. This modality is used when a function is available to all participants regardless of their roles, or when the role constraint of a participant is not relevant for a particular transition.

For the following examples, we fix $\mathsf{p}_1, \mathsf{p}_2$ as *participant variables*, $\mathit{Admin}, \mathit{User}$ as roles, and $\rho = \{(\mathsf{p}_1, \mathit{Admin}) \mapsto ■, (\mathsf{p}_2, \mathit{Admin}) \mapsto □\}$. The role assignment ρ specifies that participant variable p_1 has role Admin while participant variable p_2 does not have role Admin. We fix $\rho' = \{(\mathsf{p}_1, \mathit{Admin}) \mapsto □, (\mathsf{p}_2, \mathit{Admin}) \mapsto ■, (\mathsf{p}_1, \mathit{User}) \mapsto ■\}$ that will be applied after the transition. The role assignment ρ' specifies that participant variable p_1 loses role Admin and gains role User, while participant variable p_2 gains role Admin.

The following FSM illustrates how role-based constraints are enforced in transitions.



The coordinator begins in state q_0, where the `start` function is available to initialise the system. After initialisation, the system transitions to state q_1, where the `transferRole` function becomes available. However, this function is not accessible to all participants: it can only be successfully invoked when specific role constraints are satisfied (i.e., ρ placed before the action).

More precisely, for the `transferRole` transition to be enabled, two conditions must hold according to the role assignment ρ:

- The *caller* p_1 must have role *Admin*, ensuring that only authorized participants can transfer roles.
- The *parameter* p_2 must *not* have role *Admin*, enforcing that the parameter does not already have the Admin role.

The role-based access control mechanism in our model is *dynamic*, meaning that the roles associated with each participant can change during the execution, unlike static role systems where permissions are fixed at initialisation. Our model allows roles to be granted or revoked as the system evolves, enabling coordinators to model complex authorization scenarios where privileges change based on system events.

Role changes occur *after* a function call succeeds, as part of the transition's effect. The participants whose roles may be modified include both the *caller* and any *parameters*. This flexibility allows coordinators to model scenarios such as ownership transfer, where the caller loses a role while a parameter gains it, or role promotion, where a participant acquires additional privileges after performing certain actions.

To specify role updates, we use role assignments in the same way as role constraints, but with a different interpretation: while role constraints (written before the action) specify *requirements* that must be satisfied for the transition to occur, role updates (written after the action) specify the *new role assignments* that take effect after the transition completes.

In the above FSM, the role assignment ρ' in transition from q_1 to q_2 specifies three changes: Role *Admin* is revoked from p_1 (since $\rho'(\mathsf{p}_1, \mathit{Admin}) = \square$) while p_2 gains role *Admin* (since $\rho'(\mathsf{p}_2, \mathit{Admin}) = \blacksquare$) and p_1 gains role *User* (since $\rho'(\mathsf{p}_1, \mathit{User}) = \blacksquare$).

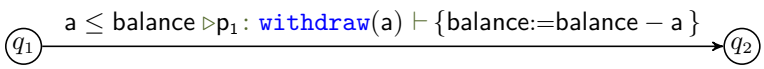
Data fields and state management. In addition to states and roles, coordinators can maintain *data fields* that store persistent information throughout the coordinator's execution. These fields are analogous to state variables in Ethereum smart contracts or instance variables in object-oriented programming languages like Java. Unlike states, which represent different phases of execution, data fields store actual values (such as numbers, strings, or other data structures) that can be read and modified by function calls.

Data fields enable coordinators to maintain and manipulate information that persists across multiple function invocations. For example, a banking coordinator might have a field that stores an account balance, a marketplace coordinator might maintain a field tracking the number of available items, or a voting coordinator might store the current vote count for each candidate. These fields can be read to check conditions and can be updated to reflect changes in the system's state.

To control when transitions can occur based on the values stored in data fields, we introduce *guards*—boolean conditions that must evaluate to true before a transition can be executed. Guards can reference data fields, function parameters, or combinations of both, allowing coordinators to enforce business logic constraints. For instance, a guard might check that a withdrawal amount does not exceed the current balance, or that an auction bid is higher than the current highest bid.

When a transition executes successfully, data fields can be updated through *assignments* that modify their values. Assignments use standard assignment notation, such as $\mathsf{amount}{:=}\mathsf{amount} + \mathsf{a}$ to increment a balance field amount by a parameter value a, or $\mathsf{amount}{:=}\mathsf{a}$ to set a field to a specific value. These updates occur *atomically* as part of the transition, ensuring that the coordinator's state remains consistent.

The following example illustrates how guards and assignments work together. Consider a coordinator with a data field $\mathsf{balance}$ representing an account balance. The transition:



where $\mathsf{a} \leq \mathsf{balance}$ is a guard and $\{\mathsf{balance}{:=}\mathsf{balance} - \mathsf{a}\}$ is an assignment, ensures that withdrawals can only occur when the requested amount does not exceed the current balance. Of course role constraints are omitted in the transition when they are the constant function mapping all elements of the domain of ρ to $\blacksquare$.

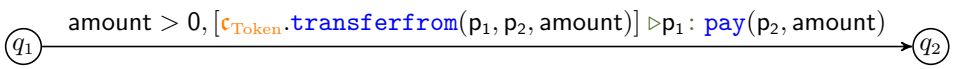
Interactions among coordinators. Real distributed systems rarely consist of isolated components; instead, they typically involve multiple coordinators that must interact and coordinate with each other to achieve complex behaviours. For instance, a payment processing coordinator might need to interact with a token coordinator to transfer funds, or a marketplace coordinator might call an escrow coordinator to hold funds during a transaction. These interactions are essential for modelling realistic distributed systems where functionality is distributed across multiple components.

To support such interactions, we extend our model with *call tries*—mechanisms that allow one coordinator to invoke functions on another coordinator as part of a transition. The term 'try' reflects the fact that these calls may succeed or fail, and the coordinator can specify different behaviours depending on the outcome. This enables coordinators to handle errors gracefully and model complex interaction patterns.

Call tries are specified within transitions by enriching the transition label with a list of external calls. Each call in this list specifies three pieces of information:

- the *identity* of the target coordinator (which uniquely identifies the target coordinator instance),
- the *operation* (function name) to invoke on the target coordinator,
- the *parameters* to pass to the operation, which can include participant variables, data values, or other expressions.

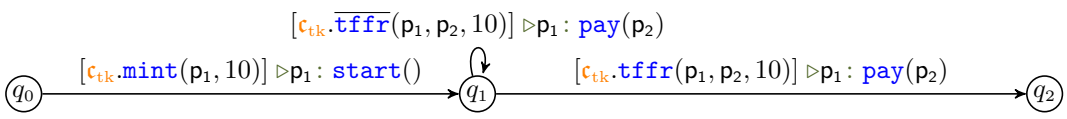
For example, consider a payment coordinator that needs to transfer tokens between participants. When processing a payment, the payment coordinator might call a token coordinator (identified as $\mathfrak{c}_{\text{Token}}$) to execute a `transferfrom` operation that moves tokens from one participant to another. This interaction allows the payment coordinator to leverage the functionality of the token coordinator without duplicating its own implementation.



The transition diagram above illustrates a concrete example of coordinator interaction. In this transition, the operation `pay` is invoked by participant p_1 with parameters p_2 (the recipient) and amount (the payment amount). As part of executing this transition, the coordinator makes a call try to another coordinator identified as $\mathfrak{c}_{\text{Token}}$, invoking its `transferfrom` operation to transfer amount tokens from participant p_1 (the caller) to recipient participant p_2 (the function parameter).

For this transition to succeed, multiple conditions must be satisfied simultaneously. First, the local guard $\mathsf{amount} > 0$ must evaluate to true, ensuring that the payment amount is positive. Second, any role-based constraints specified in the role assignment ρ must be met (in this case it is trivially the constant constraint mapping all elements of the domain of ρ to $\blacksquare$). Finally, the external call try to the token coordinator must succeed; if the token transfer fails (for example, due to insufficient balance or authorization issues), the entire transition fails and the coordinator remains in state q_1.

Our model also supports specifying different behaviours depending on whether an external call succeeds or fails. The following FSM illustrates how coordinators can handle both successful and failing call tries.



The execution of this coordinator begins in state q_0, where participant p_1 invokes the `start` function, making a call try to a token coordinator (identified as $\mathfrak{c}_{\text{tk}}$) to mint 10 tokens for participant p_1. This function makes a call try to a token coordinator (identified as $\mathfrak{c}_{\text{tk}}$) to mint 10 tokens for participant

p_1. If this call succeeds, the coordinator transitions to state q_1. From state q_1, the coordinator provides two distinct transitions for the `pay` operation, each handling a different outcome of the same external call. The first transition is a *loop transition* that remains in state q_1 when the call try to transfer 10 tokens from p_1 to p_2 fails. This is indicated by the bar notation $(-)$ in $\mathfrak{c}_{\mathrm{tk}}.\overline{\texttt{tffr}}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$, which specifies that this transition is only enabled when the external call fails. The second transition is a *forward transition* that moves to state q_2 when the same call try succeeds (indicated by the absence of the bar notation in $\mathfrak{c}_{\mathrm{tk}}.\texttt{tffr}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$).

Let us now put all these ingredients together in an example that outlines the intended behaviour of three coordinators.

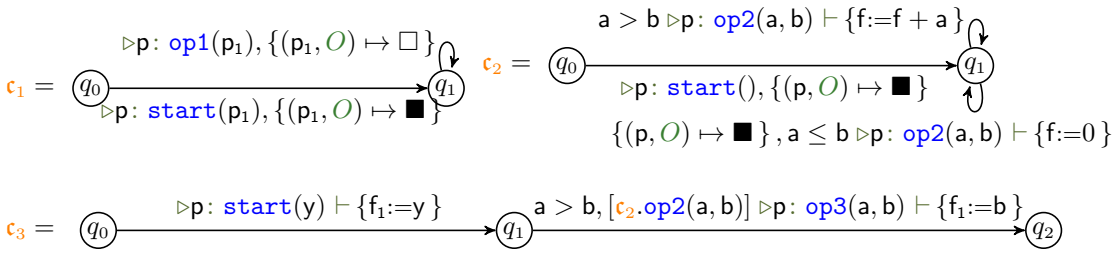


Figure III.1: Coordinators $\mathfrak{c}_1$, $\mathfrak{c}_2$ and $\mathfrak{c}_3$

The Figure III.1 presents three coordinators demonstrating how the various features of our model work together. Each coordinator includes a deployment operation (`start`) that initialises the system by transitioning from q_0 to q_1. Note that in transition labels, we omit elements when they are obvious or empty: guards when trivially true, role constraints when all modalities are ▣, and empty assignments or call tries.

Coordinator $\mathfrak{c}_1$ illustrates dynamic role management: `start` grants role O to parameter p_1, and `op1` can repeatedly revoke this role from the parameter participant, remaining in state q_1 with no guard constraints.

Coordinator $\mathfrak{c}_2$ demonstrates guards, role-based access control, and data field updates. It maintains field f and assigns role O to the participant invoking `start`. Operation `op2` takes parameters a and b with two behaviours: when the owner (with role O) invokes it with guard $\mathsf{a} \leq \mathsf{b}$, field f is reset to 0; when any participant invokes it with guard $\mathsf{a} > \mathsf{b}$, field f is incremented by a.

Coordinator $\mathfrak{c}_3$ showcases coordinator interactions through call tries. It maintains field f_1, initialised to y during deployment. Operation `op3` requires guard $\mathsf{a} > \mathsf{b}$ and a successful call try to operation `op2` of coordinator $\mathfrak{c}_2$; upon success, it updates f_1 to b and moves to q_2, illustrating how coordinators coordinate through composition.

III.2 Formal model

We now proceed to define our formal model. The foundation of our approach is the notion of Enhanced Data-Aware Machines (EDAMs), which we present in Section III.2.1. To provide the semantics for networks of EDAMs in Section III.2.3, we first introduce auxiliary notions related to role updating in Section III.2.2. All notation used throughout the document is summarised in Table III.1.

III.2.1 Enhanced Data-Aware Machines

This section introduces several auxiliary notions necessary for the formalization of EDAMs, with the formal definition of EDAMs given in Definition III.6.

Before presenting the formal model, we establish some notational conventions. Hereafter, when we declare a metavariable over a set, we implicitly allow indexed and primed versions of that metavariable to also range over the same set. For example, if x is a metavariable for the set X, then so are $x', x_i, x_1, x_2, \ldots$, all ranging over X. This convention simplifies our notation by avoiding the need to explicitly declare every variant of a metavariable. Given a function f, the *update*

$\mathcal{P}$	participants	$\mathfrak{p}$, $\mathfrak{q}$, ...	$\mathcal{T}_D$	data types	$\mathtt{t}, \mathtt{t}', \ldots$
$\mathcal{C}$	coordinators	$\mathfrak{c}, \mathfrak{c}', \ldots$	$\mathcal{T}_O$	operation types	$[\mathtt{t}, \mathtt{t}', \ldots]$
R	roles	$R, R', \ldots$	E	expressions	$\mathfrak{e}, \mathfrak{e}_i, \ldots$
F	fields	$\mathsf{f}, \mathsf{f}', \ldots$	$\mathcal{A}$	Σ-actions	$\alpha, \alpha_i, \ldots$
V	variables	$\mathsf{x}, \mathsf{y}, \ldots$	$\mathcal{R}$	role assignment	$\rho, \rho', \ldots$
O	operations	$\mathtt{op}, \mathtt{op}', \ldots$	Π	roles mapping	$\pi, \pi', \ldots$
T	typing environment	$\mathsf{T}, \mathsf{T}', \ldots$	A	assignments	$\mathsf{A}, \mathsf{A}', \ldots$
$\mathbb{Q}$	states	$q_0, q, q' \ldots$	Γ	network	$\eta, \eta', \ldots$
Σ	interfaces	$\Sigma, \Sigma', \ldots$	N	network configuration	$N, N', \ldots$
$\mathcal{T}$	types	$\mathtt{t}, [\mathtt{t}, \mathtt{t}', \ldots]$	Ω	environments	$\sigma, \sigma', \ldots$

Table III.1: Summary of notation

$f[x_1, \ldots, x_n \mapsto v_1, \ldots, v_n]$ is a new function that maps each x_i to the corresponding value v_i and behaves exactly as f for all other elements in the domain.

Our model centres around the concept of *participants*, which can be thought of as agents that trigger computations within the distributed system. A distinguished class of participants are *coordinators* (formally defined in Definition III.6), which are participants that expose a public interface consisting of operations and fields. While coordinators are a special type of participant, our model also admits participants that are not coordinators.[1]Participants coordinate with each other by invoking coordinators' operations or by reading their publicly available fields. We fix a finite set $\mathcal{C}$ of *coordinator identities* (ranged over by $\mathfrak{c}, \mathfrak{c}', \ldots$) which uniquely identify each coordinator instance in the system. We also fix a set $\mathcal{P}$ of *participant identities* (ranged over by $\mathfrak{p}, \mathfrak{p}', \ldots$) that properly contains $\mathcal{C}$, meaning that every coordinator identity is also a participant identity, but there may be additional participants that are not coordinators. We fix three finite sets of symbols: R for *roles* (ranged over by $R, R', \ldots$) which are used in access control, F for *fields* (ranged over by $\mathsf{f}, \mathsf{f}', \ldots$) which store the coordinator's persistent data, and V for *variables* (ranged over by $\mathsf{x}, \mathsf{x}', \ldots$) which appear in the coordinator's expressions and operations. We use R and f as metavariables over R and F respectively. Likewise, lowercase Latin letters in sans-serif font such as x, y, a, p, etc., are metavariables over V.

We also fix a finite set O of symbols for *operations*, with a distinguished symbol $\mathtt{start} \in \mathsf{O}$ that is used to deploy and initialise coordinators. We require that $\mathsf{F} \cap \mathsf{O} = \emptyset$ to ensure that field names and operation names do not conflict. We use $\mathtt{op}, \mathtt{op}'$ as metavariables over O to denote operations. To support type checking and ensure type safety, we introduce a set $\mathcal{T}_D$ of *data types*. While we do not fix the exact contents of $\mathcal{T}_D$, it represents the usual programming language types such as `bool` (booleans), `int` (integers), `str` (strings), `array` (arrays), `map` (maps or dictionaries), and so forth. We assume that $\mathtt{pt} \in \mathcal{T}_D$, where `pt` is a special data type representing participants. We define the set of *operation types* $\mathcal{T}_O$ as finite sequences of data types, that is $\mathcal{T}_O = \mathcal{T}_D^*$. Intuitively, an operation type models the parameter types of an operation, specifying what types of values must be passed when invoking the operation. Note that in our model, operations do not return any value; they are purely imperative actions that modify the state of the coordinator. Finally, we let $\mathcal{T} = \mathcal{T}_D \cup \mathcal{T}_O$ be the union of all data types and operation types. A comprehensive recap of all notations used in this paper can be found in Table III.1.

A *coordinator interface* serves as a specification that defines what roles, data fields, and operations are available in a coordinator, along with their respective types. Interfaces also specify the types of other coordinators' fields, variables, and operations that are referenced by the coordinator, which enables coordinators to interact with each other. In the following definition, we use a special notation to distinguish between the current coordinator and external coordinators. Pairs of the form $(\varepsilon, \_)$ refer to elements of the 'current coordinator', while pairs of the form $(\mathfrak{c}, \_)$ refer to elements of a coordinator $\mathfrak{c}$ that is different from the current coordinator (see Section III.2.3 for

[1]These non-coordinator participants can be thought of as, for example, human users or external systems that interact with coordinators.

details on coordinator networks).

▶ **Definition III.1.** *An* interface Σ *is a tuple* $(\mathsf{R}', \mathsf{F}', \mathsf{V}', \mathsf{O}', \mathsf{T}')$ *where:*

- $\mathsf{R}' \subseteq \mathsf{R}$ *is a set of roles,* $\mathsf{F}' \subseteq \mathsf{F}$ *is a set of fields,* $\mathsf{V}' \subseteq \mathsf{V}$ *is a set of variables, and* $\mathsf{O}' \subseteq \mathsf{O}$ *is a set of operations (with* $\texttt{start} \in \mathsf{O}'$*);*
- $\mathsf{T}' : (\mathcal{C} \cup \{\varepsilon\}) \times (\mathsf{F} \cup \mathsf{V} \cup \mathsf{O}) \rightharpoonup \mathcal{T}$ *is a* typing environment *where* ε *denotes the current coordinator, fields and variables are mapped to data types, and operations are mapped to operation types. We will often write* $\mathfrak{c}.\mathsf{f}$ *rather than* $(\mathfrak{c}, \mathsf{f})$*, and just* f *for* $(\varepsilon, \mathsf{f})$*.*

▶ **Example III.2.** Based on the coordinators in Fig. III.1, we define the following interfaces:
$\Sigma_1 = (\mathsf{R}_1, \mathsf{F}_1, \mathsf{V}_1, \mathsf{O}_1, \mathsf{T}_1)$ (for coordinator $\mathfrak{c}_1$):

$$\begin{array}{l} \mathsf{R}_1 = \{O\} \\ \mathsf{F}_1 = \emptyset \\ \mathsf{V}_1 = \{\mathsf{p}, \mathsf{p}_1\} \\ \mathsf{O}_1 = \{\texttt{start}, \texttt{op1}\} \end{array} \qquad \mathsf{T}_1 = \begin{cases} (\varepsilon, \mathsf{p}), (\varepsilon, \mathsf{p}_1) \mapsto \texttt{pt}; \\ (\varepsilon, \texttt{start}) \mapsto [\texttt{pt}]; \\ (\varepsilon, \texttt{op1}) \mapsto [\texttt{pt}] \end{cases}$$

$\Sigma_2 = (\mathsf{R}_2, \mathsf{F}_2, \mathsf{V}_2, \mathsf{O}_2, \mathsf{T}_2)$ (for coordinator $\mathfrak{c}_2$):

$$\begin{array}{l} \mathsf{R}_2 = \{O\}, \\ \mathsf{F}_2 = \{\mathsf{f}\}, \\ \mathsf{V}_2 = \{\mathsf{p}, \mathsf{a}, \mathsf{b}\}, \\ \mathsf{O}_2 = \{\texttt{start}, \texttt{op2}\}, \end{array} \qquad \mathsf{T}_2 = \begin{cases} (\varepsilon, \mathsf{f}), (\varepsilon, \mathsf{a}), (\varepsilon, \mathsf{b}) \mapsto \texttt{int}; \\ (\varepsilon, \mathsf{p}) \mapsto \texttt{pt}; \\ (\varepsilon, \texttt{start}) \mapsto []; \\ (\varepsilon, \texttt{op2}) \mapsto [\texttt{int}, \texttt{int}] \end{cases}$$

$\Sigma_3 = (\mathsf{R}_3, \mathsf{F}_3, \mathsf{V}_3, \mathsf{O}_3, \mathsf{T}_3)$ (for coordinator $\mathfrak{c}_3$):

$$\begin{array}{l} \mathsf{R}_3 = \emptyset \\ \mathsf{F}_3 = \{\mathsf{f}_1\} \\ \mathsf{V}_3 = \{\mathsf{p}, \mathsf{a}, \mathsf{b}, \mathsf{y}\} \\ \mathsf{O}_3 = \{\texttt{start}, \texttt{op3}\} \end{array} \qquad \mathsf{T}_3 : \begin{cases} (\varepsilon, \mathsf{f}_1), (\varepsilon, \mathsf{a}), (\varepsilon, \mathsf{b}), (\varepsilon, \mathsf{y}) \mapsto \texttt{int} \\ (\varepsilon, \mathsf{p}) \mapsto \texttt{pt} \\ (\varepsilon, \texttt{start}) \mapsto [\texttt{int}] \\ (\mathfrak{c}_2, \texttt{op2}), (\varepsilon, \texttt{op3}) \mapsto [\texttt{int}, \texttt{int}] \end{cases}$$

◇

From now on, we fix an interface $\Sigma = (\mathsf{R}', \mathsf{F}', \mathsf{V}', \mathsf{O}', \mathsf{T}')$.

To enable coordinators to express conditions and computations, we introduce a set E of *expressions*, which we leave unspecified in full detail. It suffices to assume that E contains several fundamental components. First, it includes field references of the form $(\{\varepsilon\} \cup \mathcal{C}) \times \mathsf{F}$, allowing expressions to read both internal fields (using ε) and external fields (using coordinator identities). Second, it includes a special constant $\mathfrak{sf}$ that denotes the identity of the current coordinator, which is useful for self-referential operations. Third, it includes global functions that can be used in any coordinator, such as max, min, sum, and similar mathematical and aggregate functions. Finally, it includes standard arithmetic expressions (addition, subtraction, multiplication, division) and logical expressions (conjunction, disjunction, negation, comparisons). We assume that all expressions are uniquely typed through a typing system defined on top of the typing environment T', and that $\mathfrak{sf}$ has type pt (the participant type).

An *assignment* is a function $\mathsf{A} : \mathsf{F} \to \mathsf{E}$ that specifies how fields should be updated. An important constraint is that A must be the identity function on all external fields, meaning that assignments can only modify fields belonging to the current coordinator, not fields of other coordinators. An assignment A can be conveniently written as a set of field updates $\{\mathsf{f}_1 := \mathfrak{e}_1, \ldots, \mathsf{f}_n := \mathfrak{e}_n\}$ where $\mathsf{f}_1, \ldots, \mathsf{f}_n$ are pairwise different fields. This notation means that $\mathsf{A}(\mathsf{f}_i) = \mathfrak{e}_i$ for each i from 1 to n, and $\mathsf{A}(\mathsf{f}') = \mathsf{f}'$ (the identity) for all fields f' that are not explicitly listed in the set. In other words, only the fields mentioned in the assignment are updated, while all other fields remain unchanged.

Upon invocations of their operations, coordinators execute *actions* that comprehensively specify the behaviour of the operation. An action includes several components: *guards* are boolean conditions that must be satisfied for the action to execute, *call tries* specify any required interactions with other coordinators, the operation being invoked is identified, and *field updates* (assignments) specify how the coordinator's fields should change as a result of the operation.

▶ **Definition III.3.** *A* call try *takes the form* $\mathfrak{c}.\texttt{op}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$ *or* $\mathfrak{c}.\overline{\texttt{op}}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$ *respectively representing the check that an invocation to an operation* op *of* $\mathfrak{c}$ *is successful or not. The set* $\mathcal{A}$ *of* Σ-actions *(ranged over by* α*) over* Σ *is the set of elements of the form*

$$\mathsf{g}, \mathsf{L} \triangleright \mathsf{p} \colon \texttt{op}\,(\mathsf{x}_1, \ldots, \mathsf{x}_n) \vdash \mathsf{A} \qquad \textit{with} \qquad n \geq 0 \tag{1}$$

where g *is a boolean expression (referred to as the* guard*),* L *is a possibly empty sequence of call tries,* $\texttt{op} \in \mathsf{O}'$*,* $\mathsf{T}'(\mathsf{p}) = \texttt{pt}$*, and the set* $\{\mathsf{p}, \mathsf{x}_1, \ldots, \mathsf{x}_n\} \subseteq \mathsf{V}$ *contains all variables occurring in the*

guard g*, expressions in* L*, or in the codomain of the assignment* A*. For* $\alpha \in \mathcal{A}$ *as in* (1)*, we define* $\mathtt{ptp}(\alpha) = \{\mathsf{p}\} \cup \{\mathsf{x}_i \mid 1 \leq i \leq n \wedge \mathsf{T}'(\mathsf{x}_i) = \mathtt{pt}\}$.

Role assignments serve two important purposes in our model. First, they control access to operations by specifying whether participants should or should not have certain roles before a transition can occur. Second, they specify how roles are updated after an operation call succeeds, enabling dynamic role management. We recall the three role modalities: ■ means that the participant must have the role, □ means that the participant must not have the role, and ▣ means that it is immaterial whether the participant has the role or not (the role constraint is not relevant for this transition).

▶ **Definition III.4.** *Let* $\Sigma = (\mathsf{R}', \mathsf{F}', \mathsf{V}', \mathsf{O}', \mathsf{T}')$*. A* role assignment ρ over Σ *(abbreviated* Σ*-role assignment) is a partial from* $\mathsf{V}' \times \mathsf{R}'$ *to* $\{■, □, ▣\}$ *such that for all* $(\mathsf{p}, R) \in \mathrm{dom}\,(\rho)$[2]*:* $\mathsf{T}'(\mathsf{p}) = \mathtt{pt}$ *(only participant variables can have role assignments); for* $\circledast \in \{■, □, ▣\}$*, we define* $\rho_{\circledast}(\mathsf{p}) = \{R \in \mathsf{R}' \mid \rho(\mathsf{p}, R) = \circledast\}$*. We say that* ρ *and* ρ' *are inconsistent if* $▣ \neq \rho(\mathsf{p}, R) \neq \rho'(\mathsf{p}, R) \neq ▣$ *for some* p, R*. Let* $\mathcal{R}$ *be the set of role assignments (over any* Σ*).*

For any pair of participant variable and role that is not explicitly listed in a role assignment ρ, the corresponding role modality is implicitly set to ▣. The role modality ▣ in the role assignment is the default modality, when in the role constraint, it means that it is immaterial, but when in the role assignment, it means that we do not perform any role update for this participant for this role.

▶ **Example III.5.** Consider a role assignment ρ such than $\rho(\mathsf{p}, O) = ■, \rho(\mathsf{p}, R) = ■, \rho(\mathsf{p}, B) = □, \rho(\mathsf{p}, U) = ▣$. We have $\rho_■(\mathsf{p}) = \{O, R\}, \rho_□(\mathsf{p}) = \{B\}, \rho_▣(\mathsf{p}) = \{U\}$. ◇

From now on, we often do not explicitly write the pairs (p, R) when $\rho(\mathsf{p}, R) = ▣$. E.g., ρ Example III.5 will be written as $\rho = \{(\mathsf{p}, O) \mapsto ■, (\mathsf{p}, R) \mapsto ■, (\mathsf{p}, B) \mapsto □\}$.

A coordinator is an automaton (finite state machine) whose transitions specify when operations are enabled and, upon execution, enforce role-based access control and update the coordinator's state.

▶ **Definition III.6.** *A* coordinator *(EDAM) over* Σ *(*Σ*-coordinator for short) is a tuple* $\langle \mathbb{Q}, q_0, \longrightarrow \rangle$ *where*

- $\mathbb{Q}$ *is a finite set of* states *with* $q_0 \in \mathbb{Q}$ *being the* initial *state;*
- $\longrightarrow \subseteq \mathbb{Q} \times (\mathcal{R} \times \mathcal{A} \times \mathcal{R}) \times \mathbb{Q}$ *is the* transition relation[3] *such that, for each transition* $q \xrightarrow{\langle \rho, \alpha, \rho' \rangle} q'$ *with* $\alpha = \mathsf{g}, \mathsf{L} \triangleright \mathsf{p} \colon \mathtt{op}\,(\mathsf{x}_1, \ldots, \mathsf{x}_n) \vdash \mathsf{A}$*:*
  - ρ, ρ' *are such that* $\mathrm{dom}\,(\rho) = \mathrm{dom}\,(\rho') = \mathtt{ptp}(\alpha) \times \mathsf{R}'$*;*
  - *if* $\mathtt{op} \neq \mathtt{start}$ *then* $q \neq q_0$ *and if* $\mathtt{op} = \mathtt{start}$ *then* $q = q_0$ *and* $\rho_■(\mathsf{x}) = \emptyset$ *for all* $\mathsf{x} \in \mathrm{dom}\,(\rho)$ *(participants do not have any role before the coordinator is started);*
  - *there is at least one transition* $q_0 \xrightarrow{\langle \rho, \alpha, \rho' \rangle} q'$ *such that* $\mathtt{start}$ *is the operation of* α*.*

We let $\mathfrak{t}$ range over transitions in the transition relation, and we let $\mathtt{src}(\mathfrak{t})$ denote the source state of transition $\mathfrak{t}$. To enable the evaluation of expressions and the execution of actions, we introduce the concept of *environments*. Let Ω be the set of *environments*, which are partial maps from the set V of variables to the set of values. An environment $\sigma \in \Omega$ assigns concrete values to variables, but it must respect the variables' types as specified by the typing environment. More precisely, for any variable p in the domain of σ such that $\mathsf{T}(\mathsf{p}) = \mathtt{pt}$ (i.e., p is a participant variable), we require that $\sigma(\mathsf{p}) \in \mathcal{P}$, ensuring that participant variables are mapped to actual participant identities.

[2] The domain of a partial function is the set of elements where it is defined.
[3] We write $q \xrightarrow{\langle \rho, \alpha, \rho' \rangle} q'$ instead of $(q, (\rho, \alpha, \rho'), q') \in \longrightarrow$.

III.2.2 Updating Roles

In our model, participants may acquire or lose roles dynamically, depending on their interactions with coordinators. Dynamic role management is essential for modelling realistic distributed systems where privileges change over time based on system events and user actions. For instance, a participant might gain administrator privileges after performing certain operations, or lose access rights when transferring ownership.

To formalise this dynamic behaviour, we introduce several constructions that are instrumental for handling dynamic roles in the operational semantics presented in Section III.2.3. Let Π be the set of *role mappings*, which are functions $\pi : \mathcal{P} \to 2^{\mathsf{R}}$ that assign sets of roles to participants. A role mapping π captures the current state of role assignments in a coordinator, specifying which roles each participant currently possesses. For example, $\pi(\mathfrak{alice}) = \{Admin, User\}$ indicates that participant identity $\mathfrak{alice}$ currently has both the Admin and User roles.

For transitions to be properly executed, role mappings, environments, and role assignments must be 'compatible' with each other. This compatibility ensures that the role constraints specified in a transition are consistent with the actual roles that participants have according to the current role mapping. The following definition formalises this compatibility relation. A pair $\langle \sigma, \pi \rangle$ *satisfies* a role assignment ρ, denoted by $\langle \sigma, \pi \rangle \models \rho$, when, if a participant variable should not have a role according to ρ, then that role must not be present in the participant's role set in π. Conversely, if a participant variable should have a role according to ρ, then that role must be present in the participant's role set in π.

▶ **Definition III.7.** *Given an environment $\sigma \in \Omega$, a role mapping $\pi \in \Pi$, and a role assignment $\rho \in \mathcal{R}$, the relation $\langle \sigma, \pi \rangle \models \rho$ holds if and only if, for all participant variables $(\mathsf{p}, \_) \in \mathrm{dom}\,(\rho)$:*

$$\pi(\sigma(\mathsf{p})) \cap \rho_{\square}(\mathsf{p}) = \emptyset \quad \textit{and} \quad \rho_{\blacksquare}(\mathsf{p}) \subseteq \pi(\sigma(\mathsf{p}))$$

This definition establishes when a role assignment is satisfied by a given environment and role mapping. The relation $\langle \sigma, \pi \rangle \models \rho$ holds when two conditions are met for every participant variable in the domain of the role assignment. The first condition $\pi(\sigma(\mathsf{p})) \cap \rho_{\square}(\mathsf{p}) = \emptyset$ ensures that the participant $\sigma(\mathsf{p})$ (the concrete participant identity to which the variable p maps) has *none* of the roles that are explicitly forbidden by the role assignment. In other words, the intersection of the participant's current roles and the set of forbidden roles must be empty, guaranteeing that no prohibited roles are present. The second condition $\rho_{\blacksquare}(\mathsf{p}) \subseteq \pi(\sigma(\mathsf{p}))$ ensures that the participant has *all* the required roles specified in the role assignment. This means that every role in $\rho_{\blacksquare}(\mathsf{p})$ must also be present in the participant's current role mapping $\pi(\sigma(\mathsf{p}))$.

▶ **Example III.8.** Consider an environment $\sigma = \{\mathsf{p} \mapsto \mathfrak{alice}\}$ that maps participant variable p to the concrete participant identity $\mathfrak{alice}$, and a role mapping $\pi = \{\mathfrak{alice} \mapsto \{O, B\}\}$ that assigns roles O and B to participant $\mathfrak{alice}$.

Then $\langle \sigma, \pi \rangle \models \rho$ holds for the role assignment $\rho = \{(\mathsf{p}, O) \mapsto \blacksquare\}$, which requires that participant variable p has role O. This is satisfied because: first, the intersection $\pi(\sigma(\mathsf{p})) \cap \rho_{\square}(\mathsf{p}) = \{O, B\} \cap \emptyset = \emptyset$ is empty (there are no forbidden roles), and second, the required role is present since $\rho_{\blacksquare}(\mathsf{p}) = \{O\} \subseteq \{O, B\} = \pi(\sigma(\mathsf{p}))$.

However, $\langle \sigma, \pi \rangle \not\models \rho'$ for the role assignment $\rho' = \{(\mathsf{p}, U) \mapsto \blacksquare\}$, which requires that p has role U. This fails because $\rho'_{\blacksquare}(\mathsf{p}) = \{U\} \not\subseteq \{O, B\} = \pi(\sigma(\mathsf{p}))$, meaning that participant $\mathfrak{alice}$ does not currently have the required role U. ◇

To model how roles change after a transition executes, we define the *role update function* $\mathsf{rupd} : \Omega \times \mathcal{R} \times \Pi \to \Pi$, which computes the update of a role mapping π given an environment σ and a role assignment ρ. Intuitively, for each participant variable in ρ, rupd modifies the roles of the participant in the current mapping π by adding all roles marked as $\blacksquare$ and removing all those marked as $\square$ for that variable, while leaving other roles unchanged. If a participant variable does not appear in ρ, its roles remain as in π.

The formal definition of the role update function is as follows:

▶ **Definition III.9.** *Given an environment σ, a role mapping π, and a role assignment ρ, the role update function* $\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{p})$ *for a participant* $\mathfrak{p}$ *is defined as:*

$$\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{p}) \quad = \quad \left(\pi(\mathfrak{p}) \cup \bigcup_{\mathsf{p}\in\{\mathsf{p}\mid(\mathsf{p},\_)\in\mathrm{dom}\,(\rho)\wedge\sigma(\mathsf{p})=\mathfrak{p}\,\}} \rho_{\blacksquare}(\mathsf{p})\right) \quad \setminus \quad \bigcup_{\mathsf{p}\in\{\mathsf{p}\mid(\mathsf{p},\_)\in\mathrm{dom}\,(\rho)\wedge\sigma(\mathsf{p})=\mathfrak{p}\,\}} \rho_{\square}(\mathsf{p})$$

If there are no participant variables in the domain of ρ that map to participant identity $\mathfrak{p}$ (i.e., $\sigma(\mathsf{p}) \neq \mathfrak{p}$ for all $(\mathsf{p},\_) \in \mathrm{dom}\,(\rho)$), then both unions in the definition are empty, resulting in $\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{p}) = \pi(\mathfrak{p})$. This is the identity case, where the participant identity's roles remain unchanged. Otherwise, the roles are updated by adding all roles that participant variables mapping to $\mathfrak{p}$ should have (via $\rho_{\blacksquare}$) and removing all roles that they should not have (via $\rho_{\square}$).

▶ **Example III.10.** Consider $\rho = \{(\mathsf{p}_1, O) \mapsto \blacksquare, (\mathsf{p}_2, B) \mapsto \blacksquare, (\mathsf{p}_3, B) \mapsto \blacksquare, (\mathsf{p}_3, O) \mapsto \square\,\}$ and environment $\sigma = \{\mathsf{p}_1 \mapsto \mathfrak{p}, \mathsf{p}_2 \mapsto \mathfrak{p}, \mathsf{p}_3 \mapsto \mathfrak{q}\,\}$ and $\pi = \mathfrak{r} \mapsto \begin{cases} \{O\,\} & \text{if } \mathfrak{r} \in \{\mathfrak{p}, \mathfrak{q}\} \\ \emptyset & \text{otherwise} \end{cases}$

- $\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{p}) = \pi(\mathfrak{p}) \cup \{O, B\,\} \setminus \emptyset = \{O, B\,\}$ (both p_1 and p_2 map to $\mathfrak{p}$)
- $\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{q}) = \pi(\mathfrak{q}) \cup \{B\,\} \setminus \{O\,\} = \{O\,\} \cup \{B\,\} \setminus \{O\,\} = \{B\,\}$ (only p_3 maps to $\mathfrak{q}$)
- $\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{r}) = \pi(\mathfrak{r}) = \emptyset$ (for all $\mathfrak{r} \notin \{\mathfrak{p}, \mathfrak{q}\,\}$). ◇

An important observation is that role updates are applied sequentially: first, all roles marked as ■ are added, then all roles marked as □ are removed. This means that if the same role is both granted and revoked for participant variables mapping to the same participant, the order of operations does not matter. For instance, consider a situation where $\pi(\mathfrak{p}) = \{O\,\}$ and the role assignment is $\rho = \{(\mathsf{p}_1, O) \mapsto \square, (\mathsf{p}_2, O) \mapsto \blacksquare\,\}$ with both $\sigma(\mathsf{p}_1) = \sigma(\mathsf{p}_2) = \mathfrak{p}$. When we apply the update, we get $\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{p}) = \{O\,\} \cup \{O\,\} \setminus \{O\,\} = \emptyset$. This creates a conflict: p_1 requires that $\mathfrak{p}$ should not have the O role, while p_2 requires that $\mathfrak{p}$ should have it, yet both variables map to the same participant. To detect and prevent such conflicts, we require that the relation $\langle\sigma, \mathsf{rupd}(\sigma,\pi,\rho)\rangle \models \rho$ must hold after applying the role update. This ensures that after applying the role update, the resulting role mapping is consistent with the role assignment itself. In other words, for participant variables mapped to the same identity $\mathfrak{p}$, the role assignment ρ cannot simultaneously require and forbid the same role, preventing $\mathfrak{p}$ from having contradictory role obligations.

▶ **Example III.11.** Consider the situation described above:
$\pi(\mathfrak{p}) = \{O\,\}$, $\rho = \{(\mathsf{p}_1, O) \mapsto \square, (\mathsf{p}_2, O) \mapsto \blacksquare\,\}$, and $\sigma(\mathsf{p}_1) = \sigma(\mathsf{p}_2) = \mathfrak{p}$.
After applying the role update, we have $\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{p}) = \{O\,\} \cup \{O\,\} \setminus \{O\,\} = \emptyset$.
To check whether $\langle\sigma, \mathsf{rupd}(\sigma,\pi,\rho)\rangle \models \rho$ holds, we need to verify the conditions of Definition III.7:

- For p_1: we need $\mathsf{rupd}(\sigma,\pi,\rho)(\sigma(\mathsf{p}_1)) \cap \rho_{\square}(\mathsf{p}_1) = \emptyset$. Since $\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{p}) = \emptyset$ and $\rho_{\square}(\mathsf{p}_1) = \{O\,\}$, we have $\emptyset \cap \{O\,\} = \emptyset$, which holds.
- For p_2: we need $\rho_{\blacksquare}(\mathsf{p}_2) \subseteq \mathsf{rupd}(\sigma,\pi,\rho)(\sigma(\mathsf{p}_2))$. Since $\rho_{\blacksquare}(\mathsf{p}_2) = \{O\,\}$ and $\mathsf{rupd}(\sigma,\pi,\rho)(\mathfrak{p}) = \emptyset$, we have $\{O\,\} \not\subseteq \emptyset$, which fails.

Therefore, $\langle\sigma, \mathsf{rupd}(\sigma,\pi,\rho)\rangle \not\models \rho$, demonstrating that the role update results in an inconsistent state where the role assignment cannot be satisfied by the updated role mapping. ◇

III.2.3 Networks

A *network* represents a collection of coordinators that can interact with each other through call tries, enabling complex distributed system behaviours through composition. Each coordinator in a network maintains its own local state, role mappings, and environment, preserving encapsulation and modularity. However, coordinators can communicate and synchronize with each other by invoking operations through call tries, allowing them to coordinate their behaviour and share information in a controlled manner. The *network configuration* tracks the current state of all coordinators in the network, including their current states, role mappings, environments, and execution flags.

▶ **Definition III.12.** *For $1 \leq i \leq k$, let each $\Sigma_i = (\mathsf{R}_i, \mathsf{F}_i, \mathsf{V}_i, \mathsf{O}_i, \mathsf{T}_i)$ be a coordinator interface. A* network *η over $\Sigma_1, \ldots, \Sigma_k$ is a finite map such that:*
i) $\mathrm{dom}\,(\eta) = \{\mathfrak{c}_i \mid 1 \leq i \leq k\} \subseteq \mathcal{C}$, *and*
ii) each $1 \leq i \leq k$ with $\eta(\mathfrak{c}_i) = \langle \mathbb{Q}_i, q_{0_i}, \longrightarrow_i \rangle$ is a Σ_i-coordinator where
$\mathsf{T}_i(\mathfrak{c}_j, \mathsf{x}) = \mathsf{T}_j(\varepsilon, \mathsf{x})$ *for each $1 \leq j \neq i \leq k$.*
A configuration *of a network η is a map $N : \mathrm{dom}\,(\eta) \to \{\langle q_i, \pi_i, \sigma_i, \omega_i \rangle \mid 1 \leq i \leq k\}$ where:*
i) $q_i \in \mathbb{Q}_i$ is the current state *of the coordinator $\eta(\mathfrak{c}_i)$,*
ii) $\pi_i \in \Pi$ assigns roles to participants in the coordinator $\eta(\mathfrak{c}_i)$,
iii) $\sigma_i \in \Omega$ is the current environment of coordinator $\eta(\mathfrak{c}_i)$,
iv) $\omega_i \in \{\mathsf{true}, \mathsf{false}\}$ is an auxiliary flag, called read-only mode.

We let Γ be the set of all possible networks over $\Sigma_1, \ldots, \Sigma_k$. We require that coordinators in read-only mode cannot be invoked, but their fields may be accessed. A coordinator is set to read-only mode in the network configuration before starting the invocation of its list of call tries, by setting $\omega_i = \mathsf{true}$. We further require that a coordinator $\eta(\mathfrak{c}_i)$ cannot reference its own fields or operations using the notation $\mathfrak{c}_i.\mathsf{f}$ or $\mathfrak{c}_i.\mathsf{op}$; instead, this coordinator must use f or op (what corresponds to $(\varepsilon, \mathsf{f})$ or $(\varepsilon, \mathsf{op})$).

▶ **Definition III.13** (Restrictions to coordinators)**.** *In a network η over $\Sigma_1, \ldots, \Sigma_k$, for any coordinator $\eta(\mathfrak{c}_i)$ with interface $\Sigma_i = (\mathsf{R}', \mathsf{F}', \mathsf{V}', \mathsf{O}', \mathsf{T}_i)$, it holds that*
$(\mathfrak{c}_i, \mathsf{f}) \notin \mathrm{dom}\,(\mathsf{T}_i)$ *for all* $\mathsf{f} \in \mathsf{F}'$ *and* $(\mathfrak{c}_i, \mathsf{op}) \notin \mathrm{dom}\,(\mathsf{T}_i)$ *for all* $\mathsf{op} \in \mathsf{O}'$.

The evolution of a network starts when a participant $\mathfrak{p}$ (the *caller*) invokes an operation op, with parameters $\mathfrak{v}_1, \ldots, \mathfrak{v}_n$, of a coordinator $\mathfrak{c}$. In our model this corresponds to an attempt to fire a transition. If such execution is successful, the role mapping and the environment of $\mathfrak{c}$ are updated, together with those of the coordinators involved by call tries. We formalise this with a labelled transition semantics, inductively defined by the rules in Table III.2, whose configurations are network configurations N and labels take the form $\mathfrak{p} : \mathfrak{c}.\mathsf{op}(\mathfrak{v}_1, \ldots, \mathfrak{v}_n)$. A label refers to a specific invocation of a coordinator's operation in which all symbolic parameters—including data values and participants—have been instantiated with concrete values. For example, in Fig. III.1, an invocation of operation op2 of coordinator $\mathfrak{c}_2$ by participant $\mathfrak{alice}$, with actual arguments 2 and 3, would correspond to a label such as $\mathfrak{alice} : \mathfrak{c}_2.\mathsf{op2}(2, 3)$.

We assume an evaluation function $[\![\_]\!]_{N,\sigma,\pi}$ for expressions; the network N is used to evaluate fields of other coordinators, while σ and π are used to evaluate fields and variables of the current coordinator. We use this function to evaluate expressions used in assignments and call tries; in particular, $[\![\mathsf{A}]\!]_{N,\sigma,\pi} = \mathsf{f} \mapsto [\![\mathsf{A}(\mathsf{f})]\!]_{N,\sigma,\pi}$ for all $\mathsf{f} \in \mathrm{dom}\,(\mathsf{A})$. Moreover, $[\![\mathsf{L}]\!]_{N,\sigma,\pi} = \epsilon$ if $\mathsf{L} = \epsilon$ and $[\![\mathsf{L}]\!]_{N,\sigma,\pi} = \mathfrak{c}_\mathfrak{p}.\mathsf{op}([\![\mathfrak{e}_1]\!]_{N,\sigma,\pi}, \ldots, [\![\mathfrak{e}_n]\!]_{N,\sigma,\pi}); \mathsf{L}'$ if $\mathsf{L} = \mathfrak{c}_\mathfrak{p}.\mathsf{op}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n); \mathsf{L}'$.

The semantic rule [TR] formally specifies the execution of a single transition in the EDAM coordination model: a participant invokes an operation of a coordinator instance in a network. This rule captures the fundamental semantics of local state evolution in the presence of role-based constraints, guards, and potential call tries. Intuitively, consider a coordinator instance currently in state q, with an outgoing transition $q \xrightarrow{\langle \rho, \mathsf{g}, \mathsf{L} \,\triangleright\, \mathsf{p}:\, \mathsf{op}(\mathsf{x}_1, \ldots, \mathsf{x}_n) \,\vdash\, \mathsf{A}, \rho' \rangle} q'$. The [TR] rule states that the invocation of coordinator $\mathfrak{c}$ with a label $\mathfrak{q} : \mathfrak{c}.\mathsf{op}(\mathfrak{v}_1, \ldots, \mathfrak{v}_n)$ can only be performed if the following conditions hold:

(i) Coordinator availability (left side of the transition in the conclusion of the rule). The coordinator with identity $\mathfrak{c}$ must be present in the network N and not in read-only mode (i.e., $\omega = \mathsf{false}$).
(ii) Transition existence (top premise). A transition $q \xrightarrow{\langle \rho, \mathsf{g}, \mathsf{L} \,\triangleright\, \mathsf{p}:\, \mathsf{op}(\mathsf{x}_1, \ldots, \mathsf{x}_n) \,\vdash\, \mathsf{A}, \rho' \rangle} q'$ should exist in the coordinator's transition relation, where the action $\mathsf{g}, \mathsf{L} \,\triangleright\, \mathsf{p}:\, \mathsf{op}\,(\mathsf{x}_1, \ldots, \mathsf{x}_n) \,\vdash\, \mathsf{A}$ matches the invoked operation op.
(iii) Type-correctness of parameters (left premise in the second line). All actual parameters $\mathfrak{v}_1, \ldots, \mathfrak{v}_n$ must be well-typed with respect to the types $\mathsf{T}'(\mathsf{x}_1), \ldots, \mathsf{T}'(\mathsf{x}_n)$ declared for the operation.
(iv) Environment update (central premise in the second line). The execution of the transition updates the environment σ of the coordinator $\mathfrak{c}$ to σ' by binding the distinguished variable p to the caller identity $\mathfrak{q}$, and each formal parameter x_i to its corresponding actual argument $\mathfrak{v}_i$, this

$$\frac{\begin{array}{c} q \xrightarrow{\langle \rho, \mathsf{g}, \mathsf{L} \,\triangleright\, \mathsf{p}:\, \mathsf{op}(\mathsf{x}_1,\ldots,\mathsf{x}_n) \,\vdash\, \mathsf{A}, \rho' \rangle} q' \\ \bigwedge_{1\le i\le n} \mathfrak{v}_i \text{ has type } \mathsf{T}'(\mathsf{x}_i) \quad \sigma' = \sigma[\mathsf{p}, \mathsf{x}_1, \ldots, \mathsf{x}_n \mapsto \mathsf{q}, \mathfrak{v}_1, \ldots, \mathfrak{v}_n] \quad [\![g]\!]_{N,\sigma',\pi} \\ \mathsf{rupd}(\sigma', \pi, \rho') = \pi' \quad \langle \pi, \sigma' \rangle \models \rho \quad \langle \pi', \sigma' \rangle \models \rho' \\ \mathfrak{c} : \langle q, \pi, \sigma', \mathsf{true} \rangle \mid N \xrightarrow[\pi,\sigma']{\mathfrak{c}:[\![\mathsf{L}]\!]_{N,\pi,\sigma'}} \mathfrak{c} : \langle q, \pi, \sigma', \mathsf{true} \rangle \mid N' \end{array}}{\mathfrak{c} : \langle q, \pi, \sigma, \mathsf{false} \rangle \mid N \xrightarrow{\mathsf{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} \mathfrak{c} : \langle q', \pi', [\![\mathsf{A}]\!]_{N',\sigma',\pi}, \mathsf{false} \rangle \mid N'} \text{ [TR]}$$

$$\frac{}{N \xrightarrow[\pi,\sigma]{\mathsf{q}:\epsilon} N} \text{ [EMPTY]} \qquad \frac{N \not\xrightarrow{\mathsf{q}:\mathfrak{c}_\mathsf{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} \qquad N \xrightarrow[\pi,\sigma]{\mathsf{q}:[\![\mathsf{L}]\!]_{N,\pi,\sigma}} N'}{N \xrightarrow[\pi,\sigma]{\mathsf{q}:\mathfrak{c}_\mathsf{p}.\overline{\mathsf{op}}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n);\mathsf{L}} N'} \text{ [CALL]}$$

$$\frac{N \xrightarrow{\mathsf{q}:\mathfrak{c}_\mathsf{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N'' \qquad N'' \xrightarrow[\pi,\sigma]{\mathsf{q}:[\![\mathsf{L}]\!]_{N'',\pi,\sigma}} N'}{N \xrightarrow[\pi,\sigma]{\mathsf{q}:\mathfrak{c}_\mathsf{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n);\mathsf{L}} N'} \text{ [SEQ]}$$

Table III.2: Network transition rules

update is not yet applied to the network configuration N as it is temporary until the transition is completely executed.

(v) Guard evaluation (right premise in the second line). The guard g of the transition must evaluate to true in the context of the updated environment σ' and the current role mapping π and the network configuration N (needed for external fields). Guards act as enabling conditions that restrict the applicability of transitions.

(vi) Roles update (left premise in the third line). If the transition (top left premise) prescribes an update to the role mapping (from ρ to ρ'), then the new role mapping π' is derived from applying the role update function rupd to the updated environment σ' and role mapping π and ρ'. Updates will be applied to the network configuration N after the transition is completely executed.

(vii) Role consistency (central and right premises in the third line). The current role mapping π together with the updated environment σ' must satisfy the pre-transition role constraints ρ (i.e., $\langle \pi, \sigma' \rangle \models \rho$), and the post-transition role mapping π' together with σ' must satisfy the post-transition role update ρ' (i.e., $\langle \pi', \sigma' \rangle \models \rho'$).

(viii) Inter-coordinator calls (bottom premise). The last premise ensures that any call tries L specified in the transition are executed in the context of the updated environment σ' and current role mapping π. Before invoking the call tries, the coordinator is added back to the network configuration in read-only mode, that is $\mathfrak{c} : \langle q, \pi, \sigma', \mathsf{true} \rangle \mid N$ in the premise as well as after the call tries are executed, that is $\mathfrak{c} : \langle q, \pi, \sigma', \mathsf{true} \rangle \mid N'$. A detailed explanation is provided below, as the premise uses the transition relation inductively defined by the subsequent rules [EMPTY], [CALL], and [SEQ], which we present next.

Call tries are executed in the network state N with two arguments: the role mapping π and the environment σ of the coordinator $\mathfrak{c}$ (i.e., $\xrightarrow[\pi,\sigma]{\mathfrak{c}:[\![\mathsf{L}]\!]_{N,\pi,\sigma}}$; here we use the notation $\xrightarrow[\sigma,\pi]{}$ for network execution with two arguments. These arguments will be used later for parameters evaluation of the list of calls L). Two distinct cases arise depending on the presence of call tries in the transition:

- *Without call tries.* If the transition does not have call tries, updates made to fields and roles are applied atomically. Specifically, the assignments A of the transition are evaluated in the updated environment σ' (i.e., $[\![\mathsf{A}]\!]_{N',\sigma',\pi}$), and the coordinator instance moves to state q' under the new role mapping π' while read-only mode is set to false in the network configuration.

- *With call tries.* If the transition has a non-empty sequence of call tries L, then execution proceeds in two phases. In the first phase, the coordinator instance is placed in a *read-only* mode (indicated by $\omega = \mathsf{true}$) in the network configuration and associated with the environment σ' that binds caller and argument values but does not yet include the assignments A. The coordinator instance then becomes the caller of its call tries sequence (each coordinator is the caller of its own call tries sequence). Parameters in the sequences of call tries L are not evaluated all at once, $[\![\mathsf{L}]\!]_{N,\pi,\sigma'}$ only evaluates the parameters of the first call try of the sequence in the network state N. The evaluation of each remaining call is performed in the updated network state by the execution of the previous call in the sequence. The execution of the sequences of calls tries L is governed by three auxiliary rules that handle different aspects of call sequence processing:
  - [EMPTY]: Handles the base case when the call sequence is empty (ϵ). This rule allows the network to remain unchanged when there are no calls to process.
  - [CALL]: Handles calls tries that do not progress the network configuration due to expected failure of call tries with the form $\mathsf{c}_\mathsf{p}.\overline{\mathsf{op}}(\mathfrak{v}_1, \ldots, \mathfrak{v}_n)$. The [CALL]rule discards any changes made to the network configuration (indicated by $N \xrightarrow{\mathsf{q}:\mathsf{c}_\mathsf{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} \not$) and continues processing the remaining sequence where the parameters of the calls are evaluated in the network state N, that is $[\![\mathsf{L}]\!]_{N,\pi,\sigma}$.
  - [SEQ]: Handles successful call execution in sequences. When the head call $\mathsf{c}_\mathsf{p}.\mathsf{op}(\mathfrak{v}_1, \ldots, \mathfrak{v}_n)$ of the sequence L can be executed successfully, this rule processes it first (transitioning from N to N''), then continues with the remaining sequence L from the resulting configuration N'' where the parameters of the calls are evaluated in the network state N'', that is $[\![\mathsf{L}]\!]_{N'',\pi,\sigma}$.

  Once all call tries have been resolved, the system resumes from the suspended configuration, re-applies the assignments A, and finalises the state change as in *without call tries* case.

The combination of [TR]with the call-sequence rules ensures that the execution model enjoys *atomicity*: all call tries must be fully evaluated before the local state of the coordinator is updated.

▶ **Example III.14.** Let us consider a network with coordinators c_2 and c_3 of Fig. III.1 with initial configuration $N_0 = \{\mathsf{c}_2 \mapsto \langle q_1, \pi_2, \sigma_2, \mathsf{false}\rangle, \quad \mathsf{c}_3 \mapsto \langle q_1, \pi_3, \sigma_3, \mathsf{false}\rangle\}$ with environments and roles:

$$\sigma_2 = \{\mathsf{p} \mapsto \mathfrak{p},\ \mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto 0,\ \mathsf{f} \mapsto 5\}$$
$$\sigma_3 = \{\mathsf{p} \mapsto \mathfrak{p},\ \mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto 0,\ \mathsf{f}_1 \mapsto 0\}$$
$$\pi_2(\mathfrak{p}) = \{O\}, \quad \pi_3(\mathfrak{p}) = \emptyset$$

Now, participant $\mathfrak{p}$ initiates the call: $\mathfrak{p} : \mathsf{c}_3.\mathsf{op3}(5,3)$ corresponding to a label for the transition $\mathsf{a} > \mathsf{b}, [\mathsf{c}_2.\mathsf{op2}(\mathsf{a},\mathsf{b})]\ \triangleright \mathsf{p}\colon \mathsf{op3}(\mathsf{a},\mathsf{b}) \vdash \{\mathsf{f}_1{:=}\mathsf{b}\}$ of c_3. We now construct the derivation using semantic rules.

$$\frac{\begin{array}{c} q_1 \xrightarrow{\langle \mathsf{a}>\mathsf{b}\ \triangleright \mathsf{p}\colon\ \mathsf{op2}(\mathsf{a},\mathsf{b})\ \vdash \{\mathsf{f}:=\mathsf{f}+\mathsf{a}\}\rangle} q_1 \quad [\![a > b]\!]_{N_1,\sigma_2',\pi_2} = \mathsf{true} \quad \mathsf{op2} \neq \mathtt{start} \\ \sigma_2' = \sigma_2[\mathsf{p},\mathsf{a},\mathsf{b} \mapsto \mathsf{c}_3, 5, 3] \quad \langle \pi_2, \sigma_2'\rangle \models \{\,\} \quad \mathsf{rupd}(\sigma'_2, \pi_2, \rho') = \pi_2' \\ \langle \pi_2', \sigma_2'\rangle \models \{\,\} \quad N_1 = \{\mathsf{c}_2 \mapsto \langle q_1, \pi_2, \sigma_2', \mathsf{true}\rangle, \mathsf{c}_3 \mapsto \langle q_1, \pi_3, \sigma_3, \mathsf{true}\rangle\,\} \quad \dfrac{}{N_1 \xrightarrow[\sigma,\pi]{\mathsf{c}_3:\epsilon} N_1} \end{array}}{\mathsf{c}_2 : \langle q_1, \pi_2, \sigma_2, \mathsf{false}\rangle \mid \{\mathsf{c}_3 \mapsto \langle q_1, \pi_3, \sigma_3, \mathsf{true}\rangle\,\} \xrightarrow[\sigma,\pi]{\mathsf{c}_3:\mathsf{c}_2.\mathsf{op2}(5,3)} \mathsf{c}_2 : \langle q_1, \pi_2', [\![\mathsf{f} := \mathsf{f} + \mathsf{a}]\!]_{N_1,\sigma_2',\pi_2}, \mathsf{false}\rangle \mid \{\mathsf{c}_3 \mapsto \langle q_2, \pi_3, \sigma_3, \mathsf{true}\rangle\,\}}$$

$$\frac{\begin{array}{c} q_1 \xrightarrow{\langle \mathsf{a}>\mathsf{b}, [\mathsf{c}_2.\mathsf{op2}(\mathsf{a},\mathsf{b})]\ \triangleright \mathsf{p}\colon\ \mathsf{op3}(\mathsf{a},\mathsf{b})\ \vdash \{\mathsf{f}_1:=\mathsf{b}\}\rangle} q_2 \quad [\![a > b]\!]_{N_0,\sigma_3',\pi_3} = \mathsf{true} \quad \mathsf{op3} \neq \mathtt{start} \\ \sigma_3' = \sigma_3[\mathsf{p},\mathsf{a},\mathsf{b} \mapsto \mathfrak{p}, 5, 3] \quad \langle \pi_3, \sigma_3'\rangle \models \{\,\} \\ \mathsf{rupd}(\sigma'_3, \pi_3, \{\,\}) = \pi_3' \quad \langle \pi_3', \sigma_3'\rangle \models \{\,\} \quad N_1 = \{\mathsf{c}_2 \mapsto \langle q_1, \pi_2, \sigma_2', \mathsf{false}\rangle, \mathsf{c}_3 \mapsto \langle q_2, \pi_3, \sigma_3', \mathsf{true}\rangle\,\} \\ \mathsf{c}_3 : \langle q_1, \pi_3, \sigma_3', \mathsf{true}\rangle \mid N_0 \xrightarrow[\sigma,\pi]{\mathsf{c}_3:\mathsf{c}_2.\mathsf{op2}(5,3)} \mathsf{c}_3 : N_1[\text{from rule above where } \sigma_2 \text{ udated to } \sigma_2'] \end{array}}{\mathsf{c}_3 : \langle q_1, \pi_3, \sigma_3, \mathsf{false}\rangle \mid N_0 \xrightarrow{\mathfrak{p}:\mathsf{c}_3.\mathsf{op3}(5,3)} \mathsf{c}_3 : \langle q_2, \pi_3', [\![\mathsf{f}_1 := \mathsf{b}]\!]_{N_0,\sigma_3',\pi_3}, \mathsf{false}\rangle \mid N_1}$$

Final Configuration:

$$N_1 = \{\mathfrak{c}_2 \mapsto \langle q_1, \pi_2, \sigma_2', \mathsf{false}\rangle, \quad \mathfrak{c}_3 \mapsto \langle q_2, \pi_3, \sigma_3', \mathsf{false}\rangle\}$$

with:

$$\sigma_2' = \{\mathsf{p} \mapsto \mathfrak{c}_3,\ \mathsf{a} \mapsto 5,\ \mathsf{b} \mapsto 3,\ \mathsf{f} \mapsto 10\}$$
$$\sigma_3' = \{\mathsf{p} \mapsto \mathfrak{p},\ \mathsf{a} \mapsto 5,\ \mathsf{b} \mapsto 3,\ \mathsf{f}_1 \mapsto 3\}$$
$$\pi_3' = \pi_3, \quad \pi_2' = \pi_2$$

◇

The *initial configuration* of a network η is the state of the network after the execution of a `start` operation of each coordinator in the network. To ensure that each operation invocation from a given network configuration produces a unique successor configuration, the network must be *deterministic*.

▶ **Definition III.15.** *A network η is said to be deterministic for all* reachable *network configuration N, all $\mathfrak{c} \in \mathcal{C}$ and for all participants $\mathfrak{q} \in \mathcal{P}$ if:*

$$N \xrightarrow{\mathfrak{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N_1 \wedge N \xrightarrow{\mathfrak{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N_2 \implies N_1 = N_2.$$

Given an initial network configuration N_0, if $N_0 \xrightarrow{\star} N_1$ then N_1 is reachable.

Determinism is necessary to preserve behavioural consistency between EDAM specifications and generated smart contract code, reflecting the execution semantics of blockchain platforms where all nodes must agree on outcomes. Without determinism, reproducibility, verification, and safe deployment would be compromised.

While determinism ensures that networks produce unique results, we also need to ensure that coordinators themselves are structured in a way that prevents ambiguous transitions. A coordinator is *well-formed* when, for any state and operation, multiple transitions from that state with the same operation are sufficiently distinguished to avoid ambiguity. Specifically, if two transitions from the same state with the same operation have guards that could both be satisfied simultaneously (their conjunction is not inconsistent), then they must be distinguishable through either conflicting role constraints or through their call tries, where one expects a call to succeed and the other expects it to fail. This well-formedness condition ensures that when an operation is invoked, there is at most one transition that can be taken, guaranteeing deterministic behaviour at the coordinator level.

▶ **Definition III.16.** *An EDAM is* wellformed *when for all states q and operations op such that for each distinct transitions $\mathsf{t}_i = \langle q, \rho_1, \mathsf{g}_i, [\mathsf{L}_i] \triangleright \mathsf{p} : \mathsf{op}(), \rho_1', q_i'\rangle$ and $\mathsf{t}_j = \langle q, \rho_2, \mathsf{g}_j, [\mathsf{L}_j] \triangleright \mathsf{p} : \mathsf{op}(), \rho_2', q_j'\rangle$ we have that if the conjunction of g_i and g_j is not inconsistent then either ρ_1 and ρ_2 are inconsistent or, for some L, $\mathfrak{c}$, and op: (i) g_i and g_j are equivalent, (ii) $\rho_1 = \rho_2$, (iii) $\mathsf{L}, \mathfrak{c}.\mathsf{op}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$ is a prefix of L_i and (iv) $\mathsf{L}, \mathfrak{c}.\overline{\mathsf{op}}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$ is a prefix of L_j.*

The relationship between well-formedness and determinism is fundamental to our model. Well-formedness ensures that individual coordinators have no ambiguous transitions, while determinism ensures that networks produce unique results.

▶ **Lemma III.17.** *Let η be a network, N a network configuration, $\mathfrak{c} \in \mathrm{dom}(\eta)$ a coordinator identifier, $\mathfrak{q}$ a participant, op an operation, and $\mathfrak{v}_1, \ldots, \mathfrak{v}_n$ a list of parameters. If $\eta(\mathfrak{c})$ is well-formed and $N \xrightarrow{\mathfrak{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N_1$ and $N \xrightarrow{\mathfrak{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N_2$, then $N_1(\mathfrak{c}) = N_2(\mathfrak{c})$.*

Proof. Assume $N \xrightarrow{\mathfrak{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N_1$ and $N \xrightarrow{\mathfrak{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N_2$.

Let $\langle \mathbb{Q}, q_0, \longrightarrow\rangle$ be the coordinator $\eta(\mathfrak{c})$, and let $\langle q, \pi, \sigma, \omega\rangle$ be the configuration of $\mathfrak{c}$ in N.

Let $N_1(\mathfrak{c}) = \langle q_1, \pi_1, \sigma_1, \omega_1\rangle$ and $N_2(\mathfrak{c}) = \langle q_2, \pi_2, \sigma_2, \omega_2\rangle$.

We must show that $q_1 = q_2$, $\pi_1 = \pi_2$, $\sigma_1 = \sigma_2$, and $\omega_1 = \omega_2$.

Since both transitions originate from the same network configuration N with the same invocation $\mathfrak{q} : \mathfrak{c}.\mathsf{op}(\mathfrak{v}_1, \ldots, \mathfrak{v}_n)$, they must both satisfy the premises of the [TR] rule.

Let the transition from N to N_1 select $q_1 \xrightarrow{\langle \rho_1, g_1, [L_1] \triangleright \mathsf{p}\colon \mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n) \vdash \{A_1\}, \rho_1' \rangle} q_1'$ and the transition from N to N_2 select $q_2 \xrightarrow{\langle \rho_2, g_2, [L_2] \triangleright \mathsf{p}\colon \mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n) \vdash \{A_2\}, \rho_2' \rangle} q_2'$, where both contain operation op.

Since both transitions are executed from the same configuration N with the same parameters $\mathfrak{v}_1, \ldots, \mathfrak{v}_n$, the same environment update σ' is constructed. Therefore, all premises of the [TR] rule are satisfied in the same environment and role mapping for both transitions:

(i) Both guards g_1 and g_2 evaluate to true;
(ii) Both role constraints ρ_1 and ρ_2 are satisfied;
(iii) The list of call tries L_1 and L_2 to be executed is empty (since this lemma considers transitions without call tries).

Since $\eta(\mathfrak{c})$ is well-formed by hypothesis and the list of call tries is empty, Definition III.16 guarantees that if multiple transitions exist from a state q with operation op, then either their guards are inconsistent or their role constraints are inconsistent. However, we have shown that both guards evaluate to true and both role constraints are satisfied under the same environment and role mapping. Therefore, by well-formedness, there can only be at most one transition from a state q with operation op that can be taken. Therefore, the selected transitions must be the same: $q_1 = q_2$.

Since both executions select the same transition $\mathfrak{t} = q \xrightarrow{\langle \rho, g, \mathsf{L} \triangleright \mathsf{p}\colon \mathsf{op}(x_1,\ldots,x_n) \vdash A, \rho' \rangle} q'$ (where $q = q_1 = q_2$) and execute from the same initial configuration with the same environment σ', role mapping π, and empty call tries, the [TR] rule produces:

- The same successor state q', so $q_1' = q_2' = q'$;
- The same updated role mapping π' (determined by the role constraints in $\mathfrak{t}$), so $\pi_1 = \pi_2 = \pi'$;
- The same updated environment σ'' (from executing the assignments in $\mathfrak{t}$), so $\sigma_1 = \sigma_2 = \sigma''$;
- The same read-only flag value, so $\omega_1 = \omega_2$ as there is no call tries.

Therefore, $N_1(\mathfrak{c}) = N_2(\mathfrak{c})$. ◀

▶ **Lemma III.18.** *Let η be a network, N a network configuration, $\mathfrak{c} \in \mathrm{dom}(\eta)$ a coordinator identifier, $\mathfrak{q}$ a participant, op an operation, and $\mathfrak{v}_1, \ldots, \mathfrak{v}_n$ a list of parameters. Let $N(\mathfrak{c}) = \langle q, \pi, \sigma, \omega \rangle$ and $\eta(\mathfrak{c}) = \langle \mathbb{Q}, q_0, \longrightarrow \rangle$. Suppose $\eta(\mathfrak{c})$ has a transition $q \xrightarrow{\langle \rho, g, \mathsf{L} \triangleright \mathsf{p}\colon \mathsf{op}(x_1,\ldots,x_n) \vdash A, \rho' \rangle} q'$ whose list of call tries L is non-empty, and that the invocation $\mathfrak{q} : \mathfrak{c}.\mathsf{op}(\mathfrak{v}_1, \ldots, \mathfrak{v}_n)$ in N is processed by executing L. If $N \xhookrightarrow[\pi,\sigma]{\mathfrak{q}:[\![\mathsf{L}]\!]_{N,\pi,\sigma}} N_1$ and $N \xhookrightarrow[\pi,\sigma]{\mathfrak{q}:[\![\mathsf{L}]\!]_{N,\pi,\sigma}} N_2$, then $N_1 = N_2$.*

Proof. We must show that for all coordinators $\mathfrak{c} \in \mathrm{dom}(\eta)$, we have $N_1(\mathfrak{c}) = N_2(\mathfrak{c})$.

We prove this by induction on the structure of call tries.

Base case:

If the call try sequence is empty (ϵ), then by the [EMPTY] rule, $N \xhookrightarrow[\pi,\sigma]{\mathfrak{q}:\epsilon} N$, so $N_1 = N_2 = N$.

Therefore, for all $\mathfrak{c} \in \mathrm{dom}(\eta)$, we have $N_1(\mathfrak{c}) = N_2(\mathfrak{c}) = N(\mathfrak{c})$.

Inductive case:

Consider a call try sequence $\mathfrak{c}_\mathrm{p}.\mathsf{op}(\mathfrak{v}_1, \ldots, \mathfrak{v}_n); \mathsf{L}$.

By the [SEQ] rule, if $N \xhookrightarrow[\pi,\sigma]{\mathfrak{q}:\mathfrak{c}_\mathrm{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n);\mathsf{L}} N_1$, then there exists N'' such that:

$N \xrightarrow{\mathfrak{q}:\mathfrak{c}_\mathrm{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N''$, and $N'' \xhookrightarrow[\pi,\sigma]{\mathfrak{q}:[\![\mathsf{L}]\!]_{N'',\pi,\sigma}} N_1$.

Similarly, if $N \xhookrightarrow[\pi,\sigma]{\mathfrak{q}:\mathfrak{c}_\mathrm{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n);\mathsf{L}} N_2$, then there exists N''' such that:

$N \xrightarrow{\mathfrak{q}:\mathfrak{c}_\mathrm{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N'''$, and $N''' \xhookrightarrow[\pi,\sigma]{\mathfrak{q}:[\![\mathsf{L}]\!]_{N''',\pi,\sigma}} N_2$.

By Lemma III.17, since both transitions $N \xrightarrow{\mathfrak{q}:\mathfrak{c}_\mathrm{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N''$ and $N \xrightarrow{\mathfrak{q}:\mathfrak{c}_\mathrm{p}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N'''$ originate from the same configuration N with the same invocation, we have $N''(\mathfrak{c}_\mathrm{p}) = N'''(\mathfrak{c}_\mathrm{p})$ for the coordinator $\mathfrak{c}_\mathrm{p}$ being called.

Since a single call try affects only one coordinator(of course without recursion), for all other coordinators $\mathfrak{c} \neq \mathfrak{c}_\mathrm{p}$, we have $N''(\mathfrak{c}) = N(\mathfrak{c}) = N'''(\mathfrak{c})$ as they are not affected by the call try. Therefore, $N'' = N'''$.

Both N_1 and N_2 are obtained by processing the same remaining call tries L from this common configuration: $N'' \xrightarrow[\pi,\sigma]{\mathsf{q}:[\![\mathsf{L}]\!]_{N'',\pi,\sigma}} N_1$ and $N''' \xrightarrow[\pi,\sigma]{\mathsf{q}:[\![\mathsf{L}]\!]_{N''',\pi,\sigma}} N_2$. Since $N'' = N'''$, the induction hypothesis applied to L yields $N_1 = N_2$.

Failed call try ([CALL]):

For a sequence $\mathfrak{c}_\mathsf{p}.\overline{\mathsf{op}}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n);\mathsf{L}$, the [CALL] rule gives $N \xrightarrow[\pi,\sigma]{\mathsf{q}:\mathfrak{c}_\mathsf{p}.\overline{\mathsf{op}}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n);\mathsf{L}} N'$ if and only if $N \xrightarrow[\pi,\sigma]{\mathsf{q}:[\![\mathsf{L}]\!]_{N,\pi,\sigma}} N'$, with no change to the network on the failed call. Both derivations to N_1 and N_2 therefore process L from the same N; by the induction hypothesis, $N_1 = N_2$. ◀

▶ **Theorem III.19.** *Let η be a network. For all $\mathfrak{c}$ such that $\eta(\mathfrak{c}) = \Sigma$ and Σ is well-formed, we have that η is deterministic.*

Proof. Assume that every coordinator $\eta(\mathfrak{c})$ is well-formed. We must show that η satisfies Definition III.15. Consider a reachable configuration N and suppose $N \xrightarrow{\mathsf{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N_1$ and $N \xrightarrow{\mathsf{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N_2$.

By Lemma III.17, well-formedness of $\eta(\mathfrak{c})$ forces both derivations to use the same coordinator transition $q \xrightarrow{\langle \rho,\mathsf{g},\mathsf{L} \,\triangleright\, \mathsf{p}:\, \mathsf{op}(\mathsf{x}_1,\ldots,\mathsf{x}_n) \,\vdash\, \mathsf{A},\rho'\rangle} q'$ and hence the same updated environment σ', role mapping π', and list of call tries L arising from that transition.

If $\mathsf{L} = \epsilon$, the [TR] rule applies the assignments and state update of $\mathsf{g}, \mathsf{L} \triangleright \mathsf{p}\colon \mathsf{op}\,(\mathsf{x}_1,\ldots,\mathsf{x}_n) \vdash \mathsf{A}$ without a non-trivial call-tries phase: the [EMPTY] rule applies, and the successor configuration is determined uniquely from N, σ', and π'. The two derivations therefore coincide, so $N_1 = N_2$.

If $\mathsf{L} \neq \epsilon$, both derivations first build the same intermediate configuration N'' ($N \xrightarrow{\mathsf{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N''$ and $N \xrightarrow{\mathsf{q}:\mathfrak{c}.\mathsf{op}(\mathfrak{v}_1,\ldots,\mathfrak{v}_n)} N''$) in which $\mathfrak{c}$ is present in read-only mode with environment σ' and state q, while every other coordinator $\mathfrak{c}' \neq \mathfrak{c}$ is unchanged relative to N, i.e., $N''(\mathfrak{c}') = N(\mathfrak{c}')$. The remaining step in each derivation is $N'' \xrightarrow[\pi,\sigma']{\mathsf{q}:[\![\mathsf{L}]\!]_{N'',\pi,\sigma'}} N_i$ ([CALL] rule) for $i \in \{1,2\}$. Lemma III.18 applies to this phase and yields $N_1 = N_2$. ◀

The converse of Theorem III.19 does not hold: a deterministic network can be constructed from coordinators that are not well-formed. The following example demonstrates this.

▶ **Example III.20** (Deterministic network from non-well-formed coordinator)**.** Consider a coordinator $\mathfrak{c}_{\mathrm{nwf}}$ with two transitions from state q_1 both labelled with operation $\mathsf{op}(\mathsf{x})$:

$$\mathsf{t}_1 = \langle q_1, \rho, \mathsf{x} \geq 0 \,\triangleright \mathsf{p}\colon\, \mathsf{op}(\mathsf{x}), \rho', q_2\rangle$$
$$\mathsf{t}_2 = \langle q_1, \rho, \mathsf{x} \leq 0 \,\triangleright \mathsf{p}\colon\, \mathsf{op}(\mathsf{x}), \rho', q_2\rangle$$

where ρ and ρ' are the same for both transitions, and both transitions have empty call try sequences. Both transitions target the same state q_2 and perform the same assignments.

The conjunction of the guards is $\mathsf{x} \geq 0 \wedge \mathsf{x} \leq 0$, which is equivalent to $\mathsf{x} = 0$ and is not inconsistent (it is satisfiable when $\mathsf{x} = 0$). According to Definition III.16, since the guard conjunction is not inconsistent, the transitions must be distinguished by either: (i) inconsistent role constraints, or (ii) call tries where one expects success and the other expects failure. However, $\rho_1 = \rho_2 = \rho$ (same role constraints), and both transitions have empty call try sequences. Therefore, $\mathfrak{c}_{\mathrm{nwf}}$ is not well-formed.

Nevertheless, consider a network $\eta = \{\mathfrak{c}_{\mathrm{nwf}} \mapsto \langle\{q_0, q_1, q_2\}, q_0, \{\mathsf{t}_0, \mathsf{t}_1, \mathsf{t}_2\}\}\rangle$ where t_0 is a transition from q_0 to q_1 with operation `start`. For any reachable configuration N where $\mathfrak{c}_{\mathrm{nwf}}$ is in state q_1, and for any invocation $\mathfrak{q} : \mathfrak{c}_{\mathrm{nwf}}.\mathsf{op}(\mathfrak{v})$:

- If $\mathfrak{v} > 0$, only t_1 is enabled (guard $\mathsf{x} \geq 0$ is satisfied, $\mathsf{x} \leq 0$ is not).
- If $\mathfrak{v} < 0$, only t_2 is enabled (guard $\mathsf{x} \leq 0$ is satisfied, $\mathsf{x} \geq 0$ is not).
- If $\mathfrak{v} = 0$, both transitions are enabled, but since they have the same source state, role constraints, target state, and assignments, they produce the same successor configuration.

In all cases, the network transition produces a unique successor configuration N'. Therefore, the network is deterministic according to Definition III.15, even though the coordinator $\mathfrak{c}_{\text{nwf}}$ is not well-formed. ◇

Chapter IV: Code and tests generation from EDAM

This chapter describes how networks of EDAMs are translated into executable Solidity smart contracts and their corresponding test suites. The translation bridges the gap between formal specifications and deployable code, ensuring that the generated contracts faithfully implement the modelled behaviour.

We present two complementary processes: (i) code generation, which systematically converts formal model elements – states, transitions, roles, fields, guards, and call tries – into structured Solidity code; and (ii) test generation, which produces comprehensive test suites by deriving concrete test cases from symbolic traces generated through the formal semantics. The code generation process handles features including role-based access control, state-dependent transitions, guard conditions, and inter-contract interactions, producing maintainable and correct implementations. The test generation methodology combines symbolic exploration with constraint solving to systematically cover the behavioural space of the specified contracts, producing comprehensive test suites that validate the generated contracts against the formal specification.

The translation preserves the semantics of the formal model [1], ensuring that the generated contracts behave as specified by the EDAM network.

IV.1 Data Representation

This section presents the formal grammar for expressions and role assignments, along with the JavaScript Object Notation (JSON) data representation used as an intermediate format in our code and test generation framework. The JSON representation serves as a language-agnostic serialisation of EDAM specifications, decoupling the formal model from target implementation languages and enabling extensible code generation for multiple blockchain platforms whilst preserving the complete semantic structure of the specification.

We define three interconnected grammars: (i) the expression grammar $\mathcal{G}_{\text{exp}}$, which captures the syntax and structure of expressions used in guards, assignments, and function calls; (ii) the role assignment grammar $\mathcal{G}_{\rho}$, which encodes constraints and updates on participant roles; and (iii) the JSON grammar $\mathcal{G}_{\text{JSON}}$, which specifies the serialisation format for complete EDAM networks. All grammars are defined using Backus–Naur Form (BNF) notation [21], with OCaml algebraic data types corresponding directly to grammar productions, ensuring a direct mapping between the formal model and its serialised representation.

IV.1.1 Expression Grammar

The expression language is defined by the OCaml algebraic data types presented in Section IV.3.2 of Section IV.3.2.

▶ **Definition IV.1** (Expression Grammar). *The expression grammar $\mathcal{G}_{exp}$ is defined as follows:*

[1] Semantics preservation is not formally proven, but empirically established by simulating the formal execution via our Ocaml interpreter which implements the semantics of EDAM and by testing.

$$
\begin{aligned}
\texttt{Value} &::= BoolVal(bool) \mid IntVal(int) \mid StrVal(string) \\
&\mid PtpID(PID) \mid ListVal(\texttt{Value}^*) \mid MapVal((\texttt{Value}, \texttt{Value})^*) \\
\texttt{Exp} &::= Pvar\_a(PtpVar) \mid Dvar(Dvar) \mid Val(Value) \\
&\mid Plus(\texttt{Exp}, \texttt{Exp}) \mid Minus(\texttt{Exp}, \texttt{Exp}) \\
&\mid Times(\texttt{Exp}, \texttt{Exp}) \mid Divide(\texttt{Exp}, \texttt{Exp}) \\
&\mid ListIndex(\texttt{Exp}, \texttt{Exp}, \texttt{Exp}) \mid MapIndex(\texttt{Exp}, \texttt{Exp}, \texttt{Exp}) \\
&\mid And(\texttt{Exp}, \texttt{Exp}) \mid Or(\texttt{Exp}, \texttt{Exp}) \mid Not(\texttt{Exp}) \\
&\mid GreaterThan(\texttt{Exp}, \texttt{Exp}) \mid Equal(\texttt{Exp}, \texttt{Exp}) \\
&\mid Self(string) \\
&\mid FuncCall(string, \texttt{Exp}^*) \\
&\mid FuncCallEdamRead(string, \texttt{Exp}) \\
\texttt{ExtCall} &::= FuncCallEdamWrite(string, Operation, \texttt{Exp}^*, \texttt{Exp}^*) \\
\texttt{Guard} &::= (\texttt{Exp}, (\texttt{ExtCall} \times bool)^*)
\end{aligned}
$$

where $\texttt{PtpVar} = \texttt{Ptp}(\texttt{string})$, $\texttt{Dvar} = \texttt{Var}(\texttt{string})$, $\texttt{PID} = \texttt{PID}(\texttt{string})$, *and* * *denotes zero or more repetitions (Kleene star).*

The grammar captures the structure of expressions as they appear in the OCaml type system. Terminal symbols `string`, `int`, and `bool` represent primitive types, whilst non-terminals `Exp`, `Value`, and `ExtCall` correspond to the OCaml algebraic data types. The expression constructors serve specific purposes: arithmetic operations (`Plus`, `Minus`, `Times`, `Divide`) and boolean logic (`And`, `Or`, `Not`) enable computation; comparisons (`GreaterThan`, `Equal`) support guard conditions; collection access (`ListIndex`, `MapIndex`) enables indexed retrieval from lists and maps; `Self` provides contract self-reference for coordinator interactions; `FuncCall` invokes global functions (e.g., `sum`, `min`); and `FuncCallEdamRead` enables reading data from external coordinators. `FuncCallEdamWrite` enables writing data to external coordinators. The `Guard` non-terminal represents a guard condition, which is a boolean expression combined with a list of external call specifications. The `ExtCall` non-terminal represents an external call specification, which is a string representing the external call and a list of arguments. The `Value` non-terminal represents a value, which is a primitive type or a complex type. The `Exp` non-terminal represents an expression, which is a value or a complex expression. The `ExtCall` non-terminal represents an external call specification, which is a string representing the external call and a list of arguments.

IV.1.2 Role Assignment Grammar

Role assignments encode constraints and updates on participant roles. We define a grammar for role assignments that will be used in the JSON representation in Definition IV.3.

▶ **Definition IV.2** (Role Assignment Grammar). *The role assignment grammar* $\mathcal{G}_\rho$ *is defined as follows:*

$$
\begin{aligned}
\texttt{RoleMode} &::= Top \mid Bottom \mid Unknown \\
\texttt{RoleAssign} &::= \{user : string, role : string, mode : \texttt{RoleMode}\} \\
\texttt{Rho} &::= \texttt{RoleAssign}^* \\
\texttt{RhoPrime} &::= \texttt{RoleAssign}^*
\end{aligned}
$$

The `RoleMode` non-terminal encodes role modalities: `Top` corresponds to ■ (participant must have the role), `Bottom` corresponds to □ (participant must not have the role), and `Unknown` corresponds to ▣ (role constraint is immaterial). The `Rho` and `RhoPrime` non-terminals represent role constraint and role update sets, respectively, as arrays of role assignment objects.

IV.1.3 JSON Grammar

The JSON representation provides a structured, language-agnostic format for serialising EDAM specifications. We define the grammar using BNF notation with Kleene star (*) for zero or more repetitions and Kleene plus ($^+$) for one or more repetitions. The grammar references the expression grammar $\mathcal{G}_{\text{exp}}$ defined in Definition IV.1 and the role assignment grammar $\mathcal{G}_\rho$ defined in Definition IV.2.

▶ **Definition IV.3** (JSON Grammar). *The JSON grammar $\mathcal{G}_{JSON}$ for EDAM serialisation is defined as follows:*

$$
\begin{aligned}
\texttt{EDAM} &::= \{name : string, states : string^*, initialState : string,\\
&\quad finalStates : string^*, roles : string^*,\\
&\quad variables : \texttt{VarMap}, transitions : \texttt{Transition}^+ \}\\
\texttt{VarMap} &::= \{string : \texttt{Type}\}^*\\
\texttt{Type} &::= int \mid string \mid bool \mid user \mid string\\
\texttt{Transition} &::= \{from : string, to : string, operation : string,\\
&\quad ptpVar : string, ptpVarList : string^*,\\
&\quad paramVar : \texttt{VarMap}, guard : \texttt{GuardJSON},\\
&\quad rho : \texttt{Rho}, rhoPrime : \texttt{RhoPrime},\\
&\quad assignments : \texttt{AssignMap} \}\\
\texttt{GuardJSON} &::= [\texttt{ExprString}, \texttt{ExtCallJSON}^*]\\
\texttt{ExtCallJSON} &::= \{type : externalCall, modelName : string,\\
&\quad operation : string, args : [\texttt{ExprString}^*, \texttt{ExprString}^*],\\
&\quad enabled : bool \}\\
\texttt{AssignMap} &::= \{string : \texttt{ExprString}\}^*\\
\texttt{ExprString} &::= string \quad (\mathcal{G}_{exp})
\end{aligned}
$$

The grammar uses standard JSON notation where { ... } denotes objects, [...] denotes arrays, * indicates zero or more repetitions, and $^+$ indicates one or more repetitions. The `ExprString` non-terminal represents strings that encode OCaml expressions according to $\mathcal{G}_{\text{exp}}$. The `Transition` $^+$ production ensures that each EDAM contains at least one transition. The `Rho` and `RhoPrime` non-terminals follow $\mathcal{G}_\rho$.

IV.1.4 Transformation Pipeline

The code generation framework employs a multi-stage transformation pipeline that converts EDAM specifications into executable code and test suites. Figure IV.1 illustrates the complete transformation architecture.

The transformation pipeline operates in two parallel branches:

EDAM → JSON → Target Languages The `EDAM2JSON` transformation serialises the formal EDAM specification into JSON format according to $\mathcal{G}_{\text{JSON}}$. This JSON representation serves as a language-agnostic intermediate format that preserves the complete structure of the EDAM. From JSON, language-specific generators (`JSON2Solidity`, `JSON2Move`, `JSON2*`) produce executable code for various blockchain platforms. The JSON format enables extensibility: new target languages can be supported by implementing additional JSON-to-language transformers without modifying the core EDAM serialisation.

EDAM → OCaml → Hardhat Tests The `EDAM2OCaml` transformation converts the EDAM specification into OCaml algebraic data types that implement the formal semantics, as presented in Section IV.3.1. The OCaml representation enables symbolic execution, trace generation, and semantic validation. The `OCaml2Hardhat` transformation generates executable Hardhat test

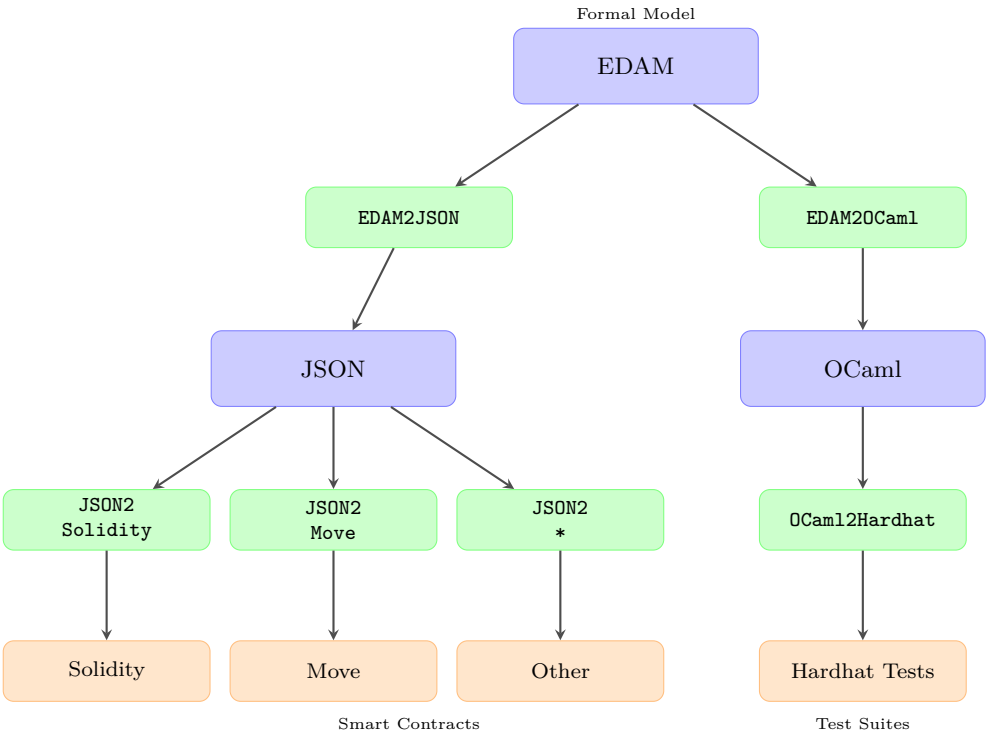


Figure IV.1: Transformation pipeline architecture

suites from the OCaml interpreter's execution traces, producing comprehensive test cases that validate the generated smart contracts against the formal specification.

This dual-path architecture separates concerns: the JSON branch focuses on code generation for multiple target languages, whilst the OCaml branch focuses on semantic validation and test generation. Both paths originate from the same EDAM specification, ensuring consistency between generated code and test suites.

IV.2 Code Generation

The code generation process transforms each EDAM in the network into a Solidity contract. The input is a JSON representation of the EDAM produced by the OCaml type checker and semantic executor (`EDAM2JSON` transformation). The code generator parses this JSON and synthesises Solidity code through systematic element-by-element conversion.

The JSON data representation, including the formal grammar for expressions and the JSON structure, is presented in Definition IV.3. The JSON format serves as a language-agnostic intermediate representation that preserves the complete semantic structure of EDAM specifications, enabling code generation for multiple target languages.

IV.2.1 Element Conversion Process

The conversion process (`JSON2Solidity` transformation) converts each JSON element to Solidity code components. The conversion is done as follows:

- States: We generate $\mathbb{Q}$ as a Solidity `enum State` type, containing every $q \in \mathbb{Q} \setminus \{q_0\}$ (e.g., `enum State { q1, q2 }`). The initial state q_0 stays implicit: the start transition is fire automatically on deployment of the contract, and no other transition may target q_0 (cf. Definition III.6).
- Roles: The set of roles R is generated as a Solidity `enum Roles`, where each role $R \in \mathsf{R}$ becomes an enum value (e.g., `enum Roles { O, B }`).

- Current State: The current state q of the coordinator is tracked by a public state variable `State public _state`, initialised in the constructor by the target state of the `start` transition which will be automatically fired on deployment of the contract.
- Fields: Each field $\mathsf{f} \in \mathsf{F}$ of the EDAM is generated as a contract variable with its corresponding type (e.g., `uint public f`, `address public buyer`).
- Role Mapping (π): The role mapping $\pi : \mathcal{P} \to 2^{\mathsf{R}}$ is implemented as a nested mapping `mapping(address => mapping(Roles => bool))public _permissions`, where `_permissions[p][R]` represents whether participant $\mathfrak{p}$ has role R or not.
- Environment (σ): The environment σ that maps variables to values is implicitly maintained through contract state variables. In blockchain, the environment is the blockchain state.
- Expressions: guards, assignments, and call-try parameters expressions are translated into Solidity expressions (e.g., $\mathfrak{sf}$ will be translated into `address(this)`, the caller address is represented by `msg.sender`, expression `Plus(Dvar (Var "a"), Dvar (Var "b"))` is translated into `a + b`).
- Transitions: Transitions are grouped by operation, with each group translated into a single Solidity function (more on this in Section IV.2.1)
- Constructor: The constructor is generated from all `start` transitions of the EDAM. More on this in Section IV.2.1.

Satisfaction Relation Implementation

The satisfaction relation is a fundamental concept in the EDAM model. It is used to check if a role assignment is satisfied by a given environment and role mapping. We implement the satisfaction relation as a helper function in the Solidity code and use it as a constraint together with the guard conditions. We also use it in the body of the functions to check if the role assignment is satisfied after the update of the role mapping with respect to the role assignment ρ'. The following Solidity code implements the satisfaction relation $\langle \sigma, \pi \rangle \models \rho$ from Definition III.7:

```
function roleSatisf(address participant, Roles[] memory hasrole_roles,
  Roles[] memory notrole_roles ) internal view returns (bool) {
  for (uint i = 0; i < notrole_roles.length; i++) {
    if (_permissions[participant][notrole_roles[i]]) {
      return false; // intersection not empty
    }
  }
  for (uint i = 0; i < hasrole_roles.length; i++) {
    if (!_permissions[participant][hasrole_roles[i]]) {
      return false; // hasrole_roles not subset of participant_roles
    }
  }
  return true;
}
```

Listing IV.1: Solidity implementation of the satisfaction relation

- **Intersection condition**: $\pi(\sigma(\mathsf{p})) \cap \rho_{\square}(\mathsf{p}) = \emptyset$ - ensures that the participant has none of the forbidden roles (Bottom roles)
- **Subset condition**: $\rho_{\blacksquare}(\mathsf{p}) \subseteq \pi(\sigma(\mathsf{p}))$ - ensures that the participant has all required roles (Top)

The function returns `true` if and only if both conditions hold for the participant variable `p` and we do the conjunction for all participant variables in the domain of the role assignment, corresponding exactly to the formal definition of satisfaction. The `roleSatisf` function is used to generate the role assertion in the constructor and the function bodies and thus is a helper function for the code generation process.

Constructor Generation

The constructor is generated from all `start` transitions of the EDAM. Unlike networks of EDAMs, where coordinators' identifiers are set by the user at network creation time, smart contracts require explicit contract addresses for inter-contract calls. When a EDAM contains call tries, the constructor receives additional parameters for callee contract addresses, which are stored as contract state

variables to avoid passing them on every call. Circular dependencies between contracts are forbidden, as they would make deployment impossible in practice.

The constructor synthesizes code from all transitions where `operation` equals `"start"`. Multiple start transitions with distinct guards generate an if-else chain.

For each transition in the list of `start` transitions, the synthesizer generates the constructor code as follows:

1. Extracts constructor parameters from `paramVar`, `ptpVarList`, and external contract types in `variables`. Participant parameters from `ptpVarList` are mapped to `address` types. Data parameters from `paramVar` are mapped via type mapping: `"int"` ↦ `uint`, `"user"` ↦ `address`, `"string"` ↦ `string` `"bool"` ↦ `bool`, ... External contract types (values in `variables` that are contract names) generate parameters `ContractType __contractVar` and import statements `import "./ContractType.sol";` for each external contract type. These parameters are used to initialise the contract variables in the constructor body.
2. Generates contract variable declarations for each external contract: `ContractType public _contractVar;`. This is used to store the address of the external contract in the constructor body.
3. Parses the guard expression string `ExprString` from the guard expression `g` into a Solidity boolean expression (e.g.,. `GreaterThan(Dvar (Var "a"), Dvar (Var "b"))` is translated into `a > b`).
4. Processes assignments: for each key-value pair in the assignment map A (`AssignMap` in the JSON representation), parses the value expression string and generates `<field> = <parsed_expr>;`. For assignments referencing other fields modified in the same transition, introduces temporary variables to preserve simultaneous assignment semantics.
5. Generates role updates: for each object in `rhoPrime` with `mode "Top"`, generates `_permissions[<user>][Roles.<role>] = true`. For `"Bottom"`, generates `_permissions[<user>][Roles.<role>] = false`.
6. Generates state update: `_state = State.<to>`.
7. Generates role assertion: if `rhoPrime` is non-empty, we generate the role assertion using the `roleSatisf` helper function. `assert(roleSatisf(<ptpVar>, _roles(...), new Roles[](0)));`
8. Wraps external calls: if `ExtCallJSON` (the external calls array) is non-empty, wraps the transition body in nested `try/catch` blocks (see Section IV.2.1).

The constructor body structure follows this template:

```
constructor(<params>) <nonReentrant> {
    // Contract variable assignments (from variables object)
    _contractVar = __contractVar;

    // Guard-based if-else chain for multiple start transitions
    if (_state == State.<from> && <guard1> && <role_checks1>) {
    // External calls wrapped in try/catch if present
        // Assignments, role updates, state update, role assertion
    } else if (_state == State.<from> && <guard2> && <role_checks2>) {
        // ...
    } else {
        revert("Condition not met");
    }
}
```

If any transition contains calls tries, the constructor includes the `nonReentrant` modifier and declares `bool private _entered;` with the modifier implementation. A Solidity modifier is a special function that modifies the behaviour of other functions. When a function is decorated with a modifier, the modifier's code executes before and/or after the function body. The underscore (`_`) in the modifier marks where the original function body is inserted. Modifiers enable code reuse and provide a mechanism to enforce preconditions and postconditions across multiple functions.

The `nonReentrant` modifier prevents reentrancy attacks, where an external contract calls back into the current contract before the original call completes. This protection is essential when making

external calls, as malicious contracts could exploit the intermediate state to execute unauthorized transitions or drain funds.

The following is the modifier implementation:

```
1 bool private _entered;
2
3 modifier nonReentrant() {
4     require(!_entered, "reentrant call");
5     _entered = true;
6     _; // the function body is executed here
7     _entered = false;
8 }
```

The `_entered` private variable (line 1) tracks whether the contract is currently executing a function protected by this modifier. Before execution, the modifier checks that `_entered` is `false` (line 4), then sets it to `true`. After the function body executes (at the '_' position, line 6), the flag is reset to `false`. If a reentrant call attempts to enter a protected function while `_entered` is `true` (line 4), the `require` statement reverts the transaction, preventing the reentrant execution.

Function Generation

Transitions are grouped by operation, with each group translated into a single Solidity function (one function per operation, as each operation has a unique function type in T'). Within each function body, transitions are further grouped by source state, operation, and guard (cf. Definition III.16). Each group is represented as an `if` statement that checks: (i) the current state (`_state == State.q1`), (ii) role satisfaction for all participants in ρ (implemented as a conjunction of role checks), and (iii) the guard conditions `g`. The `if` body executes: call-try sequences, assignments A, role updates `rupd` (if ρ' is not the constant function mapping all participants and roles to $\blacksquare$), state updates to q' and assertions to check role satisfaction (implemented as assert statements).

The grouping algorithm partitions transitions by `operation`, then within each operation group by `(from, guard, role assignments)` triple, where guards and role assignments are serialized to strings for comparison. For each group, the synthesizer produces an `if` statement with condition: `_state == State.<from> && <guard> && <role_checks>`.

Role assignments are grouped by structure rather than by participant names. The synthesizer normalizes role assignments by extracting the role structure: caller roles (mapping role names to modes: `"Top"`, `"Bottom"`, or `"Unknown"`) and participant roles (ordered list of role mappings, one per participant position). This normalization allows transitions with different participant variable names but identical role structures to be grouped together. For example, transitions requiring role O with mode `"Top"` for the caller are grouped together regardless of whether the caller is named `p`, `p1`, or `caller`. The serialization format separates caller roles from participant roles: `caller:{role1:mode1,role2:mode2};participants:{role1:mode1,role2:mode2,role3:mode3}`, where roles within each section are sorted for consistent comparison. Function parameters derive from the first transition in each operation group: `paramVar` keys map to typed parameters, `ptpVarList` entries map to `address` parameters. This is because the typing environment allows only one type for an operation.

For each transition group, the synthesizer generates the function body as follows:

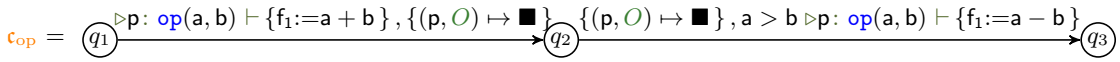
1. Builds the condition: state check `_state == State.<from>`, parsed guard expression (from `guard[0]`), and role satisfaction checks via `_permissions[msg.sender][Roles.<role>]` for each role requirement in ρ.
2. If the group contains transitions with external calls (non-empty `guard[1]`), builds nested `try/catch` blocks (see Section IV.2.1). Otherwise, generates assignment statements, role updates, state update, and role assertion directly.

The function structure follows this template:

```
1 function <operation>(<params>) public [nonReentrant] {
2     if (_state == State.<q1> && <guard1> && <role_checks1>) {
3         // External calls wrapped in try/catch if present
```

```
4         // Transition body: assignments, role updates, state update and
              role satisfaction assertion
5     } else if (_state == State.<q2> && <guard2> && <role_checks2>) {
6         // ...
7     } else {
8         revert("Condition not met");
9     }
10 }
```

To illustrate the function generation process, we consider an example of system where the coordinator has three states q_1, q_2 and q_3 and two transitions $q_1 \to q_2$ and $q_2 \to q_3$ with the same operation op. The transition from q_1 to q_2 give the caller role O with mode `"Top"` after invoking the operation op with parameters a and b and updates the field f_1 to $a + b$. The transition from q_2 to q_3 requires the caller to have role O with mode `"Bottom"` after invoking the operation op with parameters a and b and updates the field f_1 to $a - b$. The figure IV.2 illustrates the function generation process.




```
1 contract Cop {
2   enum State { q1, q2, q3 }
3   enum Roles { O }
4   State public _state;
5   uint  public f1;
6   mapping(address => mapping(Roles => bool)) public _permissions;
7   ...
8   function op(uint a, uint b) public  {
9     if ((_state == State.q1 && true)) {
10      f1 =  (a + b);
11      _permissions[msg.sender][Roles.O] = true;
12      _state = State.q2;
13      assert(
14        roleSatisf(msg.sender, _roles(Roles.O),
15        new Roles [] (0)));
16    } else if ( roleSatisf(msg.sender, _roles(Roles.O), new Roles [] (0))
17      && (_state == State.q2 && a > b)) {
18      f1 =  (a - b);
19      _state = State.q3;
20    } else { revert("Condition not met"); } }
21  ...
22 }
```

Figure IV.2: Example code generation for transition sharing same op.

Figure IV.2 illustrates how transitions sharing the same operation op are grouped into a single `function op`. The contract declarations (lines 2–6) define: line 2 the state enum, line 3 the roles enum, line 4 the current state variable to track q of the coordinator, line 5 the field f1, and line 6 the role mapping `_permissions` to keep track of π for each participant. The first branch (lines 9–15) implements $q_1 \to q_2$: line 9 checks state and guard; line 10 updates f1; the following line assigns role O via `_permissions`; line 12 updates the state; lines 13–15 verify role satisfaction using roleSatisf (from Definition III.7). The second branch (lines 16–19) implements $q_2 \to q_3$: line 16 checks role satisfaction and line 17 verifies state and guard $a > b$; line 18 updates f1; line 19 updates the state. Line 20 handles unmatched conditions by reverting. The ellipses at line 7 and line 21 represent omitted the constructor and the helpers (functions `roleSatisf` and `_roles`), respectively.

Call Tries Generation

External calls in `ExtCallJSON` are mapped to `try/catch` blocks. Each call object with `type: "externalCall"` contains `modelName` (contract identifier), `operation` (method name), `args` (argument expression strings), and `enabled` (boolean indicating expected success). The synthesizer parses each

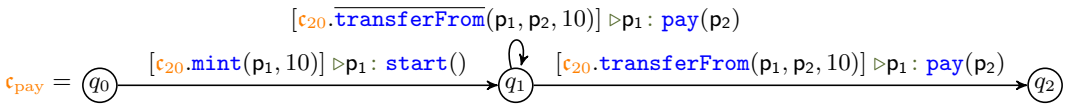
call specification and generates a `try/catch` block where the expected outcome determines control flow. For a call expecting success (`enabled = true`):

```
try _<contract>.<operation>(<params>) {
    /* Transition body: assignments, role updates, state update */
} catch {
  revert("Expected external call to succeed");
}
```

For a call expecting failure (`enabled = false`):

```
try _<contract>.<operation>(<params>) {
    revert("Expected external call to fail");
} catch {
  /* Transition body: assignments, role updates, state update */
}
```

Multiple calls in sequence generate nested blocks. The generator processes calls in reverse order (innermost first) to build from inside out. For transitions sharing a common call sequence prefix, the generator builds a call tree: nodes represent calls, success branches continue with subsequent calls or transition bodies, failure branches handle expected failures or revert. The tree structure eliminates duplicate call code. When transitions from the same state with the same operation and guard differ only in call sequence outcomes, they group into a single `if` branch. The generator builds a tree from their call sequences, then generates nested `try/catch` blocks following the tree structure. At each node, the common call executes in the `try` block; success paths continue to the success subtree, failure paths continue to the failure subtree.




```
1  import "./C20.sol";
2  contract Cpay {
3    ...
4    C20 public _C20;
5    bool private _entered;
6    modifier nonReentrant() {
7      require(!_entered, "reentrant call");
8      _entered = true;
9      _; // the function body is executed here
10     _entered = false
11   }
12   constructor(C20 __C20) {
13     if (true) {
14       _C20 = __C20;
15       try _C20.mint(msg.sender, 10) {
16         _state = State.q1;
17       } catch { revert("Expected external call to succeed"); }
18     } else {
19       revert("Condition not met");
20     }
21   }
22   function pay (address p2) public nonReentrant {
23     if (_state == State.q1 && true) {
24       try _C20.transferFrom(msg.sender, p2, 10) {
25         _state = State.q2;  /* Expected call try to succeed */
26       } catch { _state = State.q1; /* Expected call try to fail */ }
27     } else { revert("Condition not met"); }
28   }
29   ...
30 }
```

Figure IV.3: Example Solidity code for transitions with call tries.

Figure IV.3 shows the Solidity snippet code for the model $\mathfrak{c}_{\text{pay}}$ above the listing. Line 1 imports the ERC20 Token coordinator contract; line 4 declares the contract reference variable while line 5

declares the flag to prevent reentrancy. The `nonReentrant` modifier (lines 6–11) checks the flag (line 7), sets it to `true` before execution (line 8), executes the function body (line 9), and resets it after (line 10). The constructor (lines 12–21) implements the `start` transition: line 14 stores the ERC20 Token contract reference; line 15 executes the call try $\mathfrak{c}_{20}$.`mint`($\mathsf{p}_1, 10$); line 16 sets the initial state to q_1 if the call succeeds and reverts if the call fails (line 17). The `function pay` which is decorated with the `nonReentrant` modifier (line 22) groups both transitions in a `try/catch` block: line 24 executes $\mathfrak{c}_{20}$.`transferFrom`($\mathsf{p}_1, \mathsf{p}_2, 10$); the `try` body updates state to q_2 (line 25); the `catch` block resets state to q_1 (line 26). The ellipses at lines 3 and 29 represent omitted contract variables and contract helper functions, respectively.

Simultaneous Assignment Semantics

In the EDAM model, field assignments within a transition are conceptually executed atomically and simultaneously. However, Solidity executes statements sequentially, which can lead to different results when assignments reference other fields that are being modified in the same transition.

Consider a transition with assignments $\mathsf{x} := \mathsf{y} + 1$ and $\mathsf{y} := \mathsf{x} + 1$, where both fields reference each other. Under simultaneous semantics, if $\mathsf{x} = 5$ and $\mathsf{y} = 3$ initially, both assignments use the original values: x becomes $3 + 1 = 4$ and y becomes $5 + 1 = 6$. However, sequential execution would produce different results: executing $\mathsf{x} = \mathsf{y} + 1$ first updates x to 4, then $\mathsf{y} = \mathsf{x} + 1$ uses the updated value, setting y to 5 instead of 6.

To address this discrepancy, the code generator automatically introduces temporary variables to preserve the intended semantics. The generator analyses assignment dependencies: if an assignment's right-hand side references a field that is also being assigned in the same transition, it stores the original value in a temporary variable before executing any assignments.

```
1 // Transition with assignments: x := y + 1, y := x + 1
2 // Initial state: x = 5, y = 3
3
4 // Generated code preserves simultaneous semantics:
5 uint _temp_x = x;  // Store original x = 5
6 uint _temp_y = y;  // Store original y = 3
7 x = _temp_y + 1;   // x = 3 + 1 = 4 (uses original y)
8 y = _temp_x + 1;   // y = 5 + 1 = 6 (uses original x)
9 // Final state: x = 4, y = 6 (matches simultaneous semantics)
10
11 // Without temporary variables (incorrect):
12 // x = y + 1;      // x = 3 + 1 = 4
13 // y = x + 1;      // y = 4 + 1 = 5 (uses updated x, incorrect!)
14 // Final state: x = 4, y = 5 (does not match simultaneous semantics)
```

The generator detects all fields referenced in assignment right-hand sides and creates temporary variables for fields that are both referenced and assigned. This ensures that all assignments use the pre-transition values, correctly implementing the simultaneous assignment semantics of the EDAM model.

Example of Code Generation

Listing IV.2 and Listing IV.3 illustrate the generated Solidity contracts for $\mathfrak{c}_1$ and $\mathfrak{c}_2$ from Figure III.1. Listing IV.2 shows the minimal single-state contract with role assignment and revocation: the state enumeration (line 2) and role enumeration (line 3) are generated from the coordinator's interface; the state variable (line 4) and permissions mapping (line 5) implement the coordinator's configuration; the constructor (lines 6–10) implements the `start` transition, assigning the initial role (line 8), setting the initial state (line 9), and asserting role satisfaction (line 10); the `op1` function (line 15) implements the role revocation transition, revoking the role (line 17), updating the state (line 18), and asserting the post-transition role constraint (line 19). Listing IV.3 shows the richer contract with both guard-based and role-based access control: the state and role enumerations (line 2), state and field variables (lines 3–4), and permissions mapping (line 5) are generated from the coordinator's interface; the constructor (lines 6–10) implements the `start` transition, assigning

the owner role (line 8), setting the initial state (line 9), and asserting role satisfaction (line 10); the `op2` function (line 15) groups two transitions: the first branch (lines 16–18) implements the transition with guard $\mathsf{a} > \mathsf{b}$, checking the guard (line 16), updating the field (line 17), and updating the state (line 18); the second branch (lines 19–21) implements the transition with guard $\mathsf{a} \leq \mathsf{b}$ and role constraint $\rho(\mathsf{p}, O) = \blacksquare$, checking both the guard and role satisfaction (line 19), updating the field (line 20), and updating the state (line 21). These listings demonstrate how the code generation process instantiates states, roles, constructors, guards, and access checks directly from the EDAM specifications. The ellipses at lines 22 and 24 represent omitted helper functions for role satisfaction.

```
1 contract C1 {
2   enum State { q1 }
3   enum Roles { O }
4   State public _state;
5   mapping(address => mapping(Roles => bool)) public _permissions;
6   constructor(address p1)  {
7     if (true) {
8       _permissions[p1][Roles.O] = true;
9       _state = State.q1;
10      assert(roleSatisf(p1, _roles(Roles.O), new Roles [] (0)));
11    } else {
12      revert("Condition not met");
13    }
14  }
15  function op1 (address p1) public  {
16    if (_state == State.q1 && true) {
17      _permissions[p1][Roles.O] = false;
18    _state = State.q1;
19    assert(roleSatisf(p1, new Roles [] (0), _roles(Roles.O)));
20    } else {revert("Condition not met"); }
21  }
22  ...
23 }
```

Listing IV.2: C1 Contract

```
1 contract C2{
2   enum State { q1 } enum Roles { O }
3   State public _state; uint  public f;
4   uint  public a; uint  public b;
5   mapping(address => mapping(Roles => bool)) public _permissions;
6   constructor()  {
7     if (true) {
8     _permissions[msg.sender][Roles.O] = true;
9     _state = State.q1;
10    assert(roleSatisf(msg.sender, _roles(Roles.O), new Roles [] (0)));
11    } else {
12      revert("Condition not met");
13    }
14  }
15  function op2 (uint a, uint b) public  {
16    if (_state == State.q1 && a > b) {
17      f =  (f + a);
18      _state = State.q1;
19    } else if (_state == State.q1 && a <= b && roleSatisf(msg.sender, _roles(Roles.
        O), new Roles [] (0))) {
20      f =  0;
21      _state = State.q1;
22    } else {revert("Condition not met");}
23  }
24  ...
25 }
```

Listing IV.3: C2 Contract

```
1 import "./C2.sol";
2 contract C3 {
3   enum State { q1, q2 } enum Roles{_______R00_______}
4   State public _state; uint  public f1; uint  public y;
5   C2 public _C2;
6   uint  public a; uint  public b;
7   mapping(address => mapping(Roles => bool)) public _permissions; bool private
      _entered;
```

```
8   modifier nonReentrant() {
9     require(!_entered, "Reentrant call");
10    _entered = true;
11    _;
12    _entered = false;
13  }
14  constructor(uint y, C2 __C2)  {
15    if (true) {
16      _C2 = __C2;
17      f1 =  y;
18      _state = State.q1;
19    } else {
20      revert("Condition not met");
21    }
22  }
23  function op3 (uint a, uint b) public nonReentrant {
24    if (_state == State.q1 && a > b) {
25      try _C2.op2(a, b) {
26        f1 =  b;
27        _state = State.q2;
28      } catch {
29        revert("Expected external call to succeed");
30      }
31    } else {
32      revert("Condition not met");
33      }
34  }
35 }
```

Listing IV.4: C3 Contract (Call to C2 Contract)

Listing IV.4 illustrates the generated contract for c_3, where the generated contract keeps a reference to `C2` and invokes it through call try guarded by `try/catch` (lines 25-28). We add the contract to call in the contract state as a variable `C2 _C2;` (line 5). The callee contract is passed as a parameter to the constructor of the caller contract `constructor(C2 __C2)` (line 14). The callee contract is then used to call its operation op2 (lines 25-28).

IV.3 Test Generation

This section presents the implementation of an interpreter for EDAM specifications that enables simulation and execution of coordinator networks according to the formal semantics defined in Chapter III. The interpreter is implemented in OCaml and provides a runtime environment for executing EDAM specifications, including expression evaluation, transition execution, role management, and network execution for test generation.

IV.3.1 Interpreter Architecture

The interpreter is organised into a modular architecture that separates type definitions, core functionality, and supporting utilities.

The `Types` module defines the foundational type system. The `Helper` module implements expression evaluation through the `eval` function. The `Core_functions` module implements transition execution semantics and network execution, following the rules from Table III.2. The `Z3_module` integrates with Z3 for guard satisfiability checking and symbolic execution for test generation. The `Printer` and `Test_generation` modules provide debugging utilities and test case generation capabilities.

IV.3.2 Type System and Data Structures

The interpreter's type system mirrors the formal model's structure, providing concrete representations for coordinators, configurations, transitions, and expressions.

Core Types

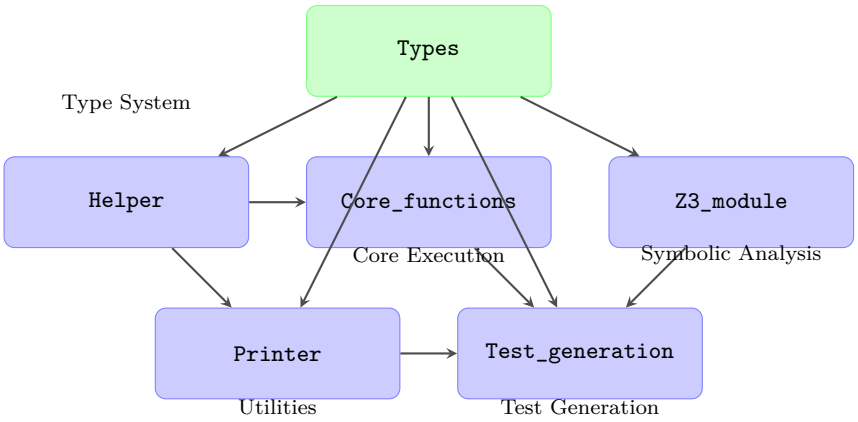


Figure IV.4: Interpreter architecture

```
type ptp_var = Ptp of string
type participant = PID of string
type role_type = Role of string
type role_mode = Top | Bottom | Unknown
type role_mode_type = role_type ->
    role_mode
type role_a_type = ptp_var ->
    role_mode_type
type operation = Operation of string
type dvar_type = VarT of string
type dvar = Var of string
```

The `ptp_var` type represents participant variables. The `participant` type encapsulates concrete participant identifiers. The `role_type` represents role names, while `role_mode` implements role constraint semantics: `Top` corresponds to ■, `Bottom` to ▣, and `Unknown` represents absence of constraint.

Value Types and Expressions

```
type value_type =
  | BoolVal of bool
  | IntVal of int
  | StrVal of string
  | PtpID of participant
  | ListVal of value_type list
  | MapVal of (value_type * value_type)
      list

type exp =
  | Pvar_a of ptp_var
  | Dvar of dvar
  | Plus of exp * exp
  | Minus of exp * exp
  | Times of exp * exp
  | Divide of exp * exp
  | ListIndex of exp * exp * exp
  | MapIndex of exp * exp * exp
  | And of exp * exp
  | Or of exp * exp
  | Not of exp
  | GreaterThan of exp * exp
  | Equal of exp * exp
  | Val of value_type
  | Self of string
  | FuncCall of string * exp list
  | FuncCallEdamRead of string * exp

type funcCallEdamWrite =
  FuncCallEdamWrite of string *
      operation * (exp list) * (exp list
      )

type guard_type = exp * ((
    funcCallEdamWrite * bool) list)
```

The `value_type` represents all possible values. The `exp` type implements the expression language supporting arithmetic, boolean logic, comparisons, and function calls. The `funcCallEdamWrite` type captures external coordinator calls which contain the coordinator name, operation, participants parameters and data variables parameters. The `guard_type` combines a guard expression with a list of external call tries with expected success/failure.

Transitions are represented as triples `(source_state, label, target_state)`, where the label encodes the transition guard, role constraints (pre/post), the caller, the invoked operation, the participant/data parameters, and the assignment list. Concretely, the interpreter uses the following types:

```
type assignment_type = dvar * exp

type label_type =
```

```
  guard_type *                    (* guard + external call tries *)
  role_a_type *                   (* rho: pre role constraints *)
  ptp_var *                       (* caller participant variable *)
  operation *                     (* operation name *)
  (ptp_var list) *                (* caller participant parameters *)
  (ptp_var list) *                (* participant parameters *)
  ((dvar_type * dvar) list) *     (* typed data parameters *)
  (assignment_type list) *        (* assignments *)
  role_a_type *                   (* rho': post role constraints *)
  string                          (* transition name/identifier *)

type transition_type = state_type * label_type * state_type (* source state, label,
    target state *)
```

This representation allows `check_transition` to (i) select candidate transitions by matching the current state and operation, (ii) evaluate the guard and call-tries, (iii) validate and update roles via ρ/ρ', and (iv) apply assignments to compute the next configuration.

Coordinator and Network Structures

```
type configuration = {
  state: state_type;
  lambda: lambda_type;
  sigma: sigma_type;
}

type edam_type = {
  name: string;
  states: state_set;
  transitions: transition_set;
  final_modes: state_set;
  initial_state: state_type;
  roles_list: role_type list;
  ptp_var_list: ptp_var list;
  variables_list: dvar list;
}

type multi_config = {
  edam_map:
    (string, edam_type) Hashtbl.t;
  config_map:
    (string, configuration) Hashtbl.t;
}
```

The `configuration` type implements the coordinator configuration from Definition III.12, where `state` corresponds to q_i, `lambda` implements λ (roles of participants π), and `sigma` implements σ (environment mapping variables to values). The `multi_config` structure maintains the network state, mapping coordinator names to their definitions and current configurations, implementing the network configuration map N from Definition III.12.

IV.3.3 Expression Evaluation

Expression evaluation implements the evaluation function $[\![\_]\!]_{N,\sigma,\pi}$ from Table III.2. The `eval` function processes expressions recursively in the context of an environment σ, a participant mapping ι, and a network configuration, returning both the computed value and an updated network configuration (in case of external calls, the network configuration is updated with the new coordinator configuration). We have that ι is the mapping of participant variables to participant identifiers with is automatically generated by the interpreter from the concrete label.

```
let rec eval (sigma: sigma_type) (iota: iota_type)
             (e: exp) (multi_cfg: multi_config)
             (called_contracts: string list)
             : value_type * multi_config =
  match e with
  | Pvar_a ptp_var -> (PtpID (iota ptp_var), multi_cfg)
  | Dvar dvar -> (sigma dvar, multi_cfg)
  | Plus (e1, e2) ->
      let (v1, multi_cfg1) = eval sigma iota e1 multi_cfg called_contracts in
      let (v2, multi_cfg2) = eval sigma iota e2 multi_cfg1 called_contracts in
      (match (v1, v2) with
      | IntVal i1, IntVal i2 -> (IntVal (i1 + i2), multi_cfg2)
      | _ -> failwith "Type error in Plus")
  | FuncCallEdamRead (edam_name, var_name) ->
      external_edam_read edam_name var_name sigma iota multi_cfg called_contracts
  (* ... other expression cases ... *)
```

The above snippet of code implements the evaluation function $[\![\_]\!]_{N,\sigma,\pi}$ from Table III.2. It handles arithmetic operations, boolean logic, comparisons, collections, and external coordinator interactions. The `called_contracts` parameter tracks coordinator invocations to prevent reentrancy.

IV.3.4 Role Management

Role management implements the role constraint semantics from Definition III.4 and Table III.2. The `role_satisf` function implements the role satisfaction semantics, checking that a role mapping π satisfies the role constraints specified by ρ given the mapping ι of participant variables to participant identifiers:

```
let role_satisf (pi: pi_type) (iota: iota_type)
                (rho: role_a_type) (participant_vars: ptp_var list)
                (roles: role_type list) : bool =
  let unique_participants =
    remove_duplicates (List.map iota participant_vars)
  in
  List.for_all (fun participant ->
    let participant_roles = pi participant  in
    let party_vars_for_participant =
      List.filter (fun partyP -> iota partyP = participant) participant_vars
    in
    let hasrole_roles = ref []  in
    List.iter (fun partyP ->
      List.iter (fun roleR ->
        if rho partyP roleR = Top && not (List.mem roleR !hasrole_roles) then
          hasrole_roles := !hasrole_roles @ [roleR]
      ) roles
    ) party_vars_for_participant;
    let notrole_roles = ref []  in
    List.iter (fun partyP ->
      List.iter (fun roleR ->
        if rho partyP roleR = Bottom && not (List.mem roleR !notrole_roles) then
          notrole_roles := !notrole_roles @ [roleR]
      ) roles
    ) party_vars_for_participant;
    let intersection_empty =
      List.for_all (fun roleR -> not (List.mem roleR participant_roles))
                   !notrole_roles
    in
    let subset_condition =
      List.for_all (fun roleR -> List.mem roleR participant_roles)
                   !hasrole_roles
    in
    intersection_empty && subset_condition
  ) unique_participants
```

The above snippet of code implements the role satisfaction check from Definition III.7. At line 4, unique participant identifiers are extracted from the participant variables using the ι mapping. For each unique participant (line 7), the function retrieves the participant's current role set $\pi(\alpha)$ at line 8. At line 9, all party variables mapping to the same participant are collected. The function then collects roles marked as ■ (Top) at line 12 and roles marked as ▣ (Bottom) at line 19 across all party variables for this participant. The satisfaction check verifies two conditions: at line 26, it ensures that no roles marked as Bottom are present ($\text{participant\_roles} \cap \text{notrole\_roles} = \emptyset$), and at line 30, it ensures that all roles marked as Top are present ($\text{hasrole\_roles} \subseteq \text{participant\_roles}$). The function returns `true` only if both conditions hold for all participants, corresponding to $\langle \pi, \sigma' \rangle \models \rho$ from Definition III.7.

The `update_pi` function implements role update semantics, computing the new role mapping by adding roles marked as Top and removing roles marked as Bottom, implementing $\mathsf{rupd}(\sigma', \pi, \rho') = \pi'$ of Definition III.9:

```
let update_pi (pi : pi_type) (iota : iota_type)
                 (rho : role_a_type) (party_list: ptp_var list)
                 (roles: role_type list) : pi_type =
  fun alpha ->
    let original_roles = pi alpha  in
```

```
6     let ptp_vars = List.filter (fun ptp -> iota ptp = alpha) party_list  in
7     let roles_to_add = ref []  in
8     List.iter (fun ptp_var ->
9       List.iter (fun role ->
10        match rho ptp_var role with
11        | Top -> if not (List.mem role !roles_to_add) &&
12                    not (List.mem role original_roles)
13                 then roles_to_add := !roles_to_add @ [role]
14        | _ -> ()
15      ) roles
16    ) ptp_vars;
17    let roles_to_remove = ref []  in
18    List.iter (fun ptp_var ->
19      List.iter (fun role ->
20        match rho ptp_var role with
21        | Bottom -> if not (List.mem role !roles_to_remove)
22                    then roles_to_remove := !roles_to_remove @ [role]
23        | _ -> ()
24      ) roles
25    ) ptp_vars;
26    let updated_roles = original_roles @ !roles_to_add  in
27    List.filter (fun r -> not (List.mem r !roles_to_remove)) updated_roles
```

The above snippet of code implements the role update function $\mathsf{rupd}(\sigma', \pi, \rho') = \pi'$ from Definition III.9. for a given participant α. At line 5, it retrieves the participant's current role set $\pi(\alpha)$. At line 6, it filters the party variables to those mapping to participant α. The update proceeds in two phases: at line 7, roles marked as ■ (Top) are collected for addition, ensuring no duplicates and that roles not already present are added. At line 17, roles marked as ▣ (Bottom) are collected for removal. Finally, at line 26, the updated role set is computed by first concatenating the original roles with roles to add, then filtering out roles marked for removal, implementing $\mathsf{rupd}(\sigma', \pi, \rho') = \pi'$ from Table III.2.

IV.3.5 Transition and Network Execution

Transition execution implements the core semantics from Table III.2, specifically the [TR] rule. Network execution executes transitions across multiple coordinators, to implement the network semantics from Definition III.12.

The `check_transition` function validates all preconditions for a transition. It checks state matching (`q = q_from`), operation matching (`op = operation`), updates the environment with operation parameters via `substitute_values`, evaluates the guard expression using `eval`, executes call tries sequentially via `eval_func_write_list`, validates role constraints before and after the transition using `role_satisf`, and applies assignments by evaluating each assignment expression and updating the environment. The function returns `Some new_config` if all preconditions are satisfied, or `None` with a failure reason otherwise.

The `perform_transition_impl` function orchestrates transition execution at the network level. It prevents reentrancy by checking if the coordinator is already in `called_contracts` and adding it to the list before execution. It filters transitions matching the current state and operation, then iteratively calls `check_transition` on each candidate transition until one succeeds. If a transition succeeds, it updates the network configuration by storing the new coordinator configuration in `config_map` and the updated coordinator definition in `edam_map`. If all transitions fail, it reverts to the original network configuration, maintaining atomicity.

The `evaluate_trace` function processes sequences of transition invocations for test generation. It takes a trace (a list of coordinator names and label configurations) and the initial network configuration. For each trace element, it generates the participant mapping `iota` from the label, calls `perform_transition_impl` to execute the transition, and accumulates the results with success/failure status, updated environment, and state. The function returns an evaluated trace where each step is annotated with its execution outcome, enabling test generation to validate expected versus actual behaviour. The algorithm 1 summarises the transition and network execution process.

Algorithm 1 Transition and Network Execution

procedure CHECKTRANSITION(coordinator, config, transition, label, ι, network)
  if config.state $\neq$ transition.source_state $\vee$ label.operation $\neq$ transition.operation **then**
    return $\perp$
  end if
  $\sigma' \leftarrow$ substitute_values(config.σ, transition.params, label.values)
  guard_val $\leftarrow$ eval(σ', ι, transition.guard, network)
  calls_success $\leftarrow$ eval_func_write_list(transition.calls, σ', ι, network)
  if guard_val $\neq$ true $\vee \neg$calls_success $\vee \neg$role_satisf(config.λ, ι, transition.ρ) **then**
    return $\perp$
  end if
  $\lambda' \leftarrow$ update_lambda(config.λ, ι, transition.ρ', roles)
  $\sigma'' \leftarrow$ evaluate_assignments(σ', ι, transition.assignments, network)
  return {state : transition.target_state, $\lambda : \lambda', \sigma : \sigma''$}
end procedure
procedure PERFORMTRANSITION(coordinator_name, label, network, called_contracts)
  if coordinator_name $\in$ called_contracts **then fail** "Reentrancy"
  end if
  candidates $\leftarrow$ {$t \in$ coordinator.transitions $\mid$ t.source_state = config.state $\wedge$ t.operation = label.operation}
  for each transition $\in$ candidates **do**
    result $\leftarrow$ CheckTransition(coordinator_name, config, transition, label, ι, network)
    if result $\neq \perp$ **then**
      network.config_map[coordinator_name] $\leftarrow$ result
      return (result, network)
    end if
  end for
  return ($\perp$, network)
end procedure
procedure EVALUATETRACE(trace, initial_network)
  current_network $\leftarrow$ initial_network
  for each (coordinator_name, label) $\in$ trace **do**
    (config, network) $\leftarrow$ PerformTransition(coordinator_name, label, current_network, [])
    result_trace.append((coordinator_name, label, config $\neq \perp$, config))
    current_network $\leftarrow$ network
  end for
  return result_trace
end procedure

IV.3.6 Test Generation

The test generation process for EDAM provides a systematic validation mechanism to verify that the generated smart contract code correctly implements the behavioural specifications defined by the EDAM. This process leverages the interpreter for EDAM execution described in Section IV.3.1 to simulate and evaluate execution traces before generating executable test cases. The test generation framework is implemented in OCaml, utilising the formal semantics of EDAM to ensure that generated tests accurately reflect the intended behaviour of the coordination models.

The test generation process addresses a fundamental challenge in smart contract validation: bridging the gap between abstract behavioural specifications and concrete executable test cases. By systematically exploring the state space of EDAM networks and generating diverse test scenarios, the framework provides comprehensive coverage of both successful execution paths and failure cases, enabling thorough validation of the generated Solidity contracts.

The test generation methodology consists of three sequential stages that transform abstract

behavioural specifications into concrete, executable test suites:

- *Symbolic trace generation*: This stage produces symbolic traces, which are sequences of transitions over the product automaton of the EDAMs under analysis, augmented with auxiliary metadata. Each symbolic trace represents a potential execution path through the network of coordinators, where variables and participants remain uninstantiated. The generation process randomly selects a finite set of symbolic traces, each bounded by a configurable maximum length.
- *Concrete trace derivation*: This stage instantiates symbolic traces by assigning concrete values to variables (data values and participants), producing concrete traces. A concrete trace is a sequence of labels from the labelled transition semantics defined in Section III.2.3, along with execution metadata. The instantiation process employs a hybrid approach combining random value generation with constraint solving techniques. Notably, concrete trace execution may result in failures when assigned values violate guard conditions, role constraints, or other transition requirements such as call tries. Both successful and failed executions are recorded, as they represent valid behavioural scenarios that must be validated against the generated contracts.
- *Test suite generation*: The final stage transforms concrete traces into structured JavaScript test cases executable against the generated Solidity contracts. Each test case encodes the expected execution outcome (success or failure) based on the EDAM semantics, along with appropriate assertions for validation. The complete test suite provides comprehensive coverage of the behavioural specifications.

Symbolic Trace Generation: The first stage generates symbolic traces through controlled simulation of the network of EDAMs. A *symbolic configuration* is a network configuration (as defined in Definition III.12) where only the current state of the coordinator undergoing a transition is updated after transition selection. The simulation begins from the initial symbolic configuration, where each EDAM is in its initial state after executing the `start` operation for each coordinator in the network.

The simulation proceeds iteratively: at each step, a coordinator is randomly selected from the network, followed by random selection of an available transition from its current state. The selection process ensures uniform distribution across all coordinators in the network, enabling comprehensive exploration of inter-coordinator interaction patterns. The selected transition is recorded in the trace as a tuple $\langle \text{Model}, \text{Transition}, \text{Label} \rangle$, where the label contains symbolic placeholders for participants and data values that will be instantiated in the subsequent stage. The coordinator's state is updated locally to reflect the transition's effects, maintaining a symbolic configuration where only the state of the coordinator undergoing the transition is modified.

Importantly, call tries within the selected transition are not executed during symbolic trace generation, meaning that state changes in other coordinators are not propagated to the configuration. This design choice maintains the symbolic nature of the trace while exploring potential execution paths without requiring concrete value instantiation. The simulation terminates when either the maximum trace length is reached or no further transitions are available from any coordinator in the current configuration. The initial symbolic configuration is established by executing the `start` operation for each coordinator in the network, ensuring that all coordinators begin in their designated initial states with appropriate role assignments and environment mappings.

Algorithm 2 formalizes the symbolic trace generation process. Figure IV.5 provides an example of a symbolic trace generated for the network of EDAMs from the AMM coordinator model with two ERC20 Token (Token 1 and Token 2). The trace shows the sequence of transitions selected by the simulator, where at each step a coordinator is randomly selected and a transition is selected from its current state.

The maximum trace length is user-configurable through the `max_trace_size` parameter parameter and serves to bound the search space and ensure computational tractability. This parameter controls the depth of exploration during symbolic trace generation, balancing between comprehensive coverage and generation time. The minimum trace length corresponds to the shortest path from the initial configuration to a terminal state (a state without outgoing transitions), if such states exist in the network. The symbolic trace generation process produces multiple independent traces, each representing a distinct execution path through the network. The number of symbolic traces

```
=== Symbolic Trace #1 ===
Step #1_C20_2: p -- _ start(s, b, n, d) -> q1
Step #2_C20: p -- _ start(s, b, n, d) -> q1
Step #3_AMM: owner -- _ start() -> S_Deployed
Step #4_AMM: user -- S_Deployed addLiquidity(_amountA, _amountB) -> S_LiquidityAdded
Step #5_C20: p -- q1 transferFrom(s, r, a) -> q1
Step #6_AMM: user -- S_LiquidityAdded swapBForA(_amountB) -> S_LiquidityAdded
Step #7_AMM: user -- S_LiquidityAdded removeLiquidity(_lpAmount) -> S_LiquidityAdded
Step #8_C20: p -- q1 transfer(r, a) -> q1
Step #9_AMM: user -- S_LiquidityAdded swapAForB(_amountA) -> S_LiquidityAdded
Step #10_C20_2: p -- q1 burn(a) -> q1
Step #11_C20_2: p -- q1 transfer(r, a) -> q1
Step #12_AMM: user -- S_LiquidityAdded swapAForB(_amountA) -> S_LiquidityAdded
Step #13_C20: p -- q1 transfer(r, a) -> q1
Step #14_C20_2: p -- q1 transferFrom(s, r, a) -> q1
Step #15_C20_2: p -- q1 burn(a) -> q1
Step #16_C20: p -- q1 mint(r, a) -> q1
Step #17_C20: p -- q1 burn(a) -> q1
Step #18_AMM: user -- S_LiquidityAdded swapBForA(_amountB) -> S_LiquidityAdded
Step #19_C20: p -- q1 approve(s, a) -> q1
Step #20_AMM: user -- S_LiquidityAdded removeLiquidity(_lpAmount) -> S_LiquidityAdded
Step #21_C20: p -- q1 mint(r, a) -> q1
Step #22_C20_2: p -- q1 transfer(r, a) -> q1
Step #23_C20_2: p -- q1 mint(r, a) -> q1
=== End of Trace ===
```

Figure IV.5: Symbolic trace generation process

Algorithm 2 Symbolic Trace Generation Algorithm

procedure GENERATESYMBOLICTRACE(Network, MaxLength)
  Trace ← ∅
  CurrentConfig ← InitialConfiguration(Network)
  while Length(Trace) < MaxLength and HasAvailableTransitions(CurrentConfig) **do**
    SelectedModel ← SelectRandomModel(CurrentConfig)
    AvailableTransitions ← GetAvailableTransitions(SelectedModel)
    SelectedTransition ← SelectRandomTransition(AvailableTransitions)
    Label ← CreateLabel(SelectedModel, SelectedTransition)
    Trace ← Trace ∪ {(*SelectedModel*, *SelectedTransition*, *Label*)}
    CurrentConfig ← UpdateConfiguration(SelectedModel, SelectedTransition)
  end while
  return Trace
end procedure

generated is configurable, allowing users to control the breadth of exploration. Each symbolic trace serves as a template for generating multiple concrete traces in the subsequent stage, where concrete values are assigned to the symbolic placeholders.

Concrete Trace Derivation: The second stage instantiates symbolic traces by assigning concrete values to variables and participants, producing executable concrete traces. This transformation bridges the gap between abstract behavioural descriptions and executable test scenarios. The derivation process handles two categories of parameters:

(i) Participant Variables: Participant assignment follows configurable simulation policies controlled by several parameters. The `probability_new_participant` parameter determines the probability of generating a new participant identity versus reusing an existing one. When new participant generation is enabled, participants are created randomly with a specified probability distribution. Otherwise, participants are selected from the set of participants introduced in previous trace steps, with preference given to those satisfying role constraints ρ when such constraints are present. The `probability_right_participant` parameter controls the likelihood of selecting participants that satisfy role requirements versus those that do not. If no suitable participants exist from previous steps, new participants are generated randomly.

(ii) Data Value Generation: Data values are initially generated using random sampling. The random generation process respects type constraints, generating values of appropriate types (integers, booleans, strings, addresses) within configurable bounds. If random generation fails to produce values satisfying guard conditions after a configurable number of attempts (controlled by `max_fail_try`), the Z3 SMT solver [68] is invoked to find satisfying assignments. The Z3 integration translates guard conditions into SMT-LIB constraints, including arithmetic operations, boolean logic, and comparisons. When Z3 finds a satisfying concrete values, these values are used to instantiate the trace. Each attempt, whether successful or not, is recorded as a separate concrete trace. Further configuration parameters can be made to control the data

value generation process.

- `min_int_value` and `max_int_value`: The minimum and maximum values for integer data.
- `min_gen_string_length` and `max_gen_string_length`: The minimum and maximum length for string data.
- `max_gen_array_size`: The maximum size for array data.
- `probability_true_for_bool`: The probability of generating a true value for boolean data.

For each concrete trace, a *label* is constructed by substituting the concrete participants and data values into the symbolic label from the corresponding symbolic trace. This label encapsulates all information necessary to invoke the transition on a deployed contract: the target coordinator, the operation name, participant identities, and data values. The label follows the format $\mathfrak{p} : \mathfrak{c}.\mathsf{op}(\mathfrak{v}_1, \ldots, \mathfrak{v}_n)$ from the labelled transition semantics, where $\mathfrak{p}$ is the caller participant, $\mathfrak{c}$ is the coordinator identifier, op is the operation name, and $\mathfrak{v}_1, \ldots, \mathfrak{v}_n$ is the list of concrete parameter values.

The concrete trace is then executed using the OCaml implementation of the EDAM interpreter (as described in Section IV.3.1) to determine the expected execution outcome (success or failure) according to the formal semantics. The interpreter evaluates the transition by checking guard conditions, role constraints, and executing call tries, updating the network configuration accordingly. The execution result, encoded as an *expectedFailure* flag, is attached to the trace along with the resulting network configuration (including updated states, role mappings, and environment mappings) for use in test case generation. This execution phase ensures that the expected behaviour encoded in test cases accurately reflects the formal semantics of EDAM, which is used to validate the generated contracts.

The interpreter generates the textual representation of test cases in JavaScript format, which is then transmitted to a Python-based processing pipeline. The Python component parses the generated test text, extracts trace information, and orchestrates the compilation and execution of the OCaml test generation code. This pipeline manages the transformation of interpreter output into executable JavaScript test suites, handles file system operations for test artifacts, and coordinates the integration with the Hardhat testing framework. The pipeline ensures that test cases are properly formatted, that contract deployment follows dependency ordering (respecting inter-coordinator dependencies), and that test assertions are correctly generated based on the execution results from the interpreter. Algorithm 3 details the concrete trace derivation procedure.

Test Suite Generation: The final stage converts concrete traces into executable JavaScript test suites targeting the generated Solidity contracts. Each concrete trace, represented as a tuple $\langle \text{Model}, \text{Label}, \text{expectedFailure} \rangle$, is transformed into a structured test case following a standardised template.

The test case generation process includes the following components:

- Contract deployment and setup: Each test case initialises the network of coordinator contracts with their initial states and role assignments. This setup phase establishes the execution environment matching the initial configuration used during trace generation. Contract deployment follows a topological ordering based on inter-coordinator dependencies, ensuring that contracts are deployed in an order that respects call try relationships. For each coordinator, the `start` operation is invoked with appropriate parameters to establish the initial state, role mappings, and environment. Participant accounts are mapped to Hardhat test accounts, maintaining the participant identity relationships established during trace generation.
- Transition execution: The test invokes the specified operation on the target coordinator contract using the participants and data values encoded in the trace label. The invocation is wrapped in appropriate error handling to capture both successful executions and expected failures. For operations involving inter-coordinator calls, the test ensures that the target coordinator contracts are properly deployed and accessible, with addresses correctly passed as parameters. The execution uses Hardhat's contract interaction mechanisms, allowing tests to simulate realistic blockchain transaction scenarios.

Algorithm 3 Concrete Trace Derivation

```
1: procedure ConcreteTrace(Network, symbolicTrace, MaxAttempts, allowNewP)
2:     concreteTraces ← []
3:     allParticipants ← ∅
4:     for each trace in symbolicTrace do
5:         (Model, Trans, Label) ← trace
6:         participants ← GetParticipants(Label, allowNewP, allParticipants)
7:         allParticipants ← allParticipants ∪ participants
8:         attempts ← MaxAttempts
9:         guardsSatisfied ← false
10:        while attempts > 0 and not guardsSatisfied do
11:            dataValues ← generateRandomData(Trans.variables)
12:            guardsSatisfied ← evalGuard(Trans.guard, dataValues, participants)
13:            newLabel ← createLabel(Label, Model, Trans, dataValues, participants)
14:            expectedFailure ← execTransition(Network, newLabel)
15:            concreteTraces ← concreteTraces ∪ {Model, newLabel, expectedFailure}
16:            attempts ← attempts − 1
17:        end while
18:        if not guardsSatisfied then
19:            z3Result ← z3Solver.solve(Trans.guard, Trans.variables, participants)
20:            if z3Result.satisfiable then
21:                dataValues ← z3Result.model
22:                newLabel ← createLabel(Label, Model, Trans, dataValues, participants)
23:                expectedFailure ← execTransition(Network, newLabel)
24:                concreteTraces ← concreteTraces ∪ {Model, newLabel, expectedFailure}
25:            end if
26:        end if
27:    end for
28:    return concreteTraces
29: end procedure
```

- Assertions for expected behaviour: Based on the *expectedFailure* flag from the concrete trace, the test validates that the transition execution matches the predicted outcome. Successful transitions are verified to complete without exceptions using Hardhat's expect (...). to.not.be.reverted assertion, while expected failures are verified to raise appropriate exceptions or revert conditions using expect (...). to.be.reverted. This validation ensures that the generated Solidity contracts correctly implement the guard conditions, role constraints, and other transition requirements specified in the EDAM.
- Data value assertions: After successful transition execution, the test verifies that field updates specified in the transition are correctly reflected in the contract's persistent state. This includes checking that state variables are updated to their expected values according to the transition's update expressions. The assertions compare the contract's state after execution with the expected state computed by the interpreter, ensuring that assignment semantics are correctly implemented. These assertions are configurable through the `add_test_of_variables` parameter, allowing users to control the granularity of state validation.
- Role change assertions: When enabled (via the `add_pi_to_test` configuration parameter), the test validates that role assignments and updates specified in the transition are properly applied. This ensures that participants acquire or maintain the expected roles after successful transition execution, verifying the correctness of role management logic. The assertions check both pre-transition role constraints (specified in ρ) and post-transition role updates (specified in ρ'), ensuring that role-based access control is correctly implemented in the generated contracts.
- State transition assertions: When enabled (via the `add_test_of_state` configuration parameter), the test verifies that the coordinator transitions to the expected target state after successful execution. This validation ensures that state machine semantics are correctly implemented, with state transitions occurring only when all transition requirements are satisfied.

Figure IV.6 shows the transformation of a concrete trace into structured JavaScript test cases.

The test suite is generated from the symbolic trace in Fig. IV.5. The configuration allows to try the concrete trace just once in this case. So maximum number of attempts is 2 thus a call can fail at most two times. As we can see from the figure, the call to `addLiquidity` fails twice, both from the external failure.

```
it("Trace #1", async function () {
  const participant__0 = accounts[0];
  const contract_factory_C20_2 = await ethers.getContractFactory("C20_2", participant__0);
  const instance_C20_2 = await contract_factory_C20_2.deploy(43, "tczstazspu", "gwgpagale", 81);
  const C20_2 = instance_C20_2.target;
  const contract_factory_C20 = await ethers.getContractFactory("C20", participant__0);
  const instance_C20 = await contract_factory_C20.deploy(6, "djvmu", "amcoc", 24);
  const C20 = instance_C20.target;
  const contract_factory_AMM = await ethers.getContractFactory("AMM", participant__0);
  const instance_AMM = await contract_factory_AMM.deploy(instance_C20.target, instance_C20_2.target);
  const AMM = instance_AMM.target;
  await expect(instance_AMM.connect(participant__0).addLiquidity(61, 50)).to.be.reverted;
  await expect(instance_AMM.connect(participant__0).addLiquidity(1, 1)).to.be.reverted;
  await expect(instance_C20.connect(participant__0).transferFrom(C20_2, participant__0, 98)).to.be.reverted;
  await expect(instance_C20.connect(participant__0).transferFrom(C20_2, AMM, 0)).to.not.be.reverted;
  await expect(instance_AMM.connect(participant__0).swapBForA(78)).to.be.reverted;
  await expect(instance_AMM.connect(participant__0).swapBForA(29)).to.be.reverted;
  await expect(instance_AMM.connect(participant__0).removeLiquidity(50)).to.be.reverted;
  await expect(instance_AMM.connect(participant__0).removeLiquidity(45)).to.be.reverted;
  await expect(instance_C20.connect(participant__0).transfer(participant__0, 29)).to.be.reverted;
  await expect(instance_C20.connect(participant__0).transfer(AMM, 4)).to.not.be.reverted;
  await expect(instance_AMM.connect(participant__0).swapAForB(18)).to.be.reverted;
  await expect(instance_AMM.connect(participant__0).swapAForB(35)).to.be.reverted;
  await expect(instance_C20_2.connect(participant__0).burn(9)).to.not.be.reverted;
  await expect(instance_C20_2.connect(participant__0).transfer(participant__0, 46)).to.be.reverted;
  await expect(instance_C20_2.connect(participant__0).transfer(C20, 10)).to.not.be.reverted;
  await expect(instance_AMM.connect(participant__0).swapAForB(84)).to.be.reverted;
  await expect(instance_AMM.connect(participant__0).swapAForB(62)).to.be.reverted;
  await expect(instance_C20.connect(participant__0).transfer(participant__0, 25)).to.be.reverted;
  await expect(instance_C20.connect(participant__0).transfer(AMM, 25)).to.be.reverted;
  await expect(instance_C20_2.connect(participant__0).transferFrom(AMM, C20, 16)).to.be.reverted;
  await expect(instance_C20_2.connect(participant__0).transferFrom(AMM, AMM, 0)).to.not.be.reverted;
  await expect(instance_C20_2.connect(participant__0).burn(31)).to.be.reverted;
  await expect(instance_C20_2.connect(participant__0).burn(48)).to.be.reverted;
  await expect(instance_C20.connect(participant__0).mint(participant__0, 41)).to.not.be.reverted;
  await expect(instance_C20.connect(participant__0).burn(7)).to.not.be.reverted;
  await expect(instance_AMM.connect(participant__0).swapBForA(76)).to.be.reverted;
  await expect(instance_AMM.connect(participant__0).swapBForA(91)).to.be.reverted;
  await expect(instance_C20.connect(participant__0).approve(participant__0, 66)).to.not.be.reverted;
  await expect(instance_AMM.connect(participant__0).removeLiquidity(21)).to.be.reverted;
  await expect(instance_AMM.connect(participant__0).removeLiquidity(71)).to.be.reverted;
  await expect(instance_C20.connect(participant__0).mint(AMM, 98)).to.not.be.reverted;
  await expect(instance_C20_2.connect(participant__0).transfer(AMM, 42)).to.be.reverted;
  await expect(instance_C20_2.connect(participant__0).transfer(participant__0, 72)).to.be.reverted;
  await expect(instance_C20_2.connect(participant__0).mint(AMM, 96)).to.not.be.reverted;
});
```

Figure IV.6: JavaScript test suite generation from concrete traces

The generated test cases are structured for execution in the Hardhat testing framework [86], which provides comprehensive testing infrastructure for Solidity smart contracts, including contract deployment, transaction simulation, and assertion libraries. Each test case is structured as a Mocha test function within a Hardhat test suite, with proper setup and teardown handling. The test generation process produces a complete JavaScript test file that can be executed directly using Hardhat's test runner, providing automated validation of the generated contracts.

Test execution results provide direct feedback on the correctness of the code generation process: successful test executions indicate that the generated contracts correctly implement the EDAM specifications, while test failures reveal discrepancies between the model semantics and the contract implementation. The test generation framework tracks statistics including the number of successful and failed transitions, providing metrics on test coverage and execution outcomes. These metrics enable quantitative assessment of the quality of both the generated test suites and the underlying code generation process, supporting iterative improvement of the code generation algorithms. For the example we generated one symbolic trace and set the configuration to generate five concrete traces with maximum number of attempts set to 2. Figure IV.7 demonstrates the execution of the generated test suite against the deployed Solidity contracts in the Hardhat environment. As we can see from the figure, the test suite passes successfully.

```
Compiled 3 Solidity files successfully (evm target: paris).

  EDAMs Tests
    ✔ Trace #1 (283ms)
    ✔ Trace #2 (236ms)
    ✔ Trace #3 (277ms)
    ✔ Trace #4 (300ms)
    ✔ Trace #5 (222ms)

  5 passing (14s)
```

Figure IV.7: Execution of JavaScript test suite in Hardhat testing framework

IV.4 EDAM Studio

EDAM Studio is the software implementation of the compilation and validation architecture developed in this chapter. Its input is a *network of coordinating EDAMs* in the sense of Definition III.12; after parsing and static checking, that network is held in the serial form fixed by Definition IV.3, i.e., a single JSON document that preserves coordinators, states, transitions, roles, fields, guards, and call tries. From this one checked document the studio invokes three cooperating tools already introduced above: the OCaml interpreter (Section IV.3.1), which executes the operational semantics and supports symbolic trace construction; the `JSON2Solidity` synthesiser, which maps each coordinator to a Solidity contract; and the test generator tied to Section IV.3.6, which derives concrete traces and emits JavaScript test suites for execution with Hardhat. Because the interpreter, the Solidity back end, and the test back end all read the same JSON snapshot, there is no secondary "test-only" or "code-only" copy of the specification that could diverge silently.

Specification maintenance is therefore a matter of editing that shared serialisation. The studio provides two specification front-ends: a *graphical editor* in which the network is drawn and revised on a canvas, and a *textual editor* for the concrete EDAM syntax with parser-backed assistance. The graphical editor is intended for direct construction of coordinators, local states, transitions, guards, roles, and data dependencies, and for reading off control-flow structure without writing textual clauses line by line. The textual editor is intended for workflow based on version control (e.g., unified diffs), structured refactoring, and automation (e.g., batch regeneration through the supplied command-line interface or calls to the studio's HTTP API), where a diagram is not the primary working surface. Regardless of which front-end is used, persistence round-trips through the same JSON schema, so the interpreter, Solidity synthesis, and test synthesis always see logically identical input.

For persistence, replication, and independent scrutiny, the reference implementation is released as open-source software archived on Zenodo [123], with ongoing development mirrored in a public repository. [2] The package was evaluated under the ECOOP 2026 artifact evaluation process and received all badges awarded in that round, namely those for availability, functional adequacy, and reusability under the committee's published criteria. The distribution comprises the studio services (graphical and programmatic interfaces), build and deployment automation, the generators whose design motivated this thesis, and the ancillary artefacts needed to instantiate the toolchain in a fresh environment and to repeat the code- and test-generation workflow end-to-end.

[2] Sources and installation: https://github.com/loctet/Edam-Studio.

Chapter V: Validation of smart contract behaviour

This chapter validates the proposed approach along three complementary dimensions: expressiveness of the modelling framework, quality of generated code and tests, and mitigation of common smart-contract vulnerabilities. To this end, we evaluate our method on a benchmark of smart contracts from the Azure repository, three well-known smart contracts from the Ethereum ecosystem (namely AMM, ERC20 Token, and SimpleWallet), and report both functional and testing-oriented results. We also evaluate the mitigation of common smart-contract vulnerabilities through the model-driven design approach.

The benchmark we use for validating our approach consists of (*i*) the one used in [3] from the specifications of the smart contracts in the Azure repository [148] and (*ii*) those for AMM [174], ERC20 Token [202], and SimpleWallet [200]. Our validation considers expressiveness (cf. Section V.1) as well as the quality of both code and test suites automatically generated from EDAMs (cf. Section V.2). These examples exhibit a variety of features that are essential for the representation of smart contracts. We consider a significant range of features in our analysis, including (i) *ICI*: *I*nter-*C*oordinator *I*nteractions, (ii) *BI*: participant joining *B*y *I*nvoking an operation, (iii) *PP*: participant joining by *P*assing *P*arameters, (iv) *RR*: *R*ole *R*evocation, (v) *MPR*: *M*ulti-*P*articipant *R*ole.

						Time (min.)		Coverage (%)				
	ICI	*BI*	*PP*	*RR*	*MPR*	Gener.	Exec.	Stmts	Brch	Ops	LoC	Mut. Score
AssetTransfer	⊖	✓	✓	✓	⊖	0.4	8	93.22	94.12	100	95.83	81.4
BasicProvenance	⊖	✓	✓	✓	⊖	5.6	4	94.12	91.67	100	96.15	82.95
Bazaar	✓	✓	⊖	⊖	⊖	–	–	–	–	–	–	–
DefectiveCounter	⊖	✓	✓	⊖	⊖	0.4	0.8	81.25	62.5	100	88	71.42
DigitalLocker	⊖	✓	✓	✓	✓	1.5	5	85.29	80.77	100	87.5	77.40
FrequentFlyer	⊖	✓	✓	⊖	⊖	2.46	4	85	70	100	90.63	73.21
HelloBlockchain	⊖	✓	⊖	⊖	⊖	0.9	4	75	60	100	84	66.66
Ping Pong	✓	✓	⊖	⊖	⊖	–	–	–	–	–	–	–
RefrigeratedTransport	⊖	✓	✓	✓	✓	1.9	5	95.83	95.45	100	98.25	79.76
Simple Marketplace	⊖	✓	⊖	✓	⊖	2.8	6	82.35	75	100	89.29	78.94
ThermostatOperation	⊖	⊖	✓	⊖	⊖	6.8	3	84.21	75	100	90.91	78.26
AMM	✓	✓	✓	⊖	✓	24.65	14	74.49	62.86	92.59	83.84	62.70
ERC20 Token	⊖	✓	✓	⊖	✓	12.33	7	80	75	100	90.91	79.22
SimpleWallet	⊖	✓	⊖	⊖	⊖	2.75	1	81.25	75	100	88.46	86.44

Table V.1: Expressiveness and quality results

Table V.1 summarises our results for each smart contract in our benchmark (first column; the last three rows are not considered in [3]) together with the features of each contract as reported in [3] (columns 2 to 6).[1] Columns 7 and 8 report the time (in minutes)[2] respectively to generate a

[1]Symbol ✓indicates that the feature is required for the contract and it is supported by the model; features not required by the contract are assigned the ⊖ symbol.

[2]Experiments performed on a Dell XPS 8960, 13th Gen Intel Core (9 - 13900K) with 32 cores and 32GB RAM running Linux 6.5.0-44-generic (Ubuntu 24.04.2, 64bit).Generation and execution time are obtained by averaging

test suite containing 10000 tests and to execute them on the Solidity code, the next three columns report our results on coverage (for which we rely on the Hardhat framework [86]), and the last column on the mutation score (computed by ReSuMo [28]).

V.1 Expressiveness

We evaluate the expressiveness of EDAM using the Azure benchmark [148] and three well-known smart contracts from the Ethereum ecosystem (namely AMM, ERC20 Token, and SimpleWallet). As shown in Table V.1, EDAM can capture key features commonly encountered in realistic smart contracts (explored also by other authors, e.g., [96]). These features represent fundamental patterns in decentralised coordination scenarios, enabling EDAM to model a wide range of real-world smart contract behaviour.

The expressiveness analysis considers five essential features that are critical for modelling realistic coordination scenarios: (i) *ICI*: *I*nter-*C*oordinator *I*nteractions, (ii) *BI*: participant joining *B*y *I*nvoking an operation, (iii) *PP*: participant joining by *P*assing *P*arameters, (iv) *RR*: *R*ole *R*evocation, (v) *MPR*: *M*ulti-*P*articipant *R*ole.

In the following, we examine each feature in detail, explaining how EDAM supports it natively and providing concrete real-world scenarios where each feature is essential.

Inter-Coordinator Interactions (*ICI*)

Inter-coordinator interactions enable one coordinator to invoke operations on another coordinator as part of a transition execution. This feature is fundamental for modelling distributed systems where functionality is decomposed across multiple components that must coordinate to achieve complex behaviour.

In EDAM, inter-coordinator interactions are expressed through *call tries* $\mathfrak{c}_i.\texttt{op}(\mathfrak{v}_1, \ldots, \mathfrak{v}_n)$, which allow a coordinator to specify external calls to other coordinators within a transition. Call tries are evaluated as part of the transition's guard condition, and their success or failure determines whether the transition can proceed.

Real-world scenarios:

- **AMMs:** An Automated Market Maker (AMM) coordinator (AMM) needs to interact with an underlying token coordinator (ERC20 Token) to transfer tokens when executing swaps. When a user requests to swap tokens, the AMM coordinator calls the token coordinator's `transferFrom` operation to move tokens from the user's account, and then calls `transfer` to send tokens to the user. This interaction pattern is essential for building composable DeFi protocols where different coordinators handle different aspects of the trading process.
- **Payment Processing:** A payment coordinator processes payments by calling a token coordinator to transfer funds. When processing a payment, the payment coordinator verifies the payment amount and then invokes the token coordinator's transfer operation to move tokens from the payer to the payee. This separation allows the payment coordinator to focus on payment logic while delegating token management to a specialised coordinator.
- **Escrow Services:** A marketplace coordinator might call an escrow coordinator to hold funds during a transaction. When a buyer initiates a purchase, the marketplace coordinator calls the escrow coordinator to lock the payment, ensuring that funds are held securely until the transaction is completed or cancelled. This pattern enables trustless transactions in decentralised marketplaces.

The AMM contract exemplifies native support for *ICI* through call tries that enable the Automated Market Maker (AMM) coordinator to interact with the underlying ERC20 Token coordinator for token transfers.

the times of 10 runs.

Participant Joining by Invocation (*BI*)

The *BI* feature enables participants to join a coordinator by invoking one of its operations. When a participant calls an operation, they automatically become part of the coordinator's participant set if they were not already registered. This mechanism allows coordinators to have open membership, where any participant can join by performing an action, rather than requiring explicit registration or initialisation steps.

In EDAM, participant joining by invocation is handled automatically: when a transition is executed with a caller $\mathfrak{p}$ who is not yet in the coordinator's environment, the environment is extended to include the mapping from a participant variable to $\mathfrak{p}$. The caller is represented by a participant variable p in the transition (typically the ptpVar field in the JSON representation), and after successful execution, $\sigma(\mathsf{p}) = \mathfrak{p}$ establishes the participant's identity within the coordinator.

Real-world scenarios:

- **Decentralized Counters:** In a counter coordinator (DefectiveCounter), any participant can increment the counter by calling the increment operation. The first time a participant calls this operation, they automatically join the coordinator and their identity is recorded. Subsequent calls from the same participant are recognised through the established environment mapping.
- **Simple Wallets:** A wallet coordinator (SimpleWallet) allows any participant to deposit funds by calling a deposit operation. When a user first deposits, they join the wallet coordinator and gain the ability to later withdraw their funds. This pattern enables simple, permissionless financial services where users can interact without prior registration.
- **Token Transfers:** In a token coordinator, participants can receive tokens by having other participants call transfer operations targeting them. When a transfer operation is invoked with a recipient parameter, if the recipient is not yet registered, they join the coordinator through the transfer operation, enabling seamless token distribution without requiring recipients to explicitly register first.

The *BI* feature is natively supported in EDAM through the caller mechanism, where the participant invoking an operation automatically becomes part of the coordinator's participant set, as demonstrated in contracts such as DefectiveCounter, FrequentFlyer, ERC20 Token, and SimpleWallet.

Participant Joining by Parameters (*PP*)

The *PP* feature enables participants to join a coordinator by being passed as parameters to an operation, even if they are not the caller. This mechanism allows coordinators to establish relationships between multiple participants in a single operation call, enabling scenarios where one participant can introduce others to the coordinator.

In EDAM, when a transition includes participant variables in its parameter list (represented by the ptpVarList field in the JSON representation), these variables are evaluated in the context of the caller's environment. If a participant variable p_i in the parameter list maps to a participant identity $\mathfrak{p}$ that is not yet in the coordinator's environment, the environment is extended to include $\sigma(\mathsf{p}_\mathsf{i}) = \mathfrak{p}$. This allows multiple participants to join simultaneously through a single operation invocation.

Real-world scenarios:

- **Asset Management:** In an asset coordinator (AssetTransfer), an owner can create an asset and assign it to a beneficiary by calling an operation with the beneficiary as a parameter. The beneficiary joins the coordinator through this parameter, gaining access rights to the asset without needing to explicitly register or invoke an operation themselves. This pattern enables scenarios where asset ownership or access is delegated to others.
- **Escrow Services:** An escrow coordinator allows a buyer to initiate an escrow by specifying a seller as a parameter. When the buyer calls the escrow creation operation with the seller's identity as a parameter, the seller automatically joins the coordinator and gains the ability to interact with the escrow (e.g., to confirm receipt or request release of funds). This enables trustless transactions where participants are introduced to each other through the coordinator.

- **Multi-party Agreements:** A coordinator managing multi-party agreements allows a creator to establish an agreement by passing multiple participants as parameters. Each participant specified in the parameter list joins the coordinator and receives appropriate roles, enabling complex coordination scenarios where relationships are established dynamically.

The following transition illustrates how *PP* works in combination with role assignments:

$$q_1 \xrightarrow{\triangleright \mathsf{p}:\ \mathsf{op}(\mathsf{p}_1,\mathsf{p}_2),\{(\mathsf{p},O)\mapsto\blacksquare,(\mathsf{p}_1,B)\mapsto\blacksquare,(\mathsf{p}_2,B)\mapsto\blacksquare,\}} q_2$$

In this transition, participant p (the caller) calls operation op with parameters p_1 and p_2. The participants $\sigma(\mathsf{p}_1)$ and $\sigma(\mathsf{p}_2)$ join the coordinator by passing parameters to the operation (*PP*), while $\sigma(\mathsf{p})$ joins by invocation (*BI*). After execution, role O is assigned to the caller, and role B is assigned to both parameter participants, establishing their relationships within the coordinator.

The *PP* feature is natively supported in contracts such as AssetTransfer, BasicProvenance, and ERC20 Token, where participants can be introduced to coordinators through operation parameters.

Role Revocation (*RR*)

Role revocation enables coordinators to remove roles from participants dynamically, allowing for scenarios where privileges are revoked based on system events or user actions. This feature is essential for modelling authorization systems where access rights can change over time, such as ownership transfers, access revocation, or privilege degradation.

In EDAM, role revocation follows directly from role assignment mechanisms. When a transition includes a role assignment $\rho'(\mathsf{p}, R) = \square$ in its post-condition role updates, the role update function rupd removes role R from participant $\mathfrak{p}$'s role set, where $\mathfrak{p} = \sigma(\mathsf{p})$. The role update is applied after the transition executes successfully, ensuring that role changes occur atomically with the state transition.

Real-world scenarios:

- **Ownership Transfer:** In an asset coordinator (AssetTransfer), when an owner transfers ownership to another participant, the previous owner loses the owner role while the new owner gains it. This is modeled by a role assignment that revokes the owner role from the caller ($(\mathsf{p}, Owner) \mapsto \square$) and grants it to a parameter participant. This pattern enables secure asset transfers where access rights are properly updated.
- **Access Revocation:** In a basic coordinator (BasicProvenance), participants can lose access rights when certain conditions are met. For example, if a participant violates terms of service or fails to meet obligations, their roles can be revoked through a transition that explicitly removes their privileges. This enables coordinators to enforce policies and maintain system integrity.
- **Delegation Expiration:** In a transport coordinator (RefrigeratedTransport), temporary delegations can be revoked when they expire or when conditions change. A participant who had been granted temporary access can have those roles revoked through a transition that updates role assignments, enabling time-limited or conditional access patterns.
- **Multi-party Coordination:** In a simple multi-party coordinator (Simple Marketplace), participants can lose roles when agreements are terminated or when they are removed from a group. Role revocation ensures that participants who are no longer part of a coordination scenario lose their associated privileges, maintaining proper access control.

The DigitalLocker contract exemplifies native support for *RR*, where role revocation is expressed naturally through role assignments with the $\square$ modality. When a transition includes $\rho'(\mathsf{p}, R) = \square$, the role update function rupd immediately removes role R from $\pi(\mathfrak{p})$, where $\mathfrak{p} = \sigma(\mathsf{p})$. This mechanism is also present in contracts such as AssetTransfer, BasicProvenance, RefrigeratedTransport, and Simple Marketplace, demonstrating the widespread need for role revocation in realistic coordination scenarios.

Multi-Participant Roles (*MPR*)

Multi-participant roles enable multiple participants to share the same role within a coordinator, allowing for scenarios where groups of participants have equivalent privileges or responsibilities. This feature is essential for modelling coordination patterns where roles represent capabilities that can be held by multiple participants simultaneously, rather than being exclusive to a single participant.

In EDAM, multi-participant roles arise naturally from role assignment semantics. When multiple participant variables in a role assignment map to the same participant identity, or when different transitions assign the same role to different participants, those participants collectively share that role. Formally, if $\rho' = \{(\mathsf{p}_1, O) \mapsto \blacksquare, (\mathsf{p}_2, B) \mapsto \blacksquare\}$ and $\sigma = \{\mathsf{p}_1 \mapsto \mathfrak{p}, \mathsf{p}_2 \mapsto \mathfrak{p}\}$, then participant $\mathfrak{p}$ receives $\pi(\mathfrak{p}) = \pi(\mathfrak{p}) \cup \{O, B\}$, accumulating roles from multiple assignments. More importantly, when different participant variables map to different participant identities but are assigned the same role, multiple participants share that role, enabling group-based access control.

Real-world scenarios:

- **Multi-signature Wallets:** In a wallet coordinator, multiple participants can share the owner role, requiring multiple signatures to authorize transactions. When several participants are assigned the owner role, they collectively control the wallet, and operations may require approval from a threshold of owners. This pattern enables secure fund management where no single participant has unilateral control.
- **Transport Coordination:** In a transport coordinator (RefrigeratedTransport), multiple participants can share roles such as driver or passenger. When a transport service is established, multiple drivers can be assigned the driver role, allowing any of them to fulfil the transport requests. Similarly, multiple passengers can share the passenger role, enabling group bookings and shared transportation services.
- **Digital Lockers:** In a digital locker coordinator (DigitalLocker), multiple participants can share access roles, allowing several users to access the same locker. When a locker is created, multiple participants can be assigned access roles, enabling scenarios where family members, team members, or business partners share access to digital assets or resources.
- **Token Coordination:** In token-based coordinators, multiple participants can share roles such as minter or administrator. When several participants are assigned administrative roles, they collectively manage the token coordinator, enabling decentralised governance where multiple parties have equivalent privileges for critical operations.

The DigitalLocker and RefrigeratedTransport contracts demonstrate native support for *MPR*, where multiple participants can be assigned the same roles through role assignments. This enables natural modelling of group-based coordination scenarios where roles represent capabilities that are not exclusive to individual participants.

Feature Coverage Summary

As demonstrated in Table V.1, EDAM provides native support for all five essential features across the benchmark contracts. The native support for *ICI*, *BI*, *PP*, *RR*, and *MPR* in EDAM enables more natural modelling of coordination scenarios, as demonstrated by the Case Study.

It is worth noting that while inter-coordinator interactions (*ICI*) are natively supported in EDAM through call tries, certain patterns present in some contracts cannot be modelled faithfully due to semantic constraints. The Ping Pong contract exemplifies a reentrancy pattern which, although inter-coordinator calls are expressible in EDAM, reentrancy is explicitly prevented during runtime execution: the system's trace engine enforces a non-reentrant execution model (using read-mode only) as described in Chapter III. The Bazaar contract relies on dynamic deployment of coordinators at runtime—a feature that is not currently supported by EDAM, which assumes a statically defined network topology. These limitations represent design choices that prioritise safety and predictability over expressiveness for potentially unsafe patterns.

V.2 Quality of code and tests

To validate the correctness of our generated smart contracts, we execute the concrete test suites generated through the process described in Section IV.3.6. The validation methodology employs a multi-faceted approach combining code coverage [58] analysis, mutation testing, and cross-validation against established implementations.

The validation process involves running these test suites against the generated Solidity contracts in the Hardhat [86] testing framework, which instruments the contracts to collect execution data and reports comprehensive coverage metrics.[3] Coverage analysis provides quantitative measures of how thoroughly the test suites exercise the generated contract code, identifying both well-tested regions and potential gaps in test coverage.

Coverage Analysis: The coverage results demonstrate that our generated test suites achieve high coverage across most contracts. For almost all contracts, except AMM, the test suite covers all operations, achieving 100% operation coverage. This high operation coverage indicates that the test generation process successfully exercises all functionality defined in the EDAM specifications.

Some contracts (e.g., HelloBlockchain, DefectiveCounter, FrequentFlyer, ThermostatOperation, AMM, ERC20 Token, SimpleWallet and Case Study) exhibit lower statement, branch, and line of code coverage due to the absence of role revocation mechanisms. In these cases, certain code paths within the role satisfaction function roleSatisf remain unexecuted, specifically those handling role revocation consistency checks.[4] Despite the presence of role revocation in DigitalLocker, we observe lower coverage compared to other models with role revocation for participant parameters (AssetTransfer and BasicProvenance). This discrepancy stems from limitations in random test generation: participants satisfying the required role constraints are not consistently generated, and boundary values necessary to trigger revert branches within conditional statements are not always produced. These factors contribute to incomplete exploration of the contract's state space.

More complex contracts such as AMM, ERC20 Token and Case Study present greater challenges for test generation, achieving 74.49% and 80% statement coverage, respectively. The lower coverage in these cases occurs because certain statements require specific preconditions that depend on previous operations being executed in a particular sequence. For instance, certain revert branches (such as in approval functions) remain uncovered because the test suite primarily generates valid transitions, where participants can approve any amount without encountering constraint violations. In the case of AMM, intricate sequences of operations (i.e., token approval, liquidity provision, swap execution) are required for a swap to be executed, creating dependencies that are difficult to exercise through random test generation alone. Bazaar and Ping Pong, could not be tested due to the limitations mentioned in Section V.1 marked with – in the table.

Mutation Testing Analysis: Beyond coverage metrics, we employ mutation testing using the ReSuMo framework [28] to assess the quality and effectiveness of our generated test suites. Mutation testing systematically introduces small syntactic changes (mutants) to the generated code (all mutation operators can be found in [26]) and verifies whether the test suite can detect and kill these mutants. The mutation score, expressed as the percentage of mutants killed by the test suite, serves as a quality indicator: a high mutation score demonstrates that the test suite is effective at catching potential bugs and distinguishing between correct and incorrect implementations. The last column in Table V.1 reports the mutation scores for each contract.

Mutation scores (see Table V.1) range from 62.70% (AMM) to 86.44% (SimpleWallet), with most contracts achieving above 70%. This range demonstrates that our test generation approach produces test suites capable of detecting a substantial proportion of potential faults. Coverage directly impacts mutation scores: higher coverage (e.g., SimpleWallet at 86.44%, AssetTransfer at 81.4%) correlates with better mutation detection, as more code paths are exercised and thus more mutations can be detected.

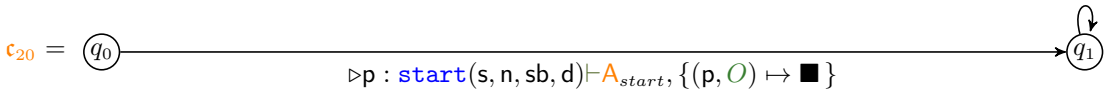
[3]The Hardhat framework instruments Solidity smart contracts to collect execution data during testing and reports statement, branch, function and line of code coverage.

[4]In future work we plan to optimise the approach to improve coverage for contracts without role revocation.

Several factors explain the presence of surviving mutants across different contracts. (a) contracts without role revocation (e.g., HelloBlockchain, DefectiveCounter, FrequentFlyer) have mutations in the roleSatisf function's revocation-consistency checks that survive because those code paths are never executed; (b) some mutations are irrelevant to most of our models: changing field visibility seldomly matters, and reordering enum values (states and roles) has no semantic effect; (c) boundary values needed to trigger certain revert branches are not always generated, leaving some conditional branches mutated not to be tested (e.g., mutate >= to > in guards); (d) AMM and Case Study, which require intricate sequences of operations (i.e., token approval, liquidity provision, swap execution), present greater challenges for random test generation and consequently achieved lower mutation scores. HelloBlockchain has the mutation score (66.66%) due to its simplicity: it has few expressions, and several are within roleSatisf functions that remain uncovered without role revocation.

Case Study Validation: We apply our framework to a case study elaborating on the Simple Marketplace smart contract in [147]. The idea is to extend Simple Marketplace to let it interact with an ERC20-like contract for explicit token-based payments[5]. Following the specifications in [201], the EDAM's of $\mathfrak{c}_{20}$ below[6] captures the behaviour of the ERC20 coordinator necessary for the call tries used in our marketplace contract:

$$\{(\mathsf{p}, O) \mapsto \blacksquare\}, \mathsf{a} \geq 0 \triangleright \mathsf{p}\colon \texttt{mint}(\mathsf{r}, \mathsf{a}) \vdash \{\mathsf{totalSupply}{+}{=}\mathsf{a}, \mathsf{balanceOf}[\mathsf{r}]{:=}\mathsf{a}\}$$
$$\mathsf{balanceOf}[\mathsf{p}] \geq \mathsf{a} \geq 0 \triangleright \mathsf{p}\colon \texttt{transfer}(\mathsf{r}, \mathsf{a}) \vdash \{\mathsf{balanceOf}[\mathsf{p}]{-}{=}\mathsf{a}, \mathsf{balanceOf}[\mathsf{r}]{+}{=}\mathsf{a}\}$$
$$\triangleright \mathsf{p}\colon \texttt{approve}(\mathsf{s}, \mathsf{a}) \vdash \{\mathsf{allowance}[\mathsf{p}][\mathsf{s}]{:=}\mathsf{a}\}$$
$$\mathsf{balanceOf}[\mathsf{p}] \geq \mathsf{a} \triangleright \mathsf{p}\colon \texttt{burn}(\mathsf{a}) \vdash \{\mathsf{totalSupply}{:=}\mathsf{totalSupply} - \mathsf{a}, \mathsf{balanceOf}[\mathsf{p}]{+}{=}\mathsf{a}\}$$
$$\mathsf{allowance}[\mathsf{s}][\mathsf{p}] \geq \mathsf{a} \wedge \mathsf{balanceOf}[\mathsf{s}] \geq \mathsf{a} \triangleright \mathsf{p}\colon \texttt{transferFrom}(\mathsf{s}, \mathsf{r}, \mathsf{a}) \vdash \{\mathsf{A}_{transferFrom}\}$$



where $\mathsf{A}_{start} = \{\mathsf{name}{:=}\mathsf{n}, \mathsf{symbol}{:=}\mathsf{sb}, \mathsf{decimals}{:=}\mathsf{d}, \mathsf{totalSupply}{:=}\mathsf{s}, \mathsf{balanceOf}[\mathsf{p}]{:=}\mathsf{s}\}$ and $\mathsf{A}_{transferFrom} = \{\mathsf{balanceOf}[\mathsf{s}]{-}{=}\mathsf{a}, \mathsf{balanceOf}[\mathsf{r}]{+}{=}\mathsf{a}, \mathsf{allowance}[\mathsf{s}][\mathsf{p}]{-}{=}\mathsf{a}\}$.

The coordinator $\mathfrak{c}_{20}$ maintains the total amount of tokens in the field totalSupply and a dictionary balanceOf to apportion the tokens to participants; the creator p of $\mathfrak{c}_{20}$ set the name, symbol, and decimals of the token and is initially assigned all the initial supply tokens and the owner role O. Moreover, the dictionary allowance is used to let a participant allow other participants to spend their tokens.

The labels $l_{\texttt{start}}$ and $l_{\texttt{makeO}}$ will be used in the EDAM $\mathfrak{c}_{\mathrm{m}}$ for our marketplace coordinator.

$$l_{\texttt{start}} = \triangleright \mathsf{p}\colon \texttt{start}(\mathsf{d}, \mathsf{b}) \vdash \{\mathsf{des}{:=}\mathsf{d}, \mathsf{pr}{:=}\mathsf{b}\}, \{(\mathsf{p}, O) \mapsto \blacksquare\}$$
$$l_{\texttt{makeO}} = \mathsf{a} > \mathsf{offer}, [\mathfrak{c}_{20}.\texttt{transferFrom}(\mathsf{p}, \mathfrak{sf}, \mathsf{a})] \triangleright \mathsf{p}\colon \texttt{makeO}(\mathsf{a}) \vdash \{\mathsf{offer}{:=}\mathsf{a}, \mathsf{u}{:=}\mathsf{p}\}, \{(\mathsf{p}, B) \mapsto \blacksquare\}$$

The former label lets a participant p create the coordinator $\mathfrak{c}_{\mathrm{m}}$ fixing description d and initial price b of the item on sale (respectively assigned to the fields des and pr) as well as the owner role O for p. Label $l_{\texttt{makeO}}$ is used by participants to submit offers invoking the makeO operation with the offered amount a; this operation is enabled on two conditions:

- the new offer must be competitive (the guard a > offer requires a to be greater than the current offer stored in offer) and
- it is possible to transfer tokens using the transferFrom operation of an ERC20 Token coordinator behaving as $\mathfrak{c}_{20}$; note that besides the identity p of the caller and the amount a this call try requires the identity $\mathfrak{c}_{\mathrm{m}}$ of the marketplace coordinator ($\mathfrak{sf}$ will return $\mathfrak{c}_{\mathrm{m}}$).

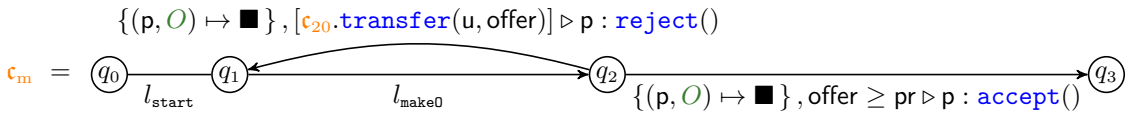
A successful invocation of makeO results in the caller p being assigned the role B, field offer being updated with the new offer, and field u storing the identity of the new buyer. In passing we note

[5] As discussed in Section V.1, the Simple Marketplace contract in [147] use a rudimentary mechanism for payments operating on a field of the contract.

[6] For readability, x+=e shortens x:=e and likewise for x-=e.

that this coordinator allows consecutive offers from a same participants; if we wanted to forbid this, we could simply replace the guard of $l_{\texttt{make0}}$ with $\mathsf{a} > \mathsf{offer} \wedge \mathsf{p} \neq \mathsf{u}$.

Let us now introduce the EDAM for our case study:



Once an offer is made, the coordinator is in state q_2; here the owner participant p can invoke either the operation `accept` or the operation `reject`. In the former case, the coordinator reaches state q_3, a final state indicating a successful sell. Instead, the rejection of an offer triggers the refund to the last buyer u via a call try `transfer` of $\mathfrak{c}_{20}$; in this case the coordinator goes back to state q_1 to enable new offers. Notice that the execution of `reject` triggers the invocation of the `transfer` operation of $\mathfrak{c}_{20}$ (rather than of `transferFrom`) because the tokens' owner s responsible for the refund at this stage is the marketplace coordinator $\mathfrak{c}_{\mathrm{m}}$.

We validated the generated code for the Case Study by executing the automatically generated test suites and applying mutation testing. Test generation and execution took 9.6 and 6 minutes respectively, demonstrating reasonable performance for a complex multi-contract system. Coverage metrics show high effectiveness: 80.95% statement coverage, 69.44% branch coverage (higher than AMM's 62.86% due to fewer interactions with the ERC20 Token coordinator), 100% operation coverage, and 90.24% line of code coverage. These results indicate that the test generation process successfully exercises the majority of the contract's functionality. The mutation score of 74.66% validates that our test generation and mutation testing approach successfully handles the complexity introduced by combining all EDAM features, including inter-coordinator interactions, role management, and complex state transitions.

We now highlight our test generation process with the model from the Case Study. The following images shows automatically generated symbolic (top) and concrete (bottom) traces, where the latter is generated from the former.

```
=== Symbolic Trace #1 ===
Step #1_C20: p -- _ start(s, b, n, d) -> q1
Step #2_Cm: p -- _ start(d, b) -> q1
Step #3_C20: p -- q1 burn(a) -> q1
Step #4_Cm: p -- q1 make0(a) -> q2
Step #5_C20: p -- q1 approve(s, a) -> q1
Step #6_Cm: p -- q2 reject() -> q1
Step #7_Cm: p -- q1 make0(a) -> q2
=== End of Trace ===
```

```
it("Trace #1", async function () {
  const participant_0 = accounts[0];
  const contract_factory_C20 = await ethers.getContractFactory("C20", participant_0);
  const instance_C20 = await contract_factory_C20.deploy(32, "aueiyclyc", "lovkzevus", 8);
  const C20 = instance_C20.target;
  const contract_factory_Cm = await ethers.getContractFactory("Cm", participant_0);
  const instance_Cm = await contract_factory_Cm.deploy("bfkjidsq", 93, instance_C20.target);
  const Cm = instance_Cm.target;
  await expect(instance_C20.connect(participant_0).burn(22)).to.not.be.reverted;
  await expect(instance_Cm.connect(participant_0).make0(69)).to.be.reverted;
  //...
  await expect(instance_Cm.connect(participant_0).make0(1)).to.be.reverted;
  await expect(instance_C20.connect(participant_0).approve(Cm, 68)).to.not.be.reverted;
  await expect(instance_Cm.connect(participant_0).reject()).to.be.reverted;
  //...
  await expect(instance_Cm.connect(participant_0).reject()).to.be.reverted;
  await expect(instance_Cm.connect(participant_0).make0(38)).to.be.reverted;
  //...
  await expect(instance_Cm.connect(participant_0).make0(22)).to.be.reverted;
  await expect(instance_Cm.connect(participant_0).make0(1)).to.not.be.reverted;
```

The ellipses represent repeated failures of the same operation; we omit it for brevity. Expected to succeed operations are calls ending with *to.not.be.reverted* and expected to fail operations are calls ending with *to.be.reverted*. The concrete trace instantiates two participants, deploys both coordinators by the same participant `participant_0` with randomly generated values (this is the concrete test from symbolic *Step #1* and *Step #2*), burns 22 tokens from `participant_0` which succeeds (*Step #3*), makes several failures offers (*Step #4*), approves $\mathfrak{c}_{20}$ tokens to $\mathfrak{c}_{\mathrm{m}}$(*Step #5*),

makes several failed reject offers (*Step #6*) and try several makeO with the last one succeeding (*Step #7*).

Cross-Validation with Established Implementations:

To further validate our approach, we conducted an experimental cross-validation by testing whether our generated code could pass existing ERC20 Token (e.g., EDAM for $\mathfrak{c}_{20}$) contract test suites, specifically the OpenZeppelin [163] implementation [163] mintable and burnable ERC20 contract, which is widely used in industry. This cross-validation serves two purposes: first, it verifies that our generated contracts conform to established standards and expected behaviour; second, it helps identify potential specification issues in our formal models by comparing against well-tested, production-ready implementations.

We removed from the OpenZeppelin test suite the tests that are not implemented in our generated code (namely events emission and custom errors, currently not supported in EDAM). Initially, we discovered that our test suite was not passing on the existing OpenZeppelin code, and conversely, their tests were not passing on our generated code. This discrepancy prompted us to investigate the root cause. Through careful analysis, we realised that our implementation forbade zero-amount transfers while the ERC20 standard allows them. This was a significant discovery, as we noticed that the guard conditions for `transfer` and `transferFrom` operations should permit amounts greater than or equal to zero (≥ 0) rather than strictly greater than zero ($\mathsf{balanceOf}[\mathsf{p}] \leq \mathsf{a}$). By making this correction to our model, the newly generated code and test suite successfully passed both our tests and the OpenZeppelin implementation, which is widely used in the industry. This demonstrates how cross-validation with established implementations can reveal subtle specification issues in formal models, and how model-based code generation enables rapid iteration and refinement when such issues are discovered.

V.2.1 Validation Against Real-World Deployments

To further validate our generated ERC20 Token test suite, we conducted an evaluation against real-world smart contracts deployed on the Ethereum blockchain. This section describes the EDAM specification for ERC20 Token, its relationship to the EIP-20 standard, the contract extraction and selection methodology, and the results. All data—extracted contract sources, the test suite, and execution results—are available in the supplementary `currator` repository.[7]

The EIP-20 Standard and the EDAM Specification

The ERC20 Token token standard is formally specified in EIP-20 [201], available at `https://eips.ethereum.org/EIPS/eip-20`. EIP-20 defines a standard interface for fungible tokens on Ethereum, requiring implementations to provide: read-only functions name(), symbol(), decimals(), totalSupply(), balanceOf(address), and allowance(address,address); and state-changing functions transfer(address,uint256), approve(address,uint256), and transferFrom(address,address,uint256). The standard prescribes exact function signatures, parameter types, and ordering; implementations that deviate from these signatures (e.g., additional parameters or different parameter order) do not conform to EIP-20. The standard does not prescribe any particular constructor format; each implementation may define its own.

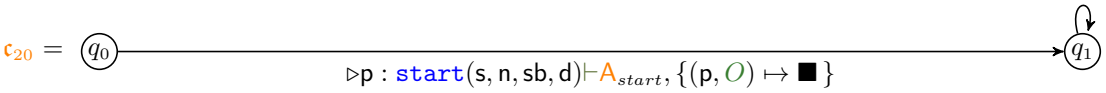
The EDAM for the ERC20 Token coordinator $\mathfrak{c}_{20}$ is shown in Fig. V.1. Where start contract by providing the initial supply, name, symbol, and decimals; and all other operations are available in state q_1.

From the EDAM, we can generate the Solidity code for the ERC20 Token coordinator $\mathfrak{c}_{20}$ as shown in Listing V.1.

```
1  contract C20 {
2    enum State { q1 }
```

[7] The `currator` repository contains: `extract/[passes|(failed/reason)]/<address>/` (verified source code per contract), `c20-test-runner/test/c20_test.js` (the generated test suite), `c20-test-runner/results/` (pass/fail outputs and `results.json`), and `EDAM-ERC20/src/C20_edam_output.py` (the EDAM specification for $\mathfrak{c}_{20}$). `https://github.com/loctet/erc20-contract-curator`.

$$\text{balanceOf}[p] \geq a \geq 0 \triangleright p\colon \texttt{transfer}(r, a) \vdash \{\text{balanceOf}[p]\text{-}{=}a, \text{balanceOf}[r]{+}{=}a\}$$
$$\triangleright p\colon \texttt{approve}(s, a) \vdash \{\text{allowance}[p][s]{:=}a\}$$
$$g_1 \triangleright p\colon \texttt{transferFrom}(s, r, a) \vdash \{\text{balanceOf}[s]\text{-}{=}a, \text{balanceOf}[r]{+}{=}a, \text{allowance}[s][p]\text{-}{=}a\}$$



Where A_{start} = {name:=n, symbol:=sb, decimals:=d, totalSupply:=s, balanceOf[p]:=s} and g_1 = allowance[s][p] ≥ a ∧ balanceOf[s] ≥ a

Figure V.1: EDAM for ERC20 standard

```
3     enum Roles { 0 }
4     State public _state;
5     uint  public totalSupply;
6     string  public symbol; string  public name; uint  public decimals;
7     mapping(address => uint)  public balanceOf;
8     mapping(address => mapping(address => uint))  public allowance;
9     mapping(address => mapping(Roles => bool)) public _permissions;
10
11    constructor(uint s, string memory b, string memory n, uint d)  {
12      if (true) {
13        totalSupply =  s; symbol =  b; name =  n; decimals =  d;
14        balanceOf[msg.sender] = s; _permissions[msg.sender][Roles.0] = true;
15        _state = State.q1;
16        assert(roleSatisf(msg.sender, _roles(Roles.0), new Roles [] (0)));
17      } else {
18        revert("Condition not met");
19      }
20    }
21    function transfer(address r, uint a) public  {
22      if ((_state == State.q1 && balanceOf[msg.sender] >= a)) {
23      balanceOf[r] = (balanceOf[r] + a);
24      balanceOf[msg.sender] = (balanceOf[msg.sender] - a);
25      _state = State.q1; } else { revert("Condition not met"); }
26    }
27
28    function approve(address s, uint a) public  {
29      if ((_state == State.q1 && true)) {
30        allowance[msg.sender][s] = a; _state = State.q1; }
31      else { revert("Condition not met"); }
32      }
33
34    function transferFrom(address s, address r, uint a) public  {
35      if ((_state == State.q1 && (allowance[s][msg.sender] >= a && balanceOf[s] >=
            a))) {
36        balanceOf[r] = (balanceOf[r] + a); balanceOf[s] = (balanceOf[s] - a);
37        allowance[s][msg.sender] = (allowance[s][msg.sender] - a);
38        _state = State.q1;
39      } else { revert("Condition not met"); }
40    }
41
42    function roleSatisf(address participant,
43        Roles[] memory hasrole_roles, Roles[] memory notrole_roles
44    ) internal view returns (bool) {
45      for (uint i = 0; i < notrole_roles.length; i++) {
46        if (_permissions[participant][notrole_roles[i]]) {
47          return false; } }
48      for (uint i = 0; i < hasrole_roles.length; i++) {
49        if (!_permissions[participant][hasrole_roles[i]]) {
50          return false; } }
51      return true;
52    }
53
54    function _roles(Roles r1) internal pure returns (Roles[] memory arr) {
55      arr = new Roles[](1) ;
56      arr[0] = r1;
57    }
```

```
58 }
```

Listing V.1: Solidity code for our ERC20 Token standard

Our EDAM specification for the ERC20 Token coordinator $\mathfrak{c}_{20}$ (see Fig. V.1 and Listing V.1) was derived manually from EIP-20. The EDAM captures the core state (totalSupply, balanceOf, allowance) and transitions for transfer, approve, and transferFrom.

A perfect match between the EDAM and EIP-20 is impossible by design. EDAMs do not model return values; EIP-20 requires transfer, approve, and transferFrom to return bool. EDAMs do not model events; EIP-20 mandates Transfer and Approval events. Nevertheless, the EIP-20 read functions (name(), symbol(), decimals(), totalSupply(), balanceOf(address), allowance(address,address)) are present in the generated code: Solidity automatically generates getter functions for public state variables, and our fields use the same names as the EIP-20 read interface. Our specification therefore captures the *behavioural semantics* of the core operations (state updates, guards) rather than the full EIP-20 interface. The generated tests accordingly validate that state changes occur as specified when operations succeed or revert, but do not assert on return values or events.

Contract Extraction, Selection, and Test Execution

We developed a Python script (`currator/scan_historical.py`) to scan the Ethereum blockchain for newly deployed contracts. The script iterates through a specified block range (for this experiment, from block 23701204 to 23729476), examining each transaction to identify contract-creation events (transactions with a null recipient and bytecode in the transaction data). For each newly created contract address, the script calls the standard EIP-20 read methods (name(), symbol(), decimals(), totalSupply(), balanceOf(address)) using a minimal EIP-20 ABI. If these calls succeed and return values of the expected types, the contract is classified as an ERC20 Token-like implementation.

This heuristic identifies contracts exposing an EIP-20-compatible read interface; it does not guarantee full EIP-20 compliance, since write methods (transfer, approve, transferFrom) may use different signatures. Applying this process yielded 1771 candidate addresses. For contracts with verified source code on Etherscan,[8] the script downloaded sources via the Etherscan API into `currator/extract/[passes|(failed/reason)]/<address>/`. In total, 377 contracts had verified source code and were retained; the remaining 1394 unverified contracts were excluded from compilation and testing.

A significant challenge in this validation was the heterogeneity of constructor signatures across ERC-20 implementations. Our EDAM ERC-20 model (Line 11) expects deployment with four parameters (*initial supply*, *name*, *symbol*, *decimals*), but contracts vary substantially: some use only a subset of these parameters, others introduce additional parameters, and ordering differs between implementations. Instead of adapting the model and regenerating tests per contract, we normalised the contracts by copying each token implementation into C20.sol, renaming the main contract to C20, and adapting its constructor to the four-parameter interface required by our framework. For OpenZeppelin-based implementations, we preserved the underlying logic and modified only the constructor invocation. In our framework, token supply and transferred values are handled directly in the intended base units; therefore, test inputs are not additionally scaled by the decimals factor.

The final set consisted of 377 contracts exposing standard EIP-20 write-method signatures (transfer(address,uint256), approve(address,uint256), transferFrom(address,address,uint256)). We executed the generated test suite (`c20-test-runner/test/c20_test.js`) on each contract with Hardhat by copying source files from each `currator/extract/[passes|(failed/reason)]/<address>/` folder into the test project and running npx hardhat test. Results were recorded in `results.json` and per-contract output files.

[8] Etherscan (https://etherscan.io) is a blockchain explorer for Ethereum that provides verified source code when developers submit and match bytecode.

Table V.2: Outcomes of applying the generated EIP-20 test suite to the 377 adapted contracts (all contracts with verified source). Percentages are relative to this adapted set.

Outcome	# contracts	Share of adapted
Passed all tests	324	86.0%
Failed: non-standard zero-amount transfer	18	4.7%
Failed: non-conservative transfer logic	25	6.6%
Failed: adapter did not compile	8	2.0%
Failed: not deployable due to size limit for contract	1	0.3%
Failed: owner not allowed to transfer	1	0.3%
Total evaluated	377	100%

Results

We ran the generated test suite on 377 contracts. The suite exercises transfer, approve, and transferFrom under both expected-success and expected-revert scenarios, including boundary cases such as zero-amount transfers and exhausted allowances. As summarised in Table V.2, 324 contracts (86.0%) passed all tests. This provides evidence that the EDAM-based generator produces tests that correctly characterise the intended EIP-20 behaviour implemented by the majority of real-world tokens in our sample.

The remaining 53 contracts systematically violated at least one of our oracles. Eighteen contracts reject zero-amount transfers, thereby deviating from the widely adopted interpretation of EIP-20 that treats a transfer of amount 0 as a no-op; these adapters are grouped under `currator/extract/failed/`*not compliant with ERC20 standard, cannot transfer 0 amount*. Twenty-five contracts implement custom transfer logic in which the debited and credited amounts differ (e.g., fee-on-transfer, deflationary, or rebasing tokens), so that our balance-preservation and injectivity properties—which are tailored to the standard ERC-20 transfer semantics—are intentionally violated; these contracts are collected in the `currator/extract/failed/`*custom transfer logic, different amount transferred* subdirectory. A single contract exceeded the maximal deployable bytecode size for our test environment and therefore could not be deployed, and one contract implements owner-only transfer restrictions that invalidate our assumption that any holder may transfer their balance. The remaining eight contracts could not be compiled in our Hardhat configuration because of unresolved external dependencies or incompatible compiler settings; they are stored under `currator/extract/failed/`*not compiling* and are excluded from the behavioural compliance counts, but still documented for completeness.

V.3 Vulnerability Mitigation through Model-Driven Design

This section demonstrates how the EDAM framework addresses common smart contract vulnerabilities through its formal model structure and code generation process. Rather than detecting vulnerabilities post-hoc through static analysis or runtime monitoring, EDAM prevents entire classes of vulnerabilities by construction through its formal semantics and model-driven code generation approach. We focus here on vulnerabilities that are prevented by the structural properties of the EDAM formal semantics. These vulnerabilities are eliminated not through runtime checks or static analysis, but through the semantics of the model itself, which is preserved through the code generation process described in Chapter IV.

The key insight underlying this approach is that many smart contract vulnerabilities stem from the gap between the intended behaviour of a contract and its actual implementation. By specifying contracts at the semantic level through formal models, and generating code that faithfully implements these models, EDAM ensures that security properties established at the model level are preserved in the executable code. This correct-by-construction methodology is more effective than post-hoc vulnerability detection [152], as it prevents vulnerabilities from being introduced in the first place.

V.3.1 Reentrancy Attacks

Reentrancy attacks are among the most devastating smart contract vulnerabilities, as demonstrated by the infamous DAO attack in 2016 that resulted in the loss of $60 million [20]. A reentrancy attack occurs when a contract makes an external call before updating its state, allowing the called contract to recursively call back into the original contract and exploit inconsistent state.

The EDAM framework prevents reentrancy attacks through its formal semantics, specifically through the read-only flag mechanism defined in the network execution rules (see Section III.2.3). According to Definition III.12, a network configuration includes a read-only flag $\omega_i \in \{\mathsf{true}, \mathsf{false}\}$ for each coordinator. The semantic rule [TR] requires that when a coordinator executes a transition containing call tries, the coordinator must be placed in read-only mode ($\omega = \mathsf{true}$) before any external calls are made.

The formal semantics ensure that coordinators in read-only mode cannot be invoked, but their fields may be accessed. This property is fundamental to the network execution model: once a coordinator enters read-only mode to execute its call tries, it cannot accept new operation invocations until the transition completes and read-only mode is cleared. This prevents reentrant calls by construction, as the definition of the network configuration explicitly forbids invocations of coordinators in read-only mode.

The execution semantics work as follows. When a transition with call tries is executed, the rule [TR] requires that the coordinator be added to the network configuration in read-only mode before the call sequence begins. The premise of the rule states:

$$\mathfrak{c} : \langle q, \pi, \sigma', \mathsf{true} \rangle \mid N \xhookrightarrow[\pi,\sigma']{\mathfrak{c}:[\![\mathsf{L}]\!]_{N,\pi,\sigma'}} \mathfrak{c} : \langle q, \pi, \sigma', \mathsf{true} \rangle \mid N'$$

This ensures that during the execution of call tries L, the coordinator $\mathfrak{c}$ is in read-only mode. Since coordinators in read-only mode cannot be invoked (as per Definition III.12), any attempt to re-enter the coordinator during the execution of its call tries will fail at the semantic level, before any code is generated or executed.

The code generation process (see Chapter IV) translates this semantic protection into a `nonReentrant` modifier that wraps functions containing external calls, ensuring that the generated Solidity code enforces the same protection at the implementation level. This dual-layer protection—at both the semantic and implementation levels—guarantees that reentrancy attacks are impossible by construction, as the protection is inherent to the formal model rather than an optional feature that could be forgotten or misconfigured.

Vulnerable Code Pattern

The following code snippet illustrates the vulnerable pattern that led to the DAO attack [20], where an external call is made before updating the contract's state:

```
contract VulnerableWallet {
    mapping(address => uint) public balances;

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        // Vulnerable: external call before state update
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
        balances[msg.sender] -= amount;  // State updated AFTER call
    }
}
```

EDAM Model

In the EDAM framework, the withdrawal operation is modeled as a transition that performs the external call, with the semantic rule ensuring read-only mode protection. The transition is defined as:

$$\mathsf{balance[p]} \geq \mathsf{amount}, [\mathfrak{c}_{20}.\mathtt{transfer}(\mathsf{p}, \mathsf{amount})] \triangleright \mathsf{p} : \mathtt{withdraw}(\mathsf{amount}) \vdash \{\mathsf{balance[p]}\text{-=}\mathsf{amount}\}$$

The semantic rule [TR] ensures that when this transition executes, the coordinator is placed in read-only mode (ω = true) before the external call $\mathfrak{c}_{20}.\mathtt{transfer}(\mathsf{p}, \mathsf{amount})$ is made. The assignment balance[p]-=amount is evaluated after the call tries complete, but the read-only flag prevents reentrant calls during execution.

Generated Code

The code generation process produces the following Solidity code for the version where instead of using the currency of the blockchain, we use a ERC20 Token contract, which enforces reentrancy protection through the `nonReentrant` modifier:

```
contract SafeWallet {
    mapping(address => uint) public balances;
    bool private _entered;

    modifier nonReentrant() {
        require(!_entered, "reentrant call");
        _entered = true;
        _;
        _entered = false;
    }

    function withdraw(uint amount) public nonReentrant {
        if (_state == State.q1 && balances[msg.sender] >= amount) {
            try _C20.transfer(msg.sender, amount) {
                balances[msg.sender] -= amount;
                _state = State.q1;
            } catch {
                revert("Expected external call to succeed");
            }
        } else {
            revert("Condition not met");
        }
    }
}
```

The `nonReentrant` modifier ensures that the function cannot be re-entered while it is executing, and the state update occurs before the external call, matching the semantic guarantees of the EDAM formal model.

V.3.2 Access Control Vulnerabilities

Improper access control is a leading cause of smart contract vulnerabilities, often resulting from missing or incorrect authorization checks. The Parity wallet hack in 2017, which resulted in the loss of $30 million, was caused by an access control vulnerability [104]. These vulnerabilities typically arise when operations that should be restricted to specific participants or roles are made accessible to unauthorized parties.

The EDAM framework addresses this through its explicit role-based access control model, where role constraints are mandatory components of every transition definition. According to Definition III.6, each transition in a coordinator must specify pre-transition role constraints ρ and post-transition role updates ρ'. The role constraint ρ is a partial function from participant variables and roles to role modalities (see Definition III.4), specifying which roles are required (■) or forbidden (□) for each participant variable involved in the transition.

The semantic rule [TR] enforces these role constraints through the satisfaction relation $\langle \pi, \sigma' \rangle \models \rho$ (see Definition III.7), which must hold before the transition can execute. This relation requires that for all participant variables $(\mathsf{p}, \_) \in \mathrm{dom}\,(\rho)$:

$$\pi(\sigma(\mathsf{p})) \cap \rho_{\square}(\mathsf{p}) = \emptyset \quad \text{and} \quad \rho_{\blacksquare}(\mathsf{p}) \subseteq \pi(\sigma(\mathsf{p}))$$

This ensures that participants have all required roles and none of the forbidden roles before the transition can execute. Since role constraints are mandatory components of every transition definition, it is impossible to define a transition without specifying its access control requirements. This eliminates the possibility of forgetting access control checks, as they are fundamental to the transition structure rather than optional annotations.

The code generation process translates these role constraints into explicit authorization checks in the generated Solidity code, ensuring that the security policy specified at the model level is enforced at the implementation level. The `role_satisf` function in the interpreter (see Chapter IV) verifies role constraints before allowing transitions to execute, and the generated code includes corresponding checks that ensure only authorized participants can invoke operations.

Vulnerable Code Pattern

The following code snippet illustrates a vulnerable token contract where the mint operation lacks access control checks:

```
contract VulnerableToken {
    mapping(address => uint) public balances;

    function mint(address to, uint amount) public {
        // Missing: require(msg.sender == owner);
        // Missing: require(hasRole(msg.sender, MINTER_ROLE));
        balances[to] += amount;  // Anyone can mint tokens
    }
}
```

EDAM Model

In the EDAM framework, the mint operation must explicitly specify role constraints. The transition is defined as:

$$\{(\mathsf{p}, \mathit{Admin}) \mapsto \blacksquare\} \quad \mathsf{true} \triangleright \mathsf{p} : \mathtt{mint}(\mathsf{to}, \mathsf{amount}) \vdash \{\mathsf{balance}[\mathsf{to}] \mathrel{+}= \mathsf{amount}\}$$

The pre-transition role constraint $\rho(\mathsf{p}, \mathit{Admin}) = \blacksquare$ specifies that the caller must have the Admin role. The semantic rule [TR] enforces this constraint through the satisfaction relation $\langle \pi, \sigma' \rangle \models \rho$, which must hold before the transition can execute.

Generated Code

The code generation process produces the following Solidity code, which includes explicit role checks:

```
contract SafeToken {
    mapping(address => uint) public balances;
    mapping(address => mapping(Roles => bool)) public _permissions;

    function mint(address to, uint amount) public {
        if (_state == State.q1 && _permissions[msg.sender][Roles.Admin]) {
            balances[to] += amount;
            _state = State.q1;
        } else {
            revert("Condition not met");
        }
    }
}
```

The role check `require(_permissions[msg.sender][Roles.Admin], "Unauthorized")` is automatically generated from the role constraint $\rho(\mathsf{p}, \mathit{Admin}) = \blacksquare$ in the transition definition, ensuring that access control cannot be forgotten.

V.3.3 External Call Vulnerabilities

Vulnerable external calls pose significant risks when contracts interact with untrusted external contracts. The BatchOverflow vulnerability, which affected multiple ERC20 token contracts, was caused by improper handling of external calls [104]. These vulnerabilities typically arise when external calls are made without proper error handling, reentrancy protection, or state consistency guarantees.

The EDAM framework addresses this through its explicit modelling of inter-contract interactions and the read-only flag mechanism. External calls are modeled as call tries in the transition label (see Definition III.3), which explicitly specify whether the call is expected to succeed ($\mathfrak{c}.\mathtt{op}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$) or fail ($\mathfrak{c}.\overline{\mathtt{op}}(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$). This explicit modelling of expected call outcomes allows the model to handle both success and failure cases, ensuring that the contract's state remains consistent regardless of whether external calls succeed or fail.

The semantic rule [TR] ensures that when call tries are present, the coordinator is placed in read-only mode before the calls execute, preventing reentrancy as discussed in Section V.3.1. Moreover, the rule [SEQ] handles the sequential execution of call tries, ensuring that each call in the sequence is executed in the updated network state resulting from previous calls. This guarantees that external calls are executed in a controlled, predictable manner, with explicit handling of both success and failure cases. The atomicity property of the execution model ensures that all call tries must be fully evaluated before the local state of the coordinator is updated. This guarantees that local consistency (types, guards, roles, and assignments) is preserved, and that external calls cannot leave the contract in an inconsistent state. The combination of the read-only flag mechanism and the explicit modelling of call outcomes ensures that external calls are handled safely by construction, without requiring manual implementation of error handling or reentrancy protection.

The atomicity property of the execution model ensures that all call tries must be fully evaluated before the local state of the coordinator is updated. This guarantees that local consistency (types, guards, roles, and assignments) is preserved, and that external calls cannot leave the contract in an inconsistent state. The combination of the read-only flag mechanism and the explicit modelling of call outcomes ensures that external calls are handled safely by construction, without requiring manual implementation of error handling or reentrancy protection.

The code generation process translates this semantic protection into proper error handling and reentrancy guards in the generated Solidity code. The generated code includes explicit checks for call success or failure, and the read-only flag mechanism is implemented through the `nonReentrant` modifier, ensuring that the semantic guarantees are preserved at the implementation level.

Vulnerable Code Pattern

The following code snippet illustrates a vulnerable escrow contract that makes external calls without proper error handling or reentrancy protection:

```
contract VulnerableEscrow {
    mapping(address => uint) public deposits;

    function withdraw(address token, uint amount) public {
        require(deposits[msg.sender] >= amount, "Insufficient deposit");
        // Vulnerable: no error handling, no reentrancy protection
        IERC20(token).transfer(msg.sender, amount);
        deposits[msg.sender] -= amount;  // State updated AFTER call
    }
}
```

EDAM Model

In the EDAM framework, external calls are explicitly modeled as call tries with expected outcomes. The withdrawal operation is modeled as transitions that handle both success and failure cases explicitly:

$$\text{deposit}[p] \geq \text{amount}, [\mathfrak{c}_{20}.\texttt{transfer}(p, \text{amount})] \triangleright p : \texttt{withdraw}(\text{token}, \text{amount}) \vdash \{\text{deposit}[p] \text{-=} \text{amount}\}$$

$$\text{deposit}[p] \geq \text{amount}, [\mathfrak{c}_{20}.\overline{\texttt{transfer}}(p, \text{amount})] \triangleright p : \texttt{withdraw}(\text{token}, \text{amount})$$

The first transition (upper loop) expects the external call to succeed ($\mathfrak{c}_p$.transfer(amount)) and updates the deposit balance. The second transition (lower loop) expects the call to fail ($\mathfrak{c}_p.\overline{\texttt{transfer}}$(amount)) and leaves the deposit unchanged. The semantic rule [TR] ensures that the coordinator is placed in read-only mode before the call tries execute, and the well-formedness condition ensures that only one transition can be taken based on the call outcome.

Generated Code

The code generation process produces the following Solidity code, which includes proper error handling and reentrancy protection:

```
contract SafeEscrow {
    mapping(address => uint) public deposits;
    bool private _entered;

    modifier nonReentrant() {
        require(!_entered, "reentrant call");
        _entered = true;
        _;
        _entered = false;
    }

    function withdraw(address token, uint amount) public nonReentrant {
        if (_state == State.q1 && deposits[msg.sender] >= amount) {
            try IERC20(token).transfer(msg.sender, amount) {
                deposits[msg.sender] -= amount;  // Success case: update state
                _state = State.q1;
            } catch {
                // Failure case: state unchanged
                _state = State.q1;
            }
        } else {
            revert("Condition not met");
        }
    }
}
```

The generated code ensures that state is updated before the external call, handles both success and failure cases explicitly through `try/catch` blocks, and includes reentrancy protection through the `nonReentrant` modifier, matching the semantic guarantees of the EDAM formal model.

The EDAM framework provides structural advantages that prevent entire classes of vulnerabilities from being introduced in the first place. By reasoning at the semantic level and generating code from verified models, the framework ensures that security properties established at the model level are preserved in the executable code. This correct-by-construction approach is more effective than post-hoc vulnerability detection [152], as it prevents vulnerabilities rather than merely detecting them after they have been introduced.

The key mechanisms through which EDAM prevents vulnerabilities are:

- **Read-only flag mechanism**: The formal semantics require coordinators to enter read-only mode before executing external calls, preventing reentrancy attacks by construction. This protection is enforced both at the semantic level (through the network execution rules) and at the implementation level (through generated reentrancy guards). The mathematical definition of the network configuration explicitly forbids invocations of coordinators in read-only mode, making reentrancy impossible at the semantic level. The atomicity property of the execution model ensures that all call tries must be fully evaluated before the local state of the coordinator

is updated. This guarantees that local consistency (types, guards, roles, and assignments) is preserved, and that external calls cannot leave the contract in an inconsistent state. The combination of the read-only flag mechanism and the explicit modelling of call outcomes ensures that external calls are handled safely by construction, without requiring manual implementation of error handling or reentrancy protection.

- **Mandatory role-based access control**: Every transition must specify role constraints, eliminating the possibility of forgetting access control checks. The satisfaction relation ensures that only authorized participants can invoke operations. Since role constraints are fundamental to the transition structure rather than optional annotations, access control cannot be omitted or forgotten.
- **Explicit external call modelling**: Inter-contract interactions are explicitly modeled through call tries, with explicit success/failure handling. The read-only flag mechanism ensures that external calls cannot leave the contract in an inconsistent state. The atomicity property of the execution model guarantees that all call tries are fully evaluated before local state is updated, ensuring state consistency by construction.

These mechanisms work together to provide a comprehensive security framework that addresses critical smart contract vulnerabilities through design rather than detection. The formal foundation of EDAM ensures that these protections are not optional features that can be forgotten or misconfigured, but rather fundamental properties of the model that are preserved through code generation. The mathematical properties of the formal semantics guarantee that vulnerabilities such as reentrancy, improper access control, and unsafe external calls are impossible by construction, as they are prevented at the semantic level before any code is generated.

Chapter VI: Conclusions & Future works

VI.1 Summary of Contributions

This thesis has presented a comprehensive framework for the formal modelling, verification, and automatic generation of smart contracts using behavioural types. The central contribution is the EDAM (Enhanced Data-Aware Machines) framework, which provides a unified approach to smart contract development.

The EDAM framework addresses fundamental challenges in smart contract development by providing a formal model that balances expressiveness with tractability. The model supports essential smart contract features including state-dependent behaviour, dynamic role-based access control, participant management, and inter-contract interactions through call tries. Unlike existing approaches that address isolated phases of the development lifecycle, EDAM provides an integrated toolchain that ensures consistency between specifications, generated code, and test suites. This integration is not merely organisational: the formal semantics of the model drive both the code generator and the test generator, so that correctness and coverage are defined with respect to the same mathematical object.

The formal semantics of EDAM are grounded in established techniques from behavioural type theory [111], process calculi [149], and finite state machine theory. The model extends data-aware finite state machines with dynamic participant management, role-based access control, and explicit modelling of inter-contract interactions. These extensions are essential for modern smart contract applications, particularly in DeFi and multi-party coordination scenarios. The thesis provides formal definitions of well-formedness and determinism for the model, so that the generated artefacts can be reasoned about rigorously.

Code Generation

We have developed a code generation engine that automatically produces Solidity smart contracts from EDAM specifications. The generator handles complex features including role management, state transitions, guard conditions, simultaneous assignment semantics, and inter-contract interactions. The generated code faithfully implements the formal model, ensuring that verified properties of the model are preserved in the executable code. The code generation process uses an intermediate JSON representation, which enables platform-agnostic code generation with ongoing extensions to support additional blockchain languages such as Move.

The code generator addresses several technical challenges, which include the preservation of simultaneous assignment semantics in sequential programming languages, proper handling of external call success and failure scenarios, and the implementation of reentrancy protection mechanisms. The generated code preserves the behaviour and semantics of the formal model while adhering to Solidity best practices. A notable design choice is the alignment of the generator's output with the labelled transition syntax of the model: states, operations, guards, and call tries are reflected in the structure of the generated contract, which facilitates traceability and future verification of the generator itself.

Test Generation

We have presented a comprehensive test generation methodology that produces executable test suites from EDAM specifications. The approach combines symbolic trace generation using the

formal semantics, concrete trace derivation through constraint solving, and test case generation to create test suites that achieve high coverage of the contract's behavioural space. The methodology captures both successful and failed execution scenarios, which provides comprehensive validation of contract behaviour. By deriving tests from the same semantics that underpin the model and the code generator, we obtain test oracles that are consistent with the intended behaviour of the specification.

The test generation process systematically explores the state space, ensuring coverage of edge cases and exceptional situations. The approach leverages the formal semantics to generate traces that exercise all transitions, guards, and role constraints, producing test suites that are both comprehensive and efficient. Symbolic traces are instantiated to concrete inputs via constraint solving, so that the resulting test cases are executable and meaningful. The generated test suites are executable in standard testing frameworks such as Hardhat, enabling seamless integration with existing development workflows.

Validation and Evaluation

We have developed a multi-faceted validation approach that combines code coverage analysis, mutation testing, and cross-validation with established implementations. The framework provides quantitative metrics for assessing the quality of generated code and test suites. Our evaluation on a diverse benchmark of smart contracts, including standard token contracts, decentralised finance protocols, and multi-coordinator systems, demonstrates that our approach produces contracts and test suites of comparable quality to manually developed implementations. The validation methodology is designed to stress both the code generator and the test generator: coverage and mutation metrics reveal how well the generated tests exercise the generated code, while cross-validation with reference implementations provides an independent check on behavioural correctness.

The validation results show that our generated test suites achieve high coverage across multiple metrics: statement coverage ranging from 74.49% to 95.83%, branch coverage from 60% to 95.45%, and mutation scores from 62.70% to 86.44%. These results demonstrate that the model-driven approach can produce high-quality smart contracts and comprehensive test suites, addressing the verification gap that has been a persistent challenge in smart contract development [152]. The variation across benchmarks reflects the differing complexity of contracts and the presence of defensive or platform-specific code that is not directly tied to the EDAM specification.

The evaluation also demonstrates the expressiveness of the EDAM framework through its ability to model a wide range of smart contract features, including inter-coordinator interactions, dynamic participant management, role assignment and revocation, and state-dependent behaviour. The framework successfully models contracts from the Azure repository benchmark, as well as standard contracts such as ERC20 tokens and AMM s. This breadth of coverage indicates that the model is not limited to toy examples but is applicable to real-world contract patterns encountered in practice.

VI.2 Key Achievements

The primary achievements of this thesis are the EDAM formal model and the toolchain that links it to practice. The model extends data-aware finite state machines with dynamic participant management, role-based access control, and inter-contract interactions. The toolchain—code generation and test generation driven by the same formal semantics—enables the model to be used in real development workflows. The shared semantics ensure consistency between specification, code, and tests.

The framework demonstrates that behavioural type systems provide a solid foundation for smart contract verification, enabling the development of unified frameworks that integrate modelling, code generation, test generation, and validation. The results show that model-driven approaches can produce high-quality smart contracts and comprehensive test suites. The thesis contributes both a

concrete instantiation of this idea in the EDAM toolchain and formal definitions of well-formedness and determinism that support the soundness of the approach.

Another significant achievement is the demonstration that formal methods can be practically applied to smart contract development without sacrificing expressiveness or usability. The EDAM framework provides a formal model that is both mathematically rigorous and practically applicable, enabling developers to specify complex smart contract behaviour while maintaining the ability to automatically generate executable code and comprehensive test suites. The model supports features that are essential in practice—roles, participants, inter-contract calls, guards, and state-dependent behaviour—without requiring developers to reason directly in process calculus or type theory.

The framework also demonstrates that correct-by-construction code generation is achievable for smart contracts. By generating code directly from verified models, we eliminate the verification gap that exists when code is written manually and then verified separately. The generated code is guaranteed to implement the formal model, ensuring that properties established at the model level are preserved in the executable code. This is achieved by design: the code generator is structured so that each construct in the model maps to a well-defined fragment of the target language, and the semantics of the model are reflected in the execution behaviour of the generated contract.

VI.3 Limitations and Challenges

While the EDAM framework addresses many challenges in smart contract development, several limitations and challenges remain.

Obtaining EDAM from specifications or code. The framework assumes that EDAM specifications are written or edited directly; there is no support for deriving EDAM models from other specification formats (e.g. requirements documents, graphical notations, or domain-specific languages) or for reverse-engineering existing code into a EDAM specification. Adopting the framework for legacy or third-party contracts therefore requires manually constructing the corresponding model, and workflows that start from non-EDAM specifications cannot yet be automated into the EDAM toolchain. Bidirectional transformations (from code to model) are identified as future work but are not currently available.

Target platforms and languages. The current implementation focuses primarily on Solidity as the target language, with ongoing work on Move support. Extending the framework to support additional blockchain platforms and programming languages requires careful consideration of platform-specific features and constraints. For example, different chains have different execution models, gas semantics, and built-in primitives; the intermediate representation and the code generator would need to be extended to capture these differences while preserving the semantics of the model.

Expressiveness. The expressiveness of the EDAM framework, while substantial, may not capture all possible smart contract behaviours. Some advanced features, such as complex cryptographic operations, advanced gas optimisation techniques, and certain platform-specific optimisations, may require manual implementation or extensions to the model. The framework's extensibility through the intermediate JSON representation provides a pathway for incorporating additional features, but each extension may require both semantic and syntactic adjustments to the model and the tools.

Test coverage and mutation. The test generation process may not always achieve perfect coverage or mutation scores. Some edge cases, particularly those involving complex boundary conditions or rare interaction patterns, may require additional manual test cases. The systematic approach to test generation ensures that the vast majority of contract behaviour is covered, but contracts that make heavy use of platform-specific or defensive code may exhibit gaps that are not directly tied to the EDAM specification.

Formal verification. The validation framework provides quantitative metrics but does not provide formal proofs of correctness. The combination of high coverage and mutation scores provides strong evidence of correctness, but formal verification of generated code against the model remains an area for future work. Such verification would require a formal semantics of the target language and a refinement or simulation relation between the model and the code.

Scalability. The scalability of the approach to very large contracts or multi-coordinator systems requires further investigation. While the framework has been successfully applied to a diverse range of contracts, the performance characteristics for extremely large systems—for example, trace generation and constraint solving for big state spaces—may require optimisation or alternative approaches.

VI.4 Future Work

The EDAM framework provides a solid foundation for future research and development. Several directions for future work are identified, ranging from theoretical extensions to practical improvements and new applications.

Theoretical Extensions

One important direction for future work is the extension of the formal model to support additional features that are common in modern smart contracts. These include support for time-dependent behaviour, gas-aware modelling, and advanced cryptographic operations. Extending the model to support these features would increase its applicability to a wider range of smart contract applications. Time-dependent behaviour, for example, is central to auctions, vesting schedules, and governance deadlines; gas-aware modelling would allow developers to reason about resource consumption at the specification level. Each extension would require both semantic definitions and updates to the code generator and test generator so that the integrated methodology is preserved.

The development of composition operators for EDAM specifications would enable the systematic construction of complex multi-coordinator systems from simpler components. This would support modular development and verification, enabling developers to reason about system properties by composing properties of individual components. Composition could take the form of parallel composition, sequential composition, or more domain-specific patterns; in each case, the goal would be to ensure that well-formedness, determinism, and other model-level properties are preserved under composition.

Platform Support and Code Generation

Extending the code generation capabilities to support additional blockchain platforms is an important practical direction. While Solidity support is mature and Move support is in progress, extending to other platforms such as Tezos (Michelson), Cardano (Plutus), or Cosmos would broaden the applicability of the framework. Each platform presents unique challenges: different programming models, execution semantics, and security considerations must be reflected in the intermediate representation and in the platform-specific back ends. The existing JSON-based intermediate representation provides a starting point, but platform-specific extensions and possibly platform-specific optimisation passes would be needed for each new target.

Improving the quality and efficiency of generated code is another important direction. This includes optimisation techniques for gas consumption, code size reduction, and execution efficiency. While the current generator produces correct code, optimisations could make the generated contracts more competitive with manually written implementations. Such optimisations would need to be correctness-preserving with respect to the model semantics, so that the link between specification and code is not weakened.

The development of bidirectional transformations between EDAM specifications and code would enable round-trip engineering, allowing developers to modify generated code and have those changes reflected in the model. This would support iterative development and maintenance of deployed contracts. Bidirectional transformations are technically demanding, as they require conflict resolution and consistency checks when the code and the model diverge; nevertheless, they would significantly improve the usability of the framework in long-lived projects.

Test Generation and Validation

Enhancing the test generation methodology to achieve even higher coverage and mutation scores is an important direction. This could include the development of specialised test generation strategies for specific contract patterns (e.g. token transfers, access control, or multi-step workflows), improved constraint solving techniques for complex guards and role constraints, and the integration of property-based testing approaches that generate inputs from declarative invariants. Automated test oracle generation would reduce the manual effort required to validate generated contracts: oracles that check contract behaviour against the formal model would further automate the validation process and strengthen the link between specification and tests.

Extending the validation framework to include additional metrics and analysis techniques would provide a comprehensive assessment of contract quality. This could include security-specific metrics (such as reentrancy or access-control checks), performance analysis (e.g. gas usage under representative workloads), and economic property verification for decentralised finance applications (e.g. invariants on reserves or liquidity in automated market makers).

Tooling and Usability

Improving the usability of the EDAM framework through better tooling and development environments is crucial for widespread adoption. This includes the development of visual modelling tools that allow developers to edit and inspect EDAM specifications graphically, integrated development environments with syntax highlighting, error reporting, and navigation between model and generated code, and debugging support that relates execution traces back to the model. Making the framework more accessible to developers without formal methods expertise would significantly increase its impact, and would require careful design of notations, error messages, and documentation.

The development of libraries and templates for common smart contract patterns would accelerate development and ensure best practices. These could include standard patterns for token contracts, voting systems, auction mechanisms, and other common decentralised finance primitives. Such libraries would reduce the effort required to build new specifications and would provide reference implementations that could be used for validation and teaching.

The integration of the EDAM framework with existing development workflows and tools would facilitate adoption. This includes integration with version control systems (e.g. diff and merge for model files), continuous integration pipelines (e.g. generate, test, and report coverage on each commit), and deployment tools (e.g. generation and deployment of contracts to testnets or mainnets). Seamless integration with existing toolchains would reduce the barrier to adoption and would allow teams to adopt the framework incrementally.

Applications and Case Studies

Applying the EDAM framework to additional domains and use cases would demonstrate its versatility and identify areas for improvement. This includes applications in supply chain management, digital identity, governance systems, and other blockchain-based applications beyond DeFi. Each domain may require domain-specific patterns or extensions to the model; documenting these patterns and evaluating the framework on representative case studies would strengthen the evidence for its applicability.

The development of case studies comparing EDAM-based development with traditional approaches would provide quantitative evidence of the benefits of the model-driven approach. This could include metrics such as development time, bug rates, security incidents, and maintenance costs. Controlled studies with developers using the framework on realistic tasks would complement the existing benchmark-based evaluation and shed light on usability and adoption barriers.

The application of the framework to real-world production systems would provide valuable feedback and identify practical challenges that may not be apparent in academic benchmarks. Collaborating with industry partners to apply the framework to production systems would accelerate its maturation and adoption, and would help prioritise future work on tooling, performance, and expressiveness.

Formal Verification and Security

The development of automated security analysis techniques specifically tailored to EDAM specifications would enhance the security guarantees of the framework. This could include the automatic detection of common vulnerability patterns (e.g. reentrancy, access control violations, or integer overflows), the verification of security invariants expressed at the model level, and the generation of security proofs that can be checked or communicated to auditors. Because the model has a clear semantics, many such analyses could be performed on the specification before code generation, so that security issues are caught early.

The integration of formal verification tools with the EDAM framework would enable the verification of temporal properties, liveness guarantees, and security invariants. This would provide stronger guarantees than testing alone, enabling the verification of properties that are difficult to test (such as absence of deadlock or fairness in multi-party protocols). Verification could target either the model or the generated code; in the latter case, a refinement or simulation relation between the model and the code would be needed to carry model-level properties over to the implementation.

The development of techniques for verifying the correctness of the code generator itself would provide additional assurance. While the current approach relies on testing and validation, formal verification of the generator would provide mathematical guarantees about the correctness of the transformation from model to code. This could take the form of a proof that the generated code refines the model semantics, or of a certified compiler that produces code together with a proof of refinement.

Bibliography

[1] Tesnim Abdellatif and Kei-Leo Brousmiche. Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2 2018. doi:10.1109/NTMS.2018.8328737.

[2] João Afonso. Mechanisms for Modeling and Validation of Smart Contracts. Master's thesis, Departamento de Informática, NOVA School of Science and Technology, 2023. Advisor: António Ravara. URL: https://run.unl.pt/bitstream/10362/167654/1/Afonso_2023.pdf.

[3] João Afonso, Elvis Konjoh Selabi, Maurizio Murgia, António Ravara, and Emilio Tuosto. TRAC: A Tool for Data-Aware Coordination (with an Application to Smart Contracts). In Ilaria and Francesco Castellani Tiezzi, editors, *Proceedings of the 26th IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION), Held as Part of the 19th International Federated Conference on Distributed Computing Techniques (DisCoTec)*, pages 239–257. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-62697-5_13.

[4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, second edition, 2006. First edition published in 1986. doi:10.1061/40854(211)16.

[5] Issam Al-Azzoni, Reiko Heckel, and Zobia Erum. Dynamic Role-Based Access Control Scenarios for Smart Contracts: Graph Rewriting for Testing Domain-specific Models. *The Journal of Object Technology*, 24:2:1, 2025. doi:10.5381/jot.2025.24.2.a4.

[6] Elvira Albert, Emanuele De Angelis, Fabio Fioravanti, Alejandro Hernández-Cerezo, and Giulia Matricardi. Verifying Smart Contracts in Yul via Transformation to CHC by Interpreter Specialization. In *Logic-Based Program Synthesis and Transformation*, pages 22–39. Springer Nature Switzerland, 2026. doi:10.1007/978-3-032-04848-6_2.

[7] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. Verification of smart contracts: A survey. *Pervasive and Mobile Computing*, 67:101227, 9 2020. doi:10.1016/j.pmcj.2020.101227.

[8] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. A formal verification approach for composite smart contracts security using FSM. *Journal of King Saud University - Computer and Information Sciences*, 35:70–86, 1 2023. doi:10.1016/j.jksuci.2022.08.029.

[9] Sarra Alqahtani, Xinchi He, Rose Gamble, and Papa Mauricio. Formal Verification of Functional Requirements for Smart Contract Compositions in Supply Chain Management Systems. 2020. doi:10.24251/HICSS.2020.650.

[10] Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. SolCMC: Solidity Compiler's Model Checker. In *International Conference on Computer Aided Verification*, pages 325–338, 2022. doi:10.1007/978-3-031-13185-1_16.

[11] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. doi:10.1016/0304-3975(94)90010-8.

[12] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77. ACM, 1 2018. doi:10.1145/3167084.

[13] Diego Norberto Senarruzza Anabia. Bisimulación de Data-aware Communicating Finite State Machines con Propiedades en las Acciones, 2023. URL: https://bibliotecadigital.exactas.uba.ar/download/seminario/seminario_nCOM000539_SenarruzzaAnabia.pdf.

[14] 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*. ACM, 2018. doi:10.1145/3190508.3190538.

[15] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: a smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 215–228, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373829.

[16] Andreas M Antonopoulos and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, 2018. URL: https://books.google.it/books?id=nJJ5DwAAQBAJ.

[17] Larry Apfelbaum and John Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300, 1997. doi:10.1080/105172397243123.

[18] Nadiah Arsat, Abu Jor Al Gefari, Md Amin Ullah Sheikh, and Falentino Sembiring. Smart Contracts Vulnerabilities and Countermeasures in Blockchain Security. In *2025 International Conference on Metaverse and Current Trends in Computing (ICMCTC)*, pages 1–6. IEEE, 4 2025. doi:10.1109/ICMCTC62214.2025.11196566.

[19] Imran Ashraf, Xiaoxue Ma, Bo Jiang, and W. K. Chan. GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities. *IEEE Access*, 8, 2020. doi:10.1109/ACCESS.2020.2995183.

[20] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *POST*, volume 10204, pages 164–186. Springer, 2017. doi:10.1007/978-3-662-54455-6_8.

[21] John W Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132, 1959. Original introduction of Backus–Naur Form (BNF) notation for formal grammar specification. doi:10.1016/b978-0-08-009217-1.50026-8.

[22] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. Formal Modeling and Verification of Smart Contracts. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, pages 322–326. ACM, 2 2018. doi:10.1145/3185089.3185138.

[23] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61:75–113, 7 2006. doi:10.1016/j.scico.2005.10.008.

[24] Sebastian Banescu, Morena Barboni, Andrea Morichetta, Andrea Polini, and Edward Zulkoski. Enhanced mutation testing of smart contracts in support of code inspection. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 558–566. IEEE, 5 2024. doi:10.1109/ICBC59979.2024.10634403.

[25] Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography Automata. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12134 LNCS, pages 86–106. Springer, 2020. doi:10.1007/978-3-030-50029-0_6.

[26] Morena Barboni, Andrea Morichetta, and Andrea Polini. SuMo: A Mutation Testing Strategy for Solidity Smart Contracts. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 50–59. IEEE, 5 2021. doi:10.1109/AST52587.2021.00014.

[27] Morena Barboni, Andrea Morichetta, Andrea Polini, Sebastian Banescu, and Edward Zulkoski. Mutation Testing of Smart Contracts As a Service. In Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini, editors, *Quality of Information*

and Communications Technology, pages 93–109, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-70245-7_7.

[28] Morena Barboni, Andrea Morichetta, Andrea Polini, and Francesco Casoni. ReSuMo: a regression strategy and tool for mutation testing of solidity smart contracts. *Software Quality Journal*, 32:225–253, 3 2024. doi:10.1007/s11219-023-09637-1.

[29] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. *Springer*, pages 305–343, 2018. doi:10.1007/978-3-319-10575-8_11.

[30] Massimo Bartoletti, Silvia Crafa, and Enrico Lipparini. Formal Verification in Solidity and Move: Insights from a Comparative Analysis. In Diego Marmsoler and Meng Xu, editors, *6th International Workshop on Formal Methods for Blockchains (FMBC 2025)*, volume 129 of *Open Access Series in Informatics (OASIcs)*, pages 3:1–3:18, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.FMBC.2025.3.

[31] Massimo Bartoletti, Angelo Ferrando, Enrico Lipparini, and Vadim Malvone. Solvent: Liquidity Verification of Smart Contracts. In Nikolai Kosmatov and Laura Kovács, editors, *Integrated Formal Methods*, pages 256–266, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-76554-4_14.

[32] Davide Basile, Pierpaolo Degano, Gian-Luigi Ferrari, and Emilio Tuosto. Playing with Our CAT and Communication-Centric Applications. In Elvira Albert and Ivan Lanese, editors, *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 9688, pages 62–73, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-39570-8_5.

[33] Davide Basile and Maurice H. ter Beek. Contract Automata Library. *Science of Computer Programming*, 221:102841, 9 2022. doi:10.1016/j.scico.2022.102841.

[34] Davide Basile and Maurice H. ter Beek. A Runtime Environment for Contract Automata. In *Proceedings of the International Symposium on Formal Methods (FM)*, volume 14000, pages 550–567. Springer, 2023. doi:10.1007/978-3-031-27481-7_31.

[35] Davide Basile, Maurice H. ter Beek, and Rosario Pugliese. Synthesis of Orchestrations and Choreographies: Bridging the Gap between Supervisory Control and Coordination of Services. *Logical Methods in Computer Science*, Volume 16, Issue 2, 6 2020. doi:10.23638/LMCS-16(2:9)2020.

[36] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28:41–48, 5 2011. doi:10.1109/MS.2011.27.

[37] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal Verification of Smart Contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 10 2016. doi:10.1145/2993600.2993611.

[38] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. Move: A Language with Programmable Resources, 2019. White paper, Libra Association. URL: https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2019-06-18.pdf.

[39] Kees Boogaard. *A model-driven approach to smart contract development*. PhD thesis, 2028. URL: https://studenttheses.uu.nl/handle/20.500.12932/29326.

[40] Chiara Braghin, Giuseppe Del Castillo, Elvinia Riccobene, and Simone Valentini. Using Symbolic Model Execution to Detect Vulnerabilities of Smart Contracts. In Michael Leuschel and Fuyuki Ishikawa, editors, *Rigorous State-Based Methods - 11th International Conference, ABZ 2025, Düsseldorf, Germany, June 10-13, 2025, Proceedings*, volume 15728, pages 31–51. Springer, 2026. doi:10.1007/978-3-031-94533-5_3.

[41] Chiara Braghin, Elvinia Riccobene, and Simone Valentini. Modeling and verification of smart contracts with Abstract State Machines. In Jiman Hong and Juw Won Park, editors, *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 1425–1432. ACM, 4 2024. doi:10.1145/3605098.3636040.

[42] Konstantin Britikov, Ilia Zlatkin, Grigory Fedyukovich, Leonardo Alt, and Natasha Sharygina. SolTG: A CHC-Based Solidity Test Case Generator. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 14681 LNCS, pages 466–479, 2024. doi:10.1007/978-3-031-65627-9_23.

[43] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J Summers. Rich specifications for Ethereum smart contract verification. *Proceedings of the ACM on Programming Languages*, 5, 2021. doi:10.1145/3485523.

[44] Vitalik Vitalin Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. By Vitalik Buterin. *Whitepaper*, 2014.

[45] Andrea Canidio and Vincent Danos. Commitment against front-running attacks. *Management Science*, 70(7):4429–4440, 2024. doi:10.1287/mnsc.2023.01239.

[46] Marco Carbone. Session-based Choreography with Exceptions. *Electronic Notes in Theoretical Computer Science*, 241:35–55, 2009. Proceedings of the First Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2008). doi:10.1016/j.entcs.2009.06.003.

[47] M. Castro and B. Liskov. Byzantine fault tolerance can be fast. In *Proceedings International Conference on Dependable Systems and Networks*, pages 513–518. IEEE Comput. Soc, 1999. doi:10.1109/DSN.2001.941437.

[48] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342, 2014. doi:10.1007/978-3-319-08867-9_22.

[49] Certora. Certora: Formal Verification Platform for Smart Contracts, 2024. Formal verification tool for smart contracts. URL: https://www.certora.com/.

[50] Patrick Chapman, Dianxiang Xu, Lin Deng, and Yin Xiong. Deviant: A Mutation Testing Tool for Solidity Smart Contracts. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 319–324. IEEE, 7 2019. doi:10.1109/Blockchain.2019.00050.

[51] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys*, 53:1–43, 5 2021. doi:10.1145/3391195.

[52] Yuchiro Chinen, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. RA: Hunting for Re-Entrancy Attacks in Ethereum Smart Contracts via Static Analysis. In *Proceedings - 2020 IEEE International Conference on Blockchain, Blockchain 2020*, 2020. doi:10.1109/Blockchain50366.2020.00048.

[53] Tzu chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science*, Volume 13, Issue 2:135–146, 6 2017. doi:10.23638/LMCS-13(2:12)2017.

[54] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:410–425, 3 2000. doi:10.1007/s100090050046.

[55] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850, pages 15–58. Springer, 2013. doi:10.1007/978-3-642-36946-9_3.

[56] Edmund M. Clarke. Model checking. pages 54–56. MIT press, 1997. doi:10.1007/BFb0058022.

[57] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Transactions on Programming Languages and Systems*, 42:1–82, 9 2020. doi:10.1145/3417516.

[58] Ethereum Community. Code Coverage for Solidity Smart-Contracts, 2020. URL: https://hardhat.org/docs/guides/testing/code-coverage.

[59] Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In Alan J Hu and Moshe Y Vardi, editors, *Computer Aided Verification, 10th Inter-*

national Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings, volume 1427, pages 268–279. Springer, 1998. doi:10.1007/BFb0028751.

[60] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. ChorChain: A Blockchain-Based Framework for Executing and Auditing BPMN Choreographies. In Janiesch Christian, Chiara, Di Francescomarino, Grisold Thomas, Kumar Akhil, Mendling Jan, T. Pentland Brian, Reijers Hajo A., Weske Mathias, and Winter Robert, editors, *BPM (PhD/Demos)*, volume 3216, pages 132–136. CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3216/paper_254.pdf.

[61] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. Engineering Trustable and Auditable Choreography-based Systems Using Blockchain. *ACM Trans. Manage. Inf. Syst.*, 13(3), February 2022. doi:10.1145/3505225.

[62] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. Blockchain-Based Execution of BPMN Choreographies with Multiple Instances. *Distributed Ledger Technologies: Research and Practice*, 4:1–21, 6 2025. doi:10.1145/3637555.

[63] Silvia Crafa and Cosimo Laneve. Liquidity Analysis in Resource-Aware Programming. In Silvia Lizeth Tapia Tarifa and José Proença, editors, *Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings*, volume 13712, pages 205–221. Springer, 2022. doi:10.1007/978-3-031-20872-0_12.

[64] Silvia Crafa, Cosimo Laneve, Giovanni Sartor, and Adele Veschetti. Pacta Sunt Servanda: Legal Contracts in Stipula. *Science of Computer Programming (SCP)*, 225, 2023. doi:10.1016/J.SCICO.2022.102911.

[65] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pages 285–294. ACM, 5 1999. doi:10.1145/302405.302640.

[66] Lafhis Universidad de Buenos Aires. Contractor Validation Tool, 2010. Accessed: 2025. URL: https://lafhis.dc.uba.ar/dependex/contractor/Welcome.html.

[67] Massimiliano de Leoni, Paolo Felli, and Marco Montali. A Holistic Approach for Soundness Verification of Decision-Aware Process Models. In Juan Trujillo, Karen C Davis, Xiaoyong Du, Zhanhuai Li, Tok Wang Ling, Guoliang Li, and Mong-Li Lee, editors, *Proceedings of the 37th International Conference on Conceptual Modeling (ER)*, volume 11157, pages 219–235. Springer, 2018. doi:10.1007/978-3-030-00847-5_17.

[68] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Internatioanl Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[69] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. *SIGPLAN Not.*, 46(1):435–446, January 2011. doi:10.1145/1925844.1926435.

[70] Monika di Angelo and Gernot Salzer. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78. IEEE, 4 2019. doi:10.1109/DAPPCON.2019.00018.

[71] ConsenSys Diligence. Mythril: A symbolic-execution-based security analysis tool for EVM bytecode, 2018. Available: https://github.com/ConsenSys/mythril. URL: https://github.com/ConsenSysDiligence/mythril.

[72] Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab, and Simon Bliudze. Combine and conquer: Relating BIP and Reo. *Journal of Logical and Algebraic Methods in Programming*, 86:134–156, 1 2017. doi:10.1016/j.jlamp.2016.09.008.

[73] John R. Douceur. The Sybil Attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 251–260. Springer-Verlag, 2002. doi:10.1007/3-540-45748-8_24.

[74] Stefan Driessen, Dario Di Nucci, Geert Monsieur, Damian A. Tamburri, and Willem-Jan van den Heuvel. Automated Test-Case Generation for Solidity Smart Contracts: the AGSolT Approach and its Evaluation. *CoRR*, 4 2022. URL: http://arxiv.org/abs/2102.08864.

[75] S.W. Driessen, D. Di Nucci, D.A. Tamburri, and W.J. van den Heuvel. SolAR: Automated test-suite generation for solidity smart contracts. *Science of Computer Programming*, 232:103036, 1 2024. doi:10.1016/j.scico.2023.103036.

[76] Jinhu Du, Song Huang, Xingya Wang, Changyou Zheng, and Jinlei Sun. Test Case Generation for Ethereum Smart Contract based on Data Dependency Analysis of State Variable. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 710–720. IEEE, 12 2022. doi:10.1109/QRS57517.2022.00077.

[77] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 530–541. ACM, 6 2020. doi:10.1145/3377811.3380364.

[78] E. Allen EMERSON. Temporal and Modal Logic. In *Formal Models and Semantics*, pages 995–1072. Elsevier, 1990. doi:10.1016/B978-0-444-88074-1.50021-4.

[79] Ittay Eyal and Emin Gün Sirer. Majority Is Not Enough: Bitcoin Mining Is Vulnerable. In *Financial Cryptography and Data Security*, volume 8437, pages 436–454. Springer, 2014. doi:10.1007/978-3-662-45472-5_28.

[80] Afonso Falcão, Andreia Mordido, and Vasco T. Vasconcelos. Protocol-Based Smart Contract Generation. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, volume 13412, pages 555–582. Springer, 2023. doi:10.1007/978-3-031-32415-4_34.

[81] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 5 2019. doi:10.1109/WETSEB.2019.00008.

[82] Rim Ben Fekih, Mariam Lahami, Salma Bradai, and Mohamed Jmaiel. Formal Verification of ERC-Based Smart Contracts: A Systematic Literature Review. *IEEE Access*, 13:11396–11422, 2025. doi:10.1109/ACCESS.2025.3527158.

[83] Rim Ben Fekih, Mariam Lahami, Mohamed Jmaiel, and Salma Bradai. Formal Verification of Smart Contracts Based on Model Checking: An Overview. In *Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE*, 2023. doi:10.1109/WETICE57085.2023.10477834.

[84] Mahdi Fooladgar, Amin Arefzadeh, and Fathiyeh Faghih. TestSmart: A Tool for Automated Generation of Effective Test Cases for Smart Contracts. In *2021 11th International Conference on Computer Engineering and Knowledge (ICCKE)*, pages 476–481. IEEE, 10 2021. doi:10.1109/ICCKE54056.2021.9721448.

[85] Ethereum Foundation. Remix IDE: Browser-based Solidity IDE, 2024. Browser-based development environment for Solidity. URL: https://remix.ethereum.org/.

[86] Nomic Foundation. Hardhat 3: Rust-powered Solidity Tests, 2025. URL: https://hardhat.org/.

[87] 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. IEEE, 9 2016. doi:10.1109/FAS-W.2016.53.

[88] H. T. M. Gamage, H. D. Weerasinghe, and N. G. J. Dias. A Survey on Blockchain Technology Concepts, Applications, and Issues. *SN Computer Science*, 1:114, 3 2020. doi:10.1007/s42979-020-00123-0.

[89] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 36:1–44, 10 2014. doi:10.1145/2629609.

[90] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42:191–225, 11 2005. doi:10.1007/s00236-005-0177-z.

[91] Simon Gay and António Ravara. Behavioural Types: from Theory to Tools. pages 1–412. River Publishers, 2017. doi:10.13052/rp-9788793519817.

[92] Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. Design-By-Contract for Flexible Multiparty Session Protocols. In *ECOOP 2022 - European Conference on Object-Oriented Programming*, volume 222, Berlin (DE), Germany, June 2022. doi:10.4230/LIPIcs.ECOOP.2022.8.

[93] Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. Design-by-Contract for Flexible Multiparty Session Protocols (Artifact). *Dagstuhl Artifacts Series*, 8:21:1–21:5, 2022. doi:10.4230/DARTS.8.2.21.

[94] Paola Giannini, Anna-Lena Lamprecht, and Tiziana Margaria. Exploring the potential of global types for adding a choreography perspective to the jabc framework. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 368–376, 2016. URL: https://ieeexplore.ieee.org/document/7954382.

[95] Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastian Uchitel. Predicate abstractions for smart contract validation. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, pages 289–299. ACM, 10 2022. doi:10.1145/3550355.3552462.

[96] Javier Godoy, Eden Torres, Juan P. Galeotti, Diego Garbervetsky, and Sebastian Uchitel. Automated construction of predicate abstractions for smart contract validation. *Software and Systems Modeling*, 10 2025. doi:10.1007/s10270-025-01333-x.

[97] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2:1–27, 2018. doi:10.1145/3276486.

[98] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560. ACM, 7 2020. doi:10.1145/3395363.3404366.

[99] Zhifeng Guo. Blockchain-enhanced smart contracts for formal verification of IoT access control mechanisms. *Alexandria Engineering Journal*, 118:315–324, 4 2025. doi:10.1016/j.aej.2024.12.109.

[100] Hacken. Top 10 Smart Contract Vulnerabilities in 2025 (With Real Hacks & How to Prevent Them), 2025. Accessed: 2025-01-XX. URL: https://hacken.io/discover/smart-contract-vulnerabilities/.

[101] Michael Hahn, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Modeling and execution of data-aware choreographies: an overview. *Springer*, 33:329–340, 2 2018. doi:10.1007/S00450-017-0387-Y.

[102] 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, 4 2022. doi:10.1016/j.infsof.2021.106762.

[103] Mohammad Hamdaqa, Lucas Alberto Pineda Metz, and Ilham Qasse. iContractML: A Domain-Specific Language for Modeling and Deploying Smart Contracts onto Multiple Blockchain Platforms. In *Proceedings of the 12th System Analysis and Modelling Conference*, pages 34–43. Association for Computing Machinery, 2020. doi:10.1145/3419804.3421454.

[104] Dominik Harz and William Knottenbelt. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. *CoRR*, abs/1809.09805, 11 2018. doi:10.48550/arXiv.1809.09805.

[105] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *Proceedings of the 24th USENIX Security Symposium*, pages 129–144. USENIX Association, 2015. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman.

[106] M Hennessy and H Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 2 1995. doi:10.1016/0304-3975(94)00172-F.

[107] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018*

IEEE 31st Computer Security Foundations Symposium (CSF), pages 204–217, 2018. doi:10.1109/CSF.2018.00022.

[108] Yoichi Hirai. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Fundamental Approaches to Software Engineering (FASE)*, volume 10202, pages 520–535. Springer, 2017. doi:10.1007/978-3-319-70278-0_33.

[109] Daniel Hirschkoff. Automatically Proving Up–to Bisimulation. *Electronic Notes in Theoretical Computer Science*, 18:75–89, 1998. doi:10.1016/S1571-0661(05)80251-8.

[110] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 5 1997. doi:10.1109/32.588521.

[111] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the European Symposium on Programming (ESOP)*, volume 1381, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[112] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1328438.1328472.

[113] Huawei Huang, Jianru Lin, Baichuan Zheng, Zibin Zheng, and Jing Bian. When Blockchain Meets Distributed File Systems: An Overview, Challenges, and Open Issues. *IEEE Access*, 8:50574–50586, 2020. Survey on DAG-based distributed ledger technologies. doi:10.1109/ACCESS.2020.2979881.

[114] Adnan Imeri, Nazim Agoulmine, Djamel Khadraoui, Adnan Imeri, Nazim Agoulmine, Djamel Khadraoui, Smart Contract, Adnan Imeri, Nazim Agoulmine, and Djamel Khadraoui. Smart Contract modeling and verification techniques: A survey. *8th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2020)*, 2020. URL: https://hal.science/hal-02495158.

[115] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37:649–678, 9 2011. doi:10.1109/TSE.2010.62.

[116] Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269. ACM, 9 2018. doi:10.1145/3238147.3238177.

[117] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1695–1712. IEEE, 5 2020. doi:10.1109/SP40000.2020.00066.

[118] Mantas Jurgelaitis, Lina Ceponiene, and Rita Butkiene. Solidity Code Generation From UML State Machines in Model-Driven Smart Contract Development. *IEEE Access*, 10:33465–33481, 2022. doi:10.1109/ACCESS.2022.3162227.

[119] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings 2018 Network and Distributed System Security Symposium*, pages 213–223. Internet Society, 2018. doi:10.14722/ndss.2018.23082.

[120] Youssef Khaoulaj, Youssef Hamdaoui, and Abdelilah Maach. Survey on Smart Contract Security: Challenges, Techniques, and Future Directions. In *Lecture Notes in Networks and Systems*, volume 1402 LNNS, 2025. doi:10.1007/978-3-031-91334-1_48.

[121] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, 7 1976. PhD Thesis. doi:10.1145/360248.360252.

[122] Sunny King and Scott Nadal. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake, 2012. URL: https://bitcoin.peryaudo.org/vendor/peercoin-paper.pdf.

[123] Elvis Konjoh Selabi, Maurizio Murgia, António Ravara, and Emilio Tuosto. Automatic Code and Test Generation of Smart Contracts from Coordination Models (Artifact), feb 2026. doi:10.5281/zenodo.18475933.

[124] Moez Krichen, Mariam Lahami, and Qasem Abu Al-Haija. Formal Methods for the Verification of Smart Contracts: A Review. In *2022 15th International Conference on Security of*

Information and Networks (SIN), pages 01–08. IEEE, 11 2022. doi:10.1109/SIN56466.2022.9970534.

[125] Jae Kwon. Tendermint: Consensus without Mining. *Tendermint*, 2014. Accessed: 2026. doi:10.1186/1756-0381-7-7.

[126] Cosimo Laneve. Liquidity Analysis in Resource-Aware Programming. *SSRN Electronic Journal*, 2022. doi:10.2139/ssrn.4282044.

[127] Huimin Lin. Symbolic transition graph with assignment. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, pages 50–65, 1996. doi:10.1007/3-540-61604-7_47.

[128] Wen ling Huang, Niklas Krafczyk, and Jan Peleska. Exhaustive property oriented model-based testing with symbolic finite state machines. *Science of Computer Programming*, 231:103005, 1 2024. doi:10.1016/j.scico.2023.103005.

[129] Jing Liu and Zhentian Liu. A Survey on Security Verification of Blockchain Smart Contracts. *IEEE Access*, 7:77894–77904, 2019. doi:10.1109/ACCESS.2019.2921624.

[130] Ye Liu, Yi Li, Shang-Wei Lin, and Qiang Yan. ModCon: a model-based testing platform for smart contracts. In Prem Devanbu, Myra B Cohen, and Thomas Zimmermann, editors, *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1601–1605. ACM, 11 2020. doi:10.1145/3368089.3417939.

[131] Eric Lombrozo, Johnson Lau, and Pieter Wuille. BIP 141: Segregated Witness (Consensus layer), 2015. Assigned: 2015-12-21. URL: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[132] Hao Luo, Yuhao Lin, Xiao Yan, Xintong Hu, Yuxiang Wang, Qiming Zeng, Hao Wang, and Jiawei Jiang. Guiding LLM-based Smart Contract Generation with Finite State Machine. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 5869–5877. International Joint Conferences on Artificial Intelligence Organization, 9 2025. doi:10.24963/ijcai.2025/653.

[133] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 10 2016. doi:10.1145/2976749.2978309.

[134] Gabor Madl, Luis Bathen, German Flores, and Divyesh Jadav. Formal Verification of Smart Contracts Using Interface Automata. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 556–563. IEEE, 7 2019. doi:10.1109/Blockchain.2019.00081.

[135] Daniele Magazzeni, Peter McBurney, and William Nash. Validation and Verification of Smart Contracts: A Research Agenda. *Computer*, 50:50–57, 2017. doi:10.1109/MC.2017.3571045.

[136] Davide Mancino, Alberto Leporati, Marco Viviani, and Giovanni Denaro. Exploiting ethereum after "the merge": The interplay between pos and mev strategies, 2023. doi:10.2139/ssrn.4447330.

[137] Felix Mannhardt, Massimiliano de Leoni, Hajo A. Reijers, and Wil M. P. van der Aalst. Balanced multi-perspective checking of process conformance. *Computing*, 98:407–437, 4 2016. doi:10.1007/s00607-015-0441-1.

[138] Adrian Manning. Solidity security: Comprehensive list of known attack vectors and common anti-patterns. *Sigma Prime*, 20, 2018. doi:10.7287/peerj-cs.25v0.2/reviews/2.

[139] Anastasia Mavridou and Aron Laszka. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Revised Selected Papers*, volume 10957, pages 523–540. Springer, 2018. doi:10.1007/978-3-662-58387-6_28.

[140] Anastasia Mavridou and Aron Laszka. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In *Proceedings of the International Conference on Integrated Formal Methods (IFM)*, volume 11023, pages 270–277. Springer, 2018. doi:10.1007/978-3-319-89722-6_11.

[141] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *23rd International Conference onFi-*

nancial Cryptography and Data Security, FC 2019, Revised Selected Papers, volume 11598, pages 446–465. Springer, 2019. doi:10.1007/978-3-030-32101-7_27.

[142] Afef Jmal Maâlej and Mariam Lahami. White-Box Mutation Testing of Smart Contracts: A Quick Review. In *International Conference on Verification and Evaluation of Computer and Communication Systems*, pages 135–148, 2024. doi:10.1007/978-3-031-49737-7_10.

[143] Silvio Meneguzzo, Elvis Gerardin Konjoh Selabi, Alfredo Favenza, Valentina Gatteschi, and Claudio Schifanella. Enabling citizen sustainable behaviors in urban mobility through blockchain and tokenization. *taylorfrancis.comS Meneguzzo, EGK Selabi, A Favenza, V Gatteschi, C SchifanellaBlockchain Technology in the Automotive Industry, 2024 • taylorfrancis.com*, pages 245–279, 2 2024. doi:10.1201/9781003450306-17.

[144] Ralph C. Merkle. Protocols for Public Key Cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, volume 26, pages 122–122. IEEE, 4 1980. doi:10.1109/SP.1980.10006.

[145] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1990.

[146] Feng Mi, Chen Zhao, Zhuoyi Wang, Sadaf MD Halim, Xiaodi Li, Zhouxiang Wu, Latifur Khan, and Bhavani Thuraisingham. An Automated Vulnerability Detection Framework for Smart Contracts. *Distributed Ledger Technologies: Research and Practice*, 11 2024. doi:10.1145/3705616.

[147] Microsoft. Simple Marketplace Sample Application for Azure Blockchain Workbench, 2019. URL: https://github.com/Azure-Samples/blockchain/blob/master/blockchain-workbench/application-and-smart-contract-samples/simple-marketplace/readme.md.

[148] Microsoft. The blockchain workbench. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples, 2019.

[149] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge university press, 1999. URL: https://www.cambridge.org/core/product/identifier/S0956796801224113/type/journal_article.

[150] Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Finding smart contract vulnerabilities with ConCert's property-based testing framework. *4th International Workshop on Formal Methods for Blockchains (FMBC 2022)*, 105:2:1–2:13, 8 2022. Keywords: Smart Contracts, Formal Verification, Property-Based Testing, Coq. doi:10.4230/OASIcs.FMBC.2022.2.

[151] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 11 2019. doi:10.1109/ASE.2019.00133.

[152] Yvonne Murray and David A. Anisi. Survey of Formal Verification Methods for Smart Contracts on Blockchain. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–6. IEEE, 6 2019. doi:10.1109/NTMS.2019.8763832.

[153] Glenford J Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley, third edition, 1 2012. First edition published in 1979. doi:10.1002/9781119202486.

[154] Malte Möser, Ittay Eyal, and Emin Gün Sirer. Bitcoin Covenants. In *Financial Cryptography and Data Security*, pages 126–141. Springer Berlin Heidelberg, 2016. doi:10.1007/978-3-662-53357-4_9.

[155] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag, 2006. doi:10.1007/3-540-32653-7.

[156] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. White paper. URL: https://bitcoin.org/bitcoin.pdf.

[157] Wonhong Nam and Hyunyoung Kil. Formal Verification of Blockchain Smart Contracts via ATL Model Checking. *IEEE Access*, 10:8151–8162, 2022. doi:10.1109/ACCESS.2022.3143145.

[158] Keerthi Nelaturu, Anastasia Mavridou, Emmanouela Stachtiari, Andreas Veneris, and Aron Laszka. Correct-by-Design Interacting Smart Contracts and a Systematic Approach for

Verifying ERC20 and ERC721 Contracts with VeriSolid. *IEEE Transactions on Dependable and Secure Computing*, 20:3110–3127, 2 2023. doi:10.1109/TDSC.2022.3200840.

[159] Enzo Nicourt, Benjamin Kushigian, Chandrakana Nandi, and Yliès Falcone. Using Mutation Testing To Improve and Minimize Test Suites for Smart Contracts. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 341–352. IEEE, 5 2024. doi:10.1109/ICST60714.2024.00038.

[160] Oscar Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28:1–15, 10 1993. doi:10.1145/167962.167976.

[161] Mitchell Olsthoorn, Dimitri Stallenberg, Arie Van Deursen, and Annibale Panichella. SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 202–206. IEEE, 5 2022. doi:10.1109/ICSE-Companion55297.2022.9793754.

[162] Object Management Group (OMG). *Business Process Model and Notation*, volume 125. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-33155-8.

[163] OpenZeppelin. OpenZeppelin Contracts: A Library for Secure Smart Contract Development, 2025. URL: https://docs.openzeppelin.com/contracts.

[164] Simone Orlando, Vairo Di Pasquale, Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Corinne, a Tool for Choreography Automata. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13077 LNCS:82–92, 2021. doi:10.1007/978-3-030-90636-8_5.

[165] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, pages 167–183, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg. doi:10.1007/BFb0017309.

[166] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677. IEEE, 5 2020. doi:10.1109/SP40000.2020.00024.

[167] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9:223–252, 1977. doi:10.1145/356698.356702.

[168] Nenad Petrović and Issam Al-Azzoni. Model-driven smart contract generation leveraging ChatGPT. In *International conference on systems engineering*, pages 387–396, 2023. doi:10.1007/978-3-031-40579-2_37.

[169] Carlos G. Lopez Pombo, Agustín E. Martinez-Suñé, and Emilio Tuosto. A dynamic temporal logic for quality of service in choreographic models. *Theoretical Computer Science*, 1043:115247, 7 2025. doi:10.1016/j.tcs.2025.115247.

[170] Serguei Popov. The Tangle, 2016. IOTA Whitepaper. URL: https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9c4a3a218e1f5/iota1_4_3.pdf.

[171] Xudong Qin, Simon Bliudze, Eric Madelaine, Zechen Hou, Yuxin Deng, and Min Zhang. SMT-based generation of symbolic automata. *Acta Informatica*, 57:627–656, 10 2020. doi:10.1007/s00236-020-00367-6.

[172] Parthasarathi R. and Puneet Kaushal. Tackle the Smart Contract Vulnerabilities. In *Encyclopedia of Criminal Activities and the Deep Web*, pages 919–931. IGI Global, 1 2020. doi:10.4018/978-1-5225-9715-5.ch062.

[173] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *Proceedings 2019 Network and Distributed System Security Symposium*, pages 1183–1198. Internet Society, 2019. doi:10.14722/ndss.2019.23413.

[174] Rootstrap. How to Create Your Own DeFi AMM - Easy Guide, 2023. Accessed: 2025-05-27. URL: https://www.rootstrap.com/blog/how-to-create-your-own-defi-amm-easy-guide.

[175] Meni Rosenfeld. Analysis of Hashrate-Based Double Spending. *arXiv preprint arXiv:1402.2009*, 2014. Analysis of 51 URL: https://arxiv.org/abs/1402.2009.

[176] Grigore Rosu. K: A Rewriting-Based Framework for Computations–Preliminary version–. Technical report, 2007. URL: https://hdl.handle.net/2142/11434.

[177] Noama Fatima Samreen and Manar H Alalfi. Smartscan: an approach to detect denial of service vulnerability in ethereum smart contracts. In *2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 17–26. IEEE, 2021. doi:10.1109/wetseb52558.2021.00010.

[178] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a Smart Contract Intermediate-Level LAnguage. *Pacific Rim Dependable Computing Conference (PRDC)*, pages 335–336, 1 2018. URL: http://arxiv.org/abs/1801.00687.

[179] 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.

[180] Neeraj Kumar Singh, Akshay M. Fajge, Raju Halder, and Md. Imran Alam. Formal verification and code generation for solidity smart contracts. In *Distributed Computing to Blockchain*, pages 125–144. Elsevier, 2023. doi:10.1016/B978-0-323-96146-2.00028-0.

[181] Rajeev Singh, Sudeep Tanwar, and Teek Parval Sharma. Utilization of blockchain for mitigating the distributed denial of service attacks. *Security and Privacy*, 3(3):e96, 2020. doi:10.1002/spy2.96.

[182] Marek Skotnica, Jan Klicpera, and Robert Pergl. Towards model-driven smart contract systems-code generation and improving expressivity of smart contract modeling. *Proceedings of the 20th CIAO*, pages 1–15, 2020. URL: https://ceur-ws.org/Vol-2825/paper1.pdf.

[183] Derek Sorensen. Towards Formally Specifying and Verifying Smart Contract Upgrades in Coq. In *OpenAccess Series in Informatics*, volume 118, 2024. doi:10.4230/OASIcs.FMBC.2024.7.

[184] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12:157–171, 1 1986. doi:10.1109/TSE.1986.6312929.

[185] Truffle Suite. Truffle: Development Framework for Ethereum, 2024. Development framework for Ethereum smart contracts. URL: https://archive.trufflesuite.com/.

[186] Dmitrii Suvorov and Vladimir Ulyantsev. Smart Contract Design Meets State Machine Synthesis: Case Studies, 2019. URL: https://arxiv.org/abs/1906.02906, arXiv:1906.02906.

[187] Nikolai M Suvorov, Irina A Lomazova, and Andrey Rivkin. Soundness Correction of Data Petri Nets. *arXiv preprint arXiv:2410.21188*, 2024. doi:10.1109/access.2025.3602229.

[188] Nick Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 18:28, 1996. URL: https://cir.nii.ac.jp/crid/1370588380617942162.

[189] Nick Szabo. The Idea of Smart Contracts, 1997. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_idea.html.

[190] Zhou Teng, Bissyandé Tegawendé F., Klein Jacques, Liu Zhe, Li Li, and Kui Liu. SMARTGIFT: Learning to Generate Practical Inputs for Testing Smart Contracts, 9 2021. doi:10.26226/morressier.613b5418842293c031b5b61a.

[191] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. SmartCheck: Static analysis of Ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16. ACM, 5 2018. doi:10.1145/3194113.3194115.

[192] Palina Tolmach. Securing smart contracts with formal verification and automated program repair, 2023. doi:10.32657/10356/169305.

[193] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A Survey of Smart Contract Formal Specification and Verification. *ACM Computing Surveys*, 54:1–38, 9 2022. doi:10.1145/3464421.

[194] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the*

2018 ACM SIGSAC conference on computer and communications security, pages 67–82, 2018. doi:10.1145/3243734.3243780.

[195] Konstantinos Tsiounis and Kostas Kontogiannis. Goal Driven Code Generation for Smart Contract Assemblies. In *Proceedings of the 2023 12th International Conference on Software and Computer Applications*, pages 112–121. ACM, 2 2023. doi:10.1145/3587828.3587846.

[196] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of software components using session types. *Fundamenta Informaticae*, 73(4):583–598, 2006. doi:10.3233/FUN-2006-73407.

[197] Andrea Vandin and Stefano Sebastio. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. In *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools*. ICST, 2014. doi:10.4108/icst.valuetools.2013.254377.

[198] Maddipati Varun, Balaji Palanisamy, and Shamik Sural. Mitigating Frontrunning Attacks in Ethereum. In *Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pages 115–124. ACM, 5 2022. doi:10.1145/3494106.3528682.

[199] Maria G. Vigliotti. What Do We Mean by Smart Contracts? Open Challenges in Smart Contracts. *Frontiers in Blockchain*, 3, 2 2021. doi:10.3389/fbloc.2020.553671.

[200] Balamurugannagarajan Vm. Simple Wallet Smart Contract, 2023. Accessed: 2025. URL: https://shorturl.at/LFsla.

[201] Fabian Vogelsteller and Vitalik Buterin. EIP-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20, 2015. Ethereum Improvement Proposal 20, November 2015.

[202] Fabian Vogelsteller and Vitalik Buterin. ERC-20 Token Standard, 2025. Accessed: 2025-05-27. URL: https://eips.ethereum.org/EIPS/eip-20.

[203] Philip Wadler. Linear Types can Change the World! In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods*, page 561. North-Holland, 1990. URL: https://www.researchgate.net/profile/Philip-Wadler/publication/2429119_Linear_Types_Can_Change_the_World/links/6410b420315dfb4cce7cf9bc/Linear-Types-Can-Change-the-World.pdf.

[204] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. Formal Specification and Verification of Smart Contracts for Azure Blockchain. *CoRR*, abs/1812.08829, 4 2019. URL: http://arxiv.org/abs/1812.08829.

[205] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014. URL: https://cryptodeep.ru/doc/paper.pdf.

[206] Gavin Wood. Polkadot: Vision for a Heterogeneous Multi-Chain Framework, 2021. Version 1.0. URL: https://www.allcryptowhitepapers.com/wp-content/uploads/2019/08/PolkaDotPaper.pdf.

[207] Jixuan Wu, Lei Xie, and Xiaoqi Li. Security Vulnerabilities in Ethereum Smart Contracts: A Systematic Analysis. *arXiv preprint arXiv:2504.05968*, 2025. doi:10.48550/arXiv.2504.05968.

[208] Xiangfan Wu, Ju Xing, and Xiaoqi Li. Exploring Vulnerabilities and Concerns in Solana Smart Contracts. *arXiv preprint arXiv:2504.07419*, 2025. URL: https://arxiv.org/abs/2504.07419.

[209] Valentin Wüstholz and Maria Christakis. Harvey: a greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409. ACM, 11 2020. doi:10.1145/3368089.3417064.

[210] Meihua Xiao, Yangping Xu, Yongtuo Zhang, Ke Yang, Sufen Yan, and Li Cen. Formal Verification of Solidity Smart Contracts via Automata Theory. *Symmetry*, 17:1275, 8 2025. doi:10.3390/sym17081275.

[211] Ziyi Xu and Xue Cheng. Are front-running HFTs harmful? *Available at SSRN 4329842*, 2023. doi:10.2139/ssrn.4329842.

[212] Zhe Yang, Meiyi Dai, and Jian Guo. Formal Modeling and Verification of Smart Contracts with Spin. *Electronics*, 11:3091, 9 2022. doi:10.3390/electronics11193091.

[213] Zheng Yang and Hang Lei. FEther: An extensible definitional interpreter for smart-contract verifications in Coq. *IEEE Access*, 7, 2019. doi:10.1109/ACCESS.2019.2905428.

[214] Jiashuo Zhang, Jiachi Chen, John Grundy, Jianbo Gao, Yanlin Wang, Ting Chen, Zhi Guan, and Zhong Chen. Automated Test Generation For Smart Contracts via On-Chain Test Case Augmentation and Migration . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1947–1959, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. doi:10.1109/ICSE55347.2025.00096.

[215] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. {TXSPECTOR}: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2775–2792, 2020.

[216] Wuqi Zhang, Lili Wei, Shing-Chi Cheung, Yepang Liu, Shuqing Li, Lu Liu, and Michael R. Lyu. Combatting Front-Running in Smart Contracts: Attack Mining, Benchmark Construction and Vulnerability Detector Evaluation. *IEEE Transactions on Software Engineering*, 49(6):3630–3646, June 2023. doi:10.1109/TSE.2023.3270117.

[217] Yuheng Zhang, Guojun Wang, Peiqiang Li, Wanyi Gu, and Houji Chen. FRACE: Front-Running Attack Classification on Ethereum using Ensemble Learning. *The Computer Journal*, 68:1137–1149, 9 2025. doi:10.1093/comjnl/bxaf027.

[218] Huijuan Zhu, Lei Yang, Liangmin Wang, and Victor S. Sheng. A Survey on Security Analysis Methods of Smart Contracts. *IEEE Transactions on Services Computing*, 17:4522–4539, 11 2024. doi:10.1109/TSC.2024.3463394.

[219] K Zipfel. The encyclopedia of smart contract attacks and vulnerabilities. 2019. URL: https://github.com/KadenZipfel/smart-contract-attack-vectors.