A Pattern-Oriented Ontology and Workflow Modeling Approach for the Sui Move Programming Language

Antonios Giatzis and Christos K. Georgiadis *

Posted Date: 28 October 2025

doi: 10.20944/preprints202510.2178.v1

Keywords: DCR graphs; design patterns; ontology engineering; smart contract security

 doi:10.20944/preprints202510.2178.v1

Article

A Pattern-Oriented Ontology and Workflow Modeling Approach for the Sui Move Programming Language

Antonios Giatzis * and Christos K. Georgiadis

University of Macedonia, Greece
* Correspondence: geor@uom.edu.gr

Abstract

Smart contracts are vulnerable to critical, design-level Business Logic Flaws (BLFs) that conventional analysis tools often fail to detect. To address this semantic gap, this study introduces and validates a novel ontological framework designed to formally model the link between high-level architectural intent and low-level code. The methodology involved constructing a multi-layered framework that integrates a comprehensive formal ontology of the Sui Move language, a library of secure design patterns, and process-aware Object-Centric Dynamic Condition Response (OC-DCR) graphs to specify expected behavior. The framework's representational adequacy was validated by demonstrating its ability to accurately and comprehensively model secure, canonical implementations of four patterns (Access Control, Circuit Breaker, Time Incentivization, and Escapability), drawn from the official Sui Framework and documentation. This validation confirms the framework provides a robust, machine-readable model that captures a diverse range of security, economic, and temporal logic found inside these official implementations. By establishing this formal descriptive model, our work provides the essential semantic foundation for a future generation of tools for pattern-aware auditing and runtime anomaly detection, contributing a comprehensive basis for engineering more verifiable and resilient decentralized applications.

Keywords: DCR graphs; design patterns; ontology engineering; smart contract security

1. Introduction

The creation and wide spread of decentralized applications (dApps) and their ability to manage high-value digital assets has driven significant innovation in fields like decentralized finance (DeFi), governance, and supply chain management [1–3]. Smart contracts [4] are self-executing programs that automate credible transactions without intermediaries, forming the programmable backbone of the decentralized applications for implementing their business logic. This achieved by providing an automated environment for their transactions to be executed and, at the same time, ensuring that these transactions adhere to the specified, predetermined business process behavior that has been hardcoded inside the smart contract. But business logic vulnerabilities arise as well [5,6], originating from the discrepancy between a smart contract's intended design and its on-chain implementation, allowing the attackers to manipulate the smart contract's business logic functionality for achieving malicious outcomes for their own benefits. Once deployed, most smart contracts are immutable and their potential vulnerabilities and flaws can lead to catastrophic and irreversible financial losses [7–9], undermining user's trust and preventing the mainstream adoption of blockchain technology [10,11].

The million dollars flash loan attack on Cetus Protocol on the Sui network [12] is an example of exploiting an application's functionality in ways that were not anticipated, where attackers exploited not a code bug, but a flaw in the protocol's economic logic. To combat these threats, a significant body of research has focused on developing automated analysis tools. Such incidents highlight the challenge that these traditional automated security tools that are used for detecting implementation errors are facing [13–16], with their main focus remaining on finding vulnerabilities at the code level but often

 doi:10.20944/preprints202510.2178.v1

fail to detect higher-level, architectural or design-level flaws that arise from incorrect or incomplete implementation of the contract's intended logic. Preventing such business vulnerabilities requires a proper formal modeling and verification of the intended business process that exist inside the smart contracts, ensuring that all possible interactions are consistent with the expected overall behavior of the system.

In this paper we introduce a comprehensive, multi-layered ontological framework for the security analysis of Sui Move smart contracts. The Sui blockchain [17], with its object-centric data model and its underlying Sui Move language's emphasis on object safety [18], represents a unique architectural formation that can be leveraged for addressing such types of vulnerabilities. Our framework is composed of three integrated layers, with the ontology acting as the formal bridge between these layers, providing the object-centric semantics that elevate individual Dynamic Condition Response (DCR) process models into an Object-Centric DCR (OC-DCR) specification:

- A formal ontology for the Sui Move language that systematically models its key components, including packages, modules, structs, functions, and access control mechanisms.
- A curated collection of secure design patterns, including Access Control, Circuit Breaker, Time Incentivization, and Escapability, formally defined as classes within the ontology.
- The mapping of each design pattern to a corresponding OC-DCR graph, which captures its expected dynamic behavior and logical dependencies in an executable format.

The foundation of our work is the argument that for building tools that can effectively explain and verify smart contracts, a semantically grounded approach that understands a contract's architectural intent is required, representing the link between high-level architectural intent and low-level code. This paper introduces and validates such a foundational framework, demonstrating its representational adequacy (through a qualitative analysis) of its ability to accurately and comprehensively model canonical implementations of Sui Move smart contracts security patterns, thus providing a robust descriptive foundation for future analytical tools.

This paper represents a foundational research, highlighting the need for higher-level abstractions that can formally capture a contract's intended design logic. A major obstacle to analysis in the overall Move ecosystem that formated our methodology is the lack of code opacity, something that also applies to a significant majority of Sui Move projects; their source code is not provided, and even when code is available, there is no guarantee it is the version that has actually been deployed on-chain [19,20]. And while extensive quantitative evaluation of diagnostic capabilities represents an important direction, such an evaluation first requires the validated semantic foundation that this paper aim to provide.

2. Background and Related Work

The foundation and motivation for the ontological modeling of the Sui Move programming language are established in this chapter. We explore the core principles of how blockchain networks operate, the critical role of smart contract security, the challenge of business logic vulnerabilities that can compromise decentralized applications, and the use of ontologies for formal reasoning. Through this analysis, the Sui Move programming language and its corresponding ontological model are placed within this research landscape, and how they can contribute to enhanced smart contract correctness.

2.1. Blockchain Networks

A blockchain network is a decentralized technology that, in combination with consensus algorithms and cryptographic methodologies, is designed to record transactions in an immutable ledger [21–23]. Two of the well-known blockchain paradigms, in which their data models dictate their functionality, are presented below:

- **Account-based model**: Ethereum and similar networks use this type of model, where balances are directly stored and updated with each transaction. The concept of accounts is used in the Ethereum network, in which the externally owned accounts (EOAs) are controlled by private keys, and the contract accounts, which are controlled by their associated code.

 doi:10.20944/preprints202510.2178.v1

- **Object-based model**: In this model, everything on-chain is represented as an object carrying properties, ownership rights, and transfer capabilities. These objects are stored inside the Sui ledger, recognized by their globally unique identifier they carry.

The above developed blockchain models influence smart contract security, scalability, and design patterns that are used. For ensuring that the balance updates are accurately reflected inside the network, the account-based model requires strict transaction sequencing, resulting in potentially limiting scalability, while the object-based approach allows unrelated transactions to be executed simultaneously through parallel transaction processing, thus limiting global bottlenecks. Thus, those who wish to use a network do so because they want to trust the network's ability to execute and record everything transparently and securely, using cryptographic methods that forms a "trustless" environment, in which the participants do not need to trust each other or a central intermediary.

2.2. Smart Contracts

Smart contracts are self-executing programs deployed on blockchain networks, with their main feature being executing automated agreements and enforcing rules, free from intermediaries' intervention. Conceptualized by Nick Szabo [4], they form the backbone of a decentralized application that translate business logic into executable code, making them enforceable by the network's protocol rather than a legal system.

Various programming languages for creating smart contracts have been developed, such as Solidity [24] (used on Ethereum and Polygon networks) and Move [18,25] (used on Aptos network, while Sui uses its Sui Move variation), providing constructs for state management, event emission, and permission handling. Equiped with these functionalities, they provide the environment for running complex business logic within a blockchain network, while maintaining transparency and security. But the safety and reliability of smart contracts depends on language design, programming paradigms, and community-driven standards. Modern approaches emphasize formal verification, comprehensive testing frameworks, and systematic validation methodologies to mitigate the risks inherent in immutable code deployment [26–30].

However, the characteristics that make smart contracts powerful also introduce and vulnerabilities, with studies showing that a significant proportion of smart contracts that have been deployed on blockchain networks contain security flaws of varying severity [8,31–35]. By such, placing trust in the correctness of the underlying code of a decentralized application, and this code has been created by humans and could potentially contain flaws that can be exploited, creates doubts among potential adopters who wish to benefit from this technology [10,11,36]. This fundamental characteristic of smart contracts, in which the built-in mechanism for security also increases the risk for vulnerabilities, underlines the non-negotiable need for achieving formal correctness and security before a contract is deployed on the blockchain.

2.3. The Sui Blockchain and Sui Move Language

The Sui blockchain takes advantage of the object-centric model, in which each asset is treated as a discrete object, with explicit ownership and mutability [17]. One major feature of the Sui network to tackle scalability problems is the parallel transaction execution. Each object within the network is treated as owned object (with a single owner) or as shared object (multiple users can access it). The Sui network has the ability to bypass the consensus mechanism and process multiple, non-conflicting transactions simultaneously, when these transactions involve shared objects. When transactions deal with shared object, the consensus mechanism is activated to ensure consistency across the network [37,38].

Sui Move is an adaptation of the Move language for use within the Sui network, allowing the direct storage of data inside the objects, passing them between transactions, and safe mutation without affecting an unrelated state [18]. This architectural difference of Sui's object-based model from Ethereum's account-based model represents a fundamental paradigm shift: while Solidity treats

 doi:10.20944/preprints202510.2178.v1

digital assets as entries locked inside a mapping structure of a smart contract, Sui Move treats them as independent, distinct entities, with explicit ownership and transfer rules [39]. Additionally, through its linear type system, the language enforces these ownership and transfer rules through the usage of the Move Prover [30,40], ensuring that digital assets maintain scarcity properties similar to physical objects. This architectural philosophy of the language allows the object (digital asset) to be transferred, involving moving the entire object from one owner to another, changing both ownership and control in a single atomic operation.

2.4. Design Patterns in Account-Based vs. Object-Centric Models

Design patterns are established solutions for secure, efficient, and maintainable code in software engineering [41]. They promote secure, efficient, and maintainable code by providing predefined, reusable templates for solving recurring problems, something that is in use in smart contract development as well. Research has identified multiple categories of smart contract design patterns that are in use for managing blockchain-specific constraints, such as access control mechanisms, code immutability, and high gas costs for computation and storage [7,42–45].

The features of the account-based model, used on a blockchain network such as the Ethereum, have led to the creation and adoption of specific design patterns for enhancing the modularity and security requirements of the network (such as the Proxy pattern used for upgradeability, or the Check-Effects-Interactions pattern used to ensure a strict order of operations) [7,46]. But in most cases, a significant burden is placed on the developer to ensure the integrity of asset management, manually implement access control rules, using general-purpose programming constructs [7,30,39,47–49]. And since these patterns are dependent on the underlying data model, while they provide flexibility, they are also the root cause of several restrictions and vulnerabilities from the wrong implementation of their logic [8,50,51].

In contrast, platforms that have adopted an object-centric approach for their blockchain network and its underlying smart contract language, have led to the creation of design patterns as well, but they are established on a different philosophy and functionality [18,52,53]. These patterns handles entities which are represented as objects (with defined ownership and state), rather than elements of a contract's storage layout, a feature that allows developers to manipulate on-chain assets more productively, constructing more expressive, reusable design patterns that can lead to enhanced contract logic.

This architectural decision of the account-based versus the object-centric model influences the entire smart contract development process [19,39,54], with this differentiation being more than just a change in syntax: critical code implementation responsibilities, such as ownership and access control, are moved from the application layer (where the majority of developer's errors are created), down to the platform layer, where rules can be enforced by the runtime itself. This not only enhances the security features of the language and consequently the blockchain network on which it operates, but also frees developers to focus on building more complex and composable applications, where the assets themselves, not the contracts that manage them, are the primary focus.

2.5. The Rise of Business Logic Flaws

Business logic vulnerabilities (or flaws) represent weaknesses in how business rules are designed and implemented within software applications [6], and in the domain of blockchain applications specifically, weaknesses in the rules and processes that control how smart contract execution and blockchain transactions behave. These flaws, in contrast to low-level coding errors, are created from gaps in the application's intended logic, such as an improper access control mechanism or an insufficiently defined order of state transitions. Recent studies highlight that business logic flaws (BLFs) happen when a smart contract's implemented behavior fails to match its intended functionality and the business rules it is supposed to enforce [55–57]. Flawed assumptions made by the developers about how the users will interact with the system, or a fundamental misunderstanding of the protocol's requirements, make business logic vulnerabilities difficult to detect, as they require a deep understanding of intended

 doi:10.20944/preprints202510.2178.v1

business rules rather than a code-level analysis [5,9,56,58,59]. Thus, a contract could be perfectly secure from a code-pattern perspective, yet still contain a critical flaw in its core business logic.

The economic consequences of such vulnerabilities are considerable [60–62], including the attack on the Cetus Protocol on the Sui network [12], where attackers exploited business logic flaws that allowed them to manipulate price curves and reserve calculations. Detecting business logic flaws creates a significant challenge for automated security tools, with over 80% of bugs remaining undetected [60], and that is because they are fundamentally a problem of semantics. This represents a gap between the high-level, often informally specified, business requirements (the intent) and the precise, formal behavior of the code implementation (the how) [5,47,57] . And bridging this semantic gap cannot be achieved by using purely code-centric analysis, but through the usage of higher-level formalisms that can capture and reason about the intended business logic itself. This is the domain in which formal ontologies and Dynamic Condition Response (DCR) graphs become valuable tools to use, offering a structured methodology to model, analyze, and verify the behavioral intent behind smart contracts.

2.6. Ontologies for Formalizing Smart Contracts

In computer and information science, an ontology is a formal, structured representation of knowledge [63]. By defining a set of concepts (classes), their properties (attributes), and the relationships between them within a specific domain, offering the semantic foundation for supporting interoperability, reasoning, and automated validation [64,65]. In the context of general programming languages, ontologies have been used for encoding grammar, constructs, and relationships for educational or code analysis purposes [66–68]. A portion of research has also been conducted on applying ontologies to the smart contract domain, either focusing on the Ethereum ecosystem [69,70] or developing platform-agnostic models that aim to capture the core elements common to all smart contracts [71], facilitating smart contract development through code generation, data validation, and semantic interoperability capabilities [72].

However, this existing body of work is fundamentally insufficient for formally modeling the Sui Move language, with the ontologies that have been developed for the Ethereum being deeply interconnected with the concepts of the account-based model, such as global state mappings or asset ownership. The architectural formalisms that define Solidity's language security and performance characteristics, the same characteristics that were built to perform in an account-based blockchain network, are not properly constructed and efficient for representing the characteristics of the Sui Move language (such as its object-centric data model that enables the creation and destruction of an object through a well-defined, strict rules).

With Sui network and its native programming language representing a paradigm shift away from the account-based model [17,18], a formalization gap is created that requires a new, tailored formal specification to capture its unique semantics. Without a dedicated ontology, a significant barrier remains for developers, auditors, and researchers seeking a deep and precise understanding of the language's semantics and safety guarantees, something essentil for validating the behavior of smart contracts throughout their lifecycle. Therefore, the purpose of this research is to address this gap by developing the first comprehensive, formal ontology specifically for the Sui Move programming language. And since there are no direct analogues in existing smart contract ontologies that can be used for the Sui Move language to sufficiently represented by them, we aim to provide a semantic foundation that will accelarate the development lifecycle of this emerging, object-centric ecosystem.

2.7. Declarative Process Modeling

Dynamic Condition Response (DCR) graphs provide a declarative, event-based process modeling formalism, and can be used as an alternative to traditional state machines, explicitly defining the rules for when activities are allowed and required to be executed [53,73]. Through their workflow, they can capture behavioral logic of events, conditions, responses, and dynamic inclusion/exclusion relationships, and at the same time, their visual nature provides a clear blueprint of how a smart contract's intended operations should be arranged and executed in a specific order.

 doi:10.20944/preprints202510.2178.v1

The research demonstrates that DCR graphs can be used for formalizing smart contract's design patterns and business logic semantics into language-independent specifications, thereby reducing the potential uncertainty created by using informal descriptions [53,74,75]. Their declarative nature makes them particularly well-suited for modeling the stateful and event-driven logic of smart contracts and by using these graphs as a base, the Object-Centric DCR (OC-DCR) methodology allows this formalism of modeling processes and state relations to be extended across multiple interacting objects, a natural fit for object-centric, smart contract languages [76]. With their research, Christfort et al. extended the DCR graphs formalism, aiming to capture object-centric behavior from event logs, through three key features: i) the creation of distinct DCR graphs, each modeling the lifecycle of an object; ii) supporting the dynamic creation of new processes by using spawn relations; iii) object-centric semantics connects process models to their respective data objects.

Discovery-based methods analyze event logs to discover a process from data (what a system is doing or has done in the past), while our methodology starts with a formal specification of secure design patterns and uses this to define what a system should be doing. And while discovery methods are useful for understanding and optimizing existing business processes, a specification-driven approach is essential for engineering verifiably correct and secure systems from the ground up, especially in an immutable blockchain environment.

Importantly, the DCR, OC-DCR and Ontology methodologies are not competing approaches but complementary layers in a comprehensive formal specification stack, which developers can utilize by creating the required graphs for specifying a business rule, and an ontology to define the terms within that rule. For those reasons, the above methodologies can be used to support high-assurance patterns and compliance checks in blockchain applications, such as the decentralized finance or the supply chain domain, where business logic correctness is paramount [77,78].

3. Constructing the Ontological Framework

The Sui Move language combines a specific object-centric type system, resource safety, and immutable on-chain semantics mechanisms. For that reason, the construction of our framework followed a specification-driven methodology aligned with the LOT (Linked Open Terms) industrial ontology engineering framework. The LOT methodology [79] provides a flexible approach to ontology development, explicitly recognizing that requirements can be specified through multiple techniques, including i) competency questions, ii) natural language statements, and iii) formal patterns. And regarding the smart contract security-critical domain, where canonical implementations provide ground truth, our approach leverages LOT's structured natural-language specification methods, by using formal pattern definitions that are validated against authoritative code, rather than use potentially flawed deployed code.

Using this methodology, the ontology was developed by applying a grammar-driven approach for its language structures, a documentation-driven approach for the Sui platform architecture, and an evidence-based pattern integration for linking DCR/OC-DCR models to canonical code implementations. All files regarding our developed framework are stored in a Github repo.[1]

3.1. Structural Foundation via Grammar-Driven Modeling

The first phase focused on creating a foundational layer that formally represents the syntax and core components of the Sui Move language. This was accomplished by following a top-down, grammar-driven ontology engineering method (LOT's specification-driven requirements phase), with the formal language grammar [80] being used as the primary source:

- **Knowledge Source**: We began building our ontology base with a systematic analysis of the official Sui Move formal grammar, which provides the ground truth for the language's structure.

[1] https://github.com/Giatzis/Sui-Move-Ontological-Framework

 doi:10.20944/preprints202510.2178.v1

- **Process**: The syntactic rules and non-terminals from the grammar were systematically translated into a corresponding hierarchy of OWL classes, with their relationships being modeled as part of a taxonomy of object and datatype properties. All the recent updates from the Move 2024 edition were also included, including Enum types and their variants.
- **Outcome**: The structural layer of the ontology was the result of this phase, and specifically, the architectural modeling of the language's components and their relationships.

3.2. Architectural Enrichment via Documentation Analysis

In the second phase, we used a documentation-driven architectural modeling methodology for security-critical platforms, aligning with LOT's natural language statement requirement specification technique. The structural foundation was enriched with concepts that are critical to the Sui architecture but are not explicit in the language grammar alone, involving the analysis of documentation for the language and the network, in order to capture runtime and ownership semantics:

- **Knowledge Source**: The official Sui and Move developer documentation was used, and an analysis was performed of the Sui Framework's source code [81–84].
- **Process**: We identified and formalized key architectural concepts that go beyond pure syntax (including the modeling of the Sui Object Model) by creating a specialized *sui:Object* class and formalizing its relationship with ownership types.
- **Outcome**: This phase produced the architectural layer, adding formal axioms to enforce rules of the Sui platform that cannot be violated or neglected (e.g., any *sui:Object* must have the key ability).

3.3. Behavioral Modeling via a Two-Phase, Evidence-Based Approach

The final phase of our model's construction involves incorporating the behavioral dimension, following LOT's formal pattern specification approach: we applied an evidence-based, pattern-integration methodology by curating abstract patterns, formalizing a library of security design patterns, and grounding them into an authoritative code. For this foundational study, our main goal is to prioritize the high-fidelity modeling of smart contract design patterns, instead of attempting a wide-scale, automated discovery of them. By manually curating the design patterns and grounding them in authoritative sources like the Sui Framework, we ensure the resulting graphical models are conceptually sound, accurately reflecting secure, idiomatic implementations, and not a potentially flawed codebase found in the wild. This provides a validated baseline that is essential before even attempting to scale the approach through automated methods:

- **Pattern Curation from Literature**: The process began with a systematic curation of high-level, language-agnostic security and business logic patterns from the academic literature and official language documentation. The objective was to identify a set of well-established patterns that represent the abstract security goals of decentralized applications. Thus, the ontology includes a library of four critical design patterns, adapting them to address the unique architectural characteristics of Sui's object-centric data model: two of these (the Access Control and Circuit Breaker), are well-established design patterns, previously documented across the academic literature [7,45,51,85,86]. The other two patterns (the Time Incentivization and Escapability) were selected based on their formal introduction in recent research as design patterns worthy of systematic analysis [51,74] combined with their demonstrated prevalence in real-world smart contracts and empirical security taxonomies.
- **Grounding in the Sui Framework**: Once a conceptual pattern was identified, the next step was to analyze the official Sui Framework and established developer resources, in order to locate its idiomatic implementation within the Sui Move language (a direct, evidence-based mapping from literature to implementation). Our decision to rely on the above resources was a deliberate methodological choice, driven by the documented lack of transparency in the wider Move ecosystem, and as research has shown, it is difficult to acquire or trust the source code for

 doi:10.20944/preprints202510.2178.v1

most of the deployed Sui contracts (until September 2024, over 75% of the top Sui Move projects had not provided their source code) [19,20]. After a pattern was identified and its implementation validated, it was then formally modeled as a DCR graph and integrated into the ontology.

3.4. Design Patterns and Framework

The proposed ontology includes a library of four design patterns, formally defined as individuals of the **sui:BusinessLogicPattern** class in the ontology's architectural layer, and were selected as representative fundamental elements of a robust smart contract design. Together, they provide a representative sample for validating the framework's ability to model a diverse range of security and logical issues:

- **Access Control**: Only authorized entities, typically the owner of a specific capability object, are allowed by this pattern to perform specific function executions, addressing the security concern of authorization, a topic frequently discussed across literature.
- **Circuit Breaker**: In response to an emergency or detected threat, this pattern provides a fail-safe mechanism, allowing the temporary suspension of critical contract functions.
- **Time Incentivization**: A mechanism that uses temporal constraints to reward or penalize actions, commonly found in vesting contracts and other DeFi protocols.
- **Escapability**: This pattern moves beyond traditional security by modelling the complex, stateful business logic of a smart contract and how it behaves over time, addressing the primary source of modern business logic flaws and vulnerabilities.

Regarding the relationship between the DCR graphs created in this phase and the resulting OC-DCR model, the graphs formalized for each pattern are DCR graphs, modeling a single, self-contained process (similar to a blueprint for a single room). The ontology provides the object-centric layer that integrates these graphs into a unified OC-DCR model (the blueprint for the entire house). This integration is achieved through the usage of the ontological properties, such as the *sui:definesLifecycleOf*, linking a specific DCR graph (a room's blueprint) to a specific *sui:Struct* (an object), thus formally defining that object's lifecycle and establishing the required object-centric semantics. Through this procedure, these graphs form the required multi-graph structure for the OC-DCR formalism of our framework. The top-down, specification-driven approach was chosen in order to overcome the code opacity of Sui Move smart contracts, differentiated from bottom-up discovery methodologies [76], in which they mine OC-DCR models from on-chain event logs rather than constructing them from curated, evidence-based design patterns.

4. The Sui Move Ontology: A Detailed Model

This section provides an analytical breakdown of our ontology, published in our GitHub repository, explaining our design decisions and reasoning for each component.

4.1. The Structural and Architectural Layer

The core elements of the Sui Move language and the Sui-specific network architecture in which it operates are formalized in this foundational layer. The primary purpose of this layer, as visualized in Figure 1 is to capture the relationships between code artifacts, their runtime representations, and the lifecycle of on-chain transactions.

4.1.1. Code Organization and Deployment Hierarchy

A clear and formal hierarchy for the needs of proper code organization is established within the ontological model, closely aligned with the native development and deployment lifecycle of the Sui ecosystem. This layer consists of three primary classes, the **sui:Package**, the **sui:Module**, and the **sui:Address**, whose relationships are defined using object properties, distinguishing the logical composition from the physical deployment.

 doi:10.20944/preprints202510.2178.v1

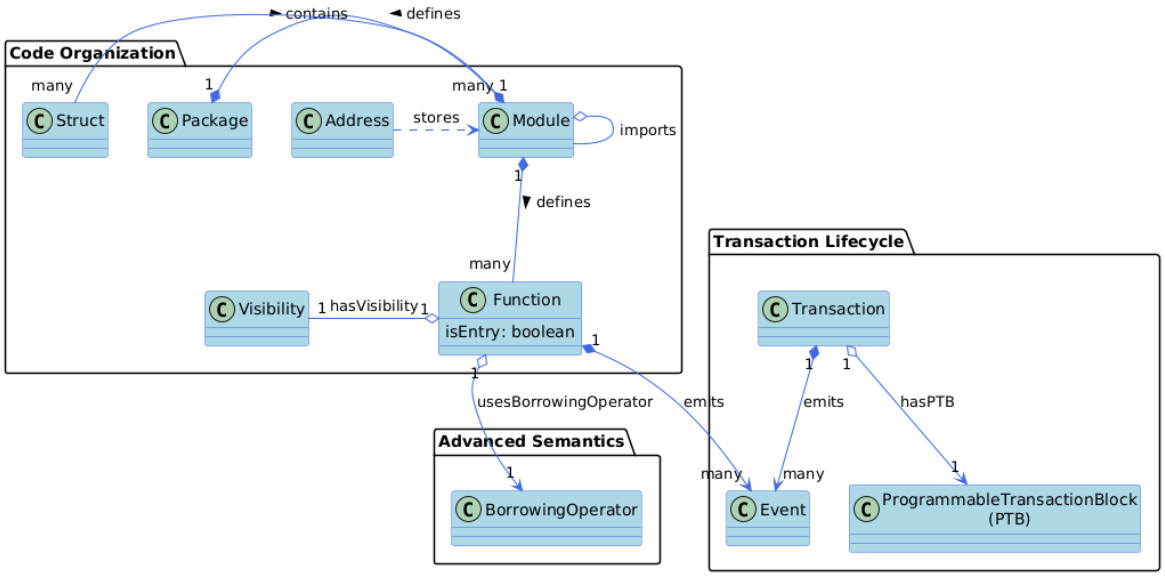

Figure 1. Architectural overview of the ontological framework and workflow modeling approach.

The fundamental unit of deployment is defined by the *sui:Package*, modeled as a container for one or more *sui:Module* instances, a relationship that is formally captured by the sui:contains object property (with its one-to-many cardinality). By such, the ontology models the pre-deployment state, in which the related modules are grouped into a single package. Once this package is published to the blockchain network, its modules are immutably stored at a specific on-chain location, with this deployed state being represented by the *sui:stores* property, linking a *sui:Address* to the *sui:Module* that is stored inside this address. Beyond being modeled as a container for one or more *sui:Module* instances, a *sui:Module* is also used as a container for the core components of a contract: a) the *sui:Functions*, that provide its executable logic and b) the *sui:Structs*, that define its data structures.

In order for our ontological model to differentiate between i) a module's role as a logical component of a software project and ii) as a physical artifact on the blockchain, the design choice of separating the *sui:contains* and *sui:stores* properties was followed. By explicitly separating these two concepts (the logical composition versus the physical deployment), the ontology facilitates a broader spectrum of analytical tasks across the entire smart contract lifecycle, such as the unveiling of the overall structure of a package, through the usage of the *sui:contains* relationship.

4.1.2. Modeling Core Language Constructs and Advanced Semantics

Moving beyond the above high-level organization, the ontology provides a granular model of function-level semantics, essential to Sui Move's security mechanisms:

- The *sui:Function* class is annotated with properties that capture its behavior and accessibility.
- The *isEntry: boolean* datatype property differentiates functions directly invoked in a transaction (where access control must be enforced) from internal helper functions, a safety measure as entry functions are the primary point of entry an attacker could try to exploit.
- Function visibility is modeled via the *sui:hasVisibility* object property, which links a *sui:Function* to an individual of the *sui:Visibility* class. The *:public*, *:publicPackage*, and *:private* instances are included inside the *sui:Visibility* class and correspond to the language's specific visibility modifiers.

What distinguishes the representation of our ontological model from a simple Abstract Syntax Tree (AST) is its explicit modeling of advanced language semantics. The *sui:BorrowingOperator* is defined as a distinct class, linked to a *sui:Function* via the *sui:usesBorrowingOperator* property. In a traditional AST, a mutable reference (&mut) would be represented as a syntactic token, but in our ontological framework, we model its use (between a function and an instance of the *sui:BorrowingOperator* class) as

 doi:10.20944/preprints202510.2178.v1

a formal relationship. With this implementation, our ontological model establishes the foundation for using possible formal queries, in order to identify semantic patterns.

4.1.3. The Transaction Lifecycle Model

The ontology provides a complete model of the on-chain execution flow, as detailed in the "Transaction Lifecycle" panel of Figure 1. The direction a transaction follows, from its creation to its on-chain deployment, is modeled with the *sui:Transaction*, *sui:ProgrammableTransactionBlock* (PTB), and *sui:Event* classes:

- A *sui:Transaction* is the unit of an execution, submitted by a user.
- The *sui:ProgrammableTransactionBlock*, which is a composable sequence of commands (e.g., function calls, object transfers), is modeled inside the ontology through the *sui:hasPTB* property, explicitly specifying that a *sui:Transaction* has exactly one *sui:ProgrammableTransactionBlock*.
- Finally, the *sui:emits* property is used for modeling the emission of events (the outcome of a transaction), linking a transaction to the *sui:Event* instances that form this outcome and is broadcasted to the blockchain network.

The ontology uses a dual-source schema to model that both sui:Function and sui:Transaction can be the source of sui:Event instances. This design choice directly reflects how the Sui runtime operates, and it is a critical element for enabling multi-level conformance checking. This two-tiered view is essential for detecting Business Logic Flaws (BLFs), such as the one exploited in the Cetus Protocol incident, in which the attacker bundled together individual operations within a single PTB and produced a malicious outcome. A simple process model might only check the transaction-level events, which might appear legitimate, but through our ontology, two distinct checks can be performed:

- **Micro-verification**: Does the sequence of events emitted by the individual *sui:Function* calls match to the low-level process model, specified for that contract?
- **Macro-verification**: Does the final set of events emitted by the *sui:Transaction* match the high-level business process rules?

With the modeled semantic hooks, the ontology provides the necessary granularity to perform two checks and detect exploits, where the logical flaw lies not in any single operation, but within their malicious composition.

4.2. Modeling Modules: Contracts vs. Libraries

Smart contract languages have the ability to separate their stateful contracts from stateless libraries. In contrast, Sui Move organizes all code into module blocks, regardless of its intended role, and for that reason, creating a separate *sui:Library* class is an incorrect representation of the language's actual structure. Instead, our ontology's semantic nature allows us to check if a module is a library or not: by simply analyzing its structure and connections, this approach provides us a more accurate model of real-world code than a syntactic classification would allow. The approach for evaluating whether a module is a library is by checking the following two conditions:

- **Statelessness**: Since a module does not define any sui:Object types (an ontological representation of a struct with the key ability), this condition recognizes those modules that do not introduce a new, on-chain, addressable state.
- **Reusability**: Other modules can target a module through numerous sui:imports relationships, suggesting that these particular module's functions are intended for shared, reusable logic.

This approach of checking a module's architectural role (based on its semantic properties) allows for an analysis that can distinguish a pure, stateless library from a module that is primarily a library but may contain a small amount of configuration state. This distinction is critical for security auditing, enabling an analyst or developer to focus on the truly stateful components of an application while still getting information about the dependency graph of its shared logic.

 doi:10.20944/preprints202510.2178.v1

4.3. The Behavioral Layer: Formalizing Processes with OC-DCR Graphs

This layer addresses the core challenge of bridging the semantic gap between the static code of a smart contract and its intended dynamic behavior. It achieves this by employing the Object-Centric Dynamic Condition Response (OC-DCR) formalism to create explicit, verifiable process models. This behavioral layer is not an isolated abstraction, but formally interconnected with the structural layer through a set of properties that serve as semantic connectors. The role of these properties is to implement the three core features of the OC-DCR formalism, transforming a collection of individual DCR graphs into a unified OC-DCR model [76]. And through this procedure, the design intent and code implementation are linked in an explicit, machine-readable way, thus performing a form of semantic reification (the process of transforming an abstract concept as a concrete entity within formal ontologies) [87,88].

Within this context, the operational logic that is embedded inside the code of a *sui:Function*, is made explicit and analyzable by reifying it as a declarative *dcr:Event*, using a formal process graph. Similarly, the way a *sui:Struct* is used across different functions creates its lifecycle, with our model reifying this lifecycle as an explicit *dcr:Process*.

Figure 2 highlights the three object properties that establish the connection between the Process Models and the Code Constructs:

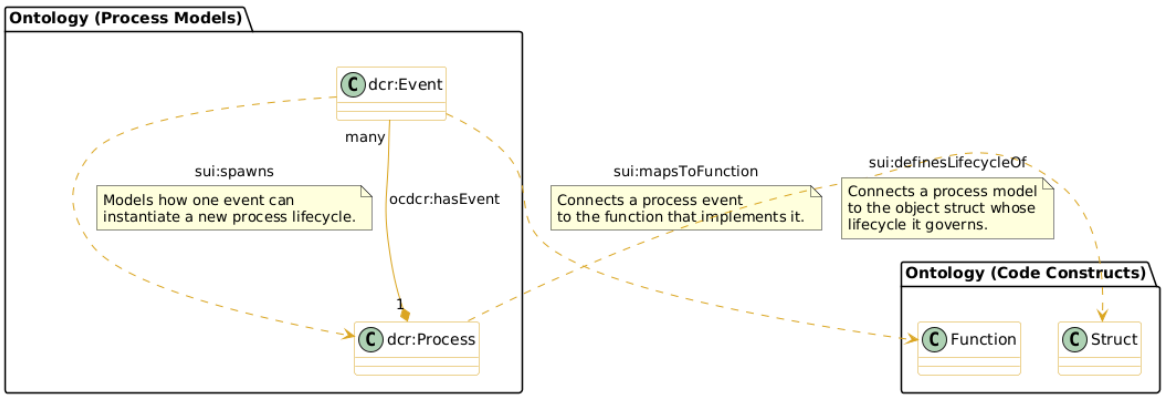

Figure 2. Behavioral Layer.

- **sui:mapsToFunction:** This property creates a direct link between an event in the process model (*dcr:Event*) and the specific *sui:Function* in the code that is responsible for implementing its logic.
- **sui:definesLifecycleOf:** This property connects an entire process model (*dcr:Process*) to the *sui:Struct*. This is the core element of our ontological model regarding its object-centric design, with the DCR graphs representing the entire set of valid state transitions an object can have.
- **sui:spawns:** The dynamic process instantiation is modeled through this property, in which the execution of one *dcr:Event* leads to the creation of a new instance of a *dcr:Process* (e.g., a new object with its own lifecycle).

Without these three object properties that provides the object-centric semantics and spawn relations, the created DCR graphs would remain an illustrative diagram, with no verifiable way to guarantee that the code faithfully implements the rules that the graphs enforce. By formally stating an axiom, we create a testable assertion that can be consumed by a static analysis tool designed to parse the body of the selected function and verify that the specified event semantics are correctly implemented.

4.3.1. Modeling Object Lifecycles

An OC-DCR graph can represent the entire lifecycle of an object type, between its creation and destruction, capturing all operations that are permitted to be executed in an ordered sequence. To model this, two distinct yet complementary representations of an object's state are introduced inside the ontology:

 doi:10.20944/preprints202510.2178.v1

- **sui:OwnershipType**: This class (the "what") is created from the Sui Move language's type system, fundamental for ensuring resource safety. Four primary instances are included (:AddressOwned, :Shared, :Immutable, and :Wrapped).
- **sui:ObjectLifecycleState**: This class (the "how") is a process-oriented representation of the state, including instances that model an object's status within a specific business process, such as :Created, :Owned, :Transferred, and :Deleted.

The following Table 1 summarizes this distinction:

Table 1. Object Lifecycles Representation.

Aspect	sui:OwnershipType	sui:ObjectLifecycleState
Models	Resource safety and memory management.	Object's status within a business process
Scope	High-level	Aplication-specific
Examples	:AddressOwned, :Shared	:Created, :Transferred
Enforced By	Sui Move Compiler and Runtime	OC-DCR Process Model and Application Logic

This formal distinction is crucial for process-aware analysis: Throughout a series of transactions, an object's ownership type may remain static as :AddressOwned, but its lifecycle state within the business process can change, progressing from :Created, to :Owned, to :Transferred. A process tool must be able to verify that these state transitions occur in the correct order and under the correct conditions, something that could not have been done if only the static *sui:OwnershipType* was taken into account.

The Cetus Protocol exploit [12] was not a violation of Sui Move's type safety, but rather a bypass of its implicit process logic. In simpler terms, the code was correct, but the design was flawed. Our model, by linking a *dcr:Process* to a struct through the *sui:definesLifecycleOf* property, can formally specify this business logic, enabling the detection of BLFs that the type system has not captured.

4.4. The Semantic Layer: The Dual-Layer Pattern Library

Our framework introduces a two-layered model that distinguishes the abstract security goals from the specific code patterns that are used to achieve them. This separation allows a clear analysis, by separating the conceptual "why" (the architectural intent) with the platform-specific "how" (the implementation technique), thus creating a distinction between them (Figure 3).

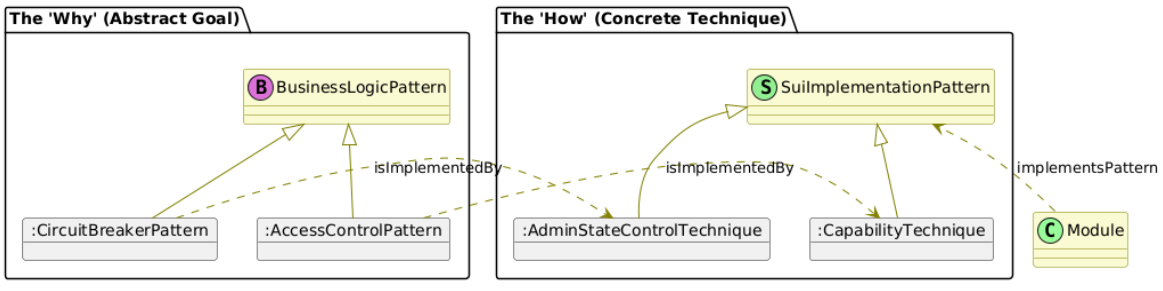

Figure 3. Semantic Layer.

Two new classes are modeled to implement this:

- **sui:BusinessLogicPattern**: This class represents the high-level, language-agnostic concept (the "why"). Instances of this class (e.g. :AccessControlPattern) represents established security design patterns.

 doi:10.20944/preprints202510.2178.v1

- **sui:SuiImplementationPattern**: his class represents the low-level, idiomatic Sui Move technique used to realize an abstract goal (the "how"). Instances, such as the :CapabilityTechnique, are identified by analyzing authoritative codebases like the Sui Framework [81].

For connecting the above hierarchies to each other and to the implemented code, two properties are used, creating a complete semantic chain:

- First, the *sui:isImplementedBy* property links high-level patterns to the technique that is used for implementation. For example, to connect a high-level pattern to its low-level technique, the statement :CircuitBreakerPattern, sui:isImplementedBy, :AdminStateControlTechnique is used.
- Second, the *sui:implementsPattern* property links a specific code module to the implementation technique. The statement validator_module, sui:implementsPattern, :AdminStateControlTechnique for example, creates a direct link between the code and the pattern.

4.5. The Verification Layer

To support the smart contract development lifecycle, our ontology models vital components of the Move Prover [30,40]. Through its usage, developers can prove the correctness of their Move code against a set of formal specifications. For supporting the analysis of this verification process, the ontology provides a structured representation of these components, enabling a form of meta-analysis, focused on a project's overall verification coverage and quality.

The core element of this layer (Figure 4) is the **sui:Specification** class, corresponding to a spec block in source code, with each block containing formal logical assertions about the code's behavior. The *sui:specifies* object property creates the crucial link between an instance of *sui:Specification* and the code element it annotates. The range of this layer is the combination of the *sui:Module*, *sui:Function*, and *sui:Struct* classes, showing that specifications can be applied at different levels of granularity.

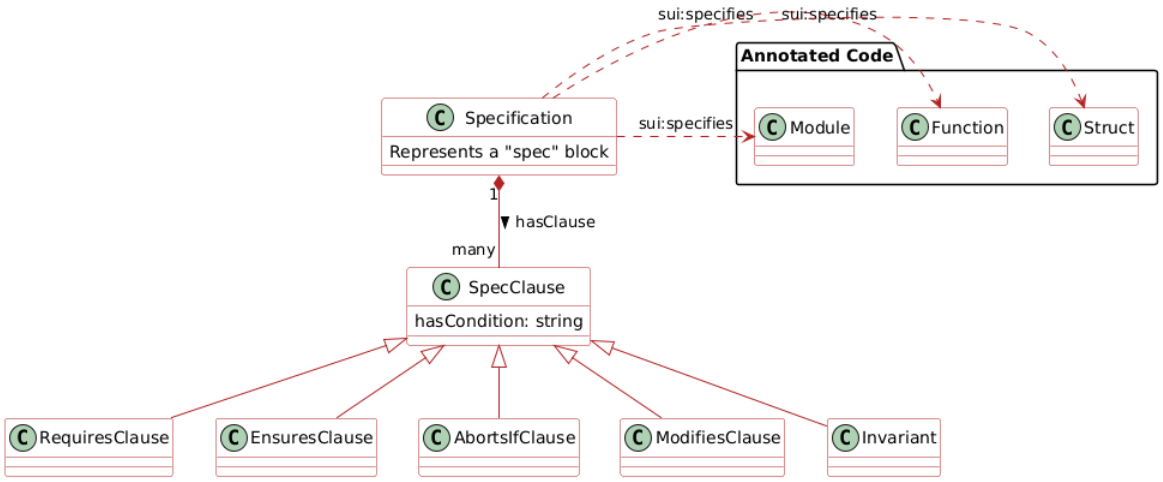

Figure 4. Verification Layer.

The ontology also models the internal structure of these specification blocks, with each *sui:Specification* linked (through the hasClause class) to one or more instances of *sui:SpecClause*. And this class serves as the parent for a detailed hierarchy of specific clause types, allowing the ontology to capture not just that a function is specified, but how it is specified. Each clause type represents a different kind of formal assertion, including:

- *sui:RequiresClause* for function preconditions,
- *sui:EnsuresClause* for postconditions,
- *sui:AbortsIfClause* for defining failure conditions,
- *sui:ModifiesClause* for specifying which memory locations a function is allowed to change,
- *sui:Invariant* for defining state properties that must hold true across all operations.

 doi:10.20944/preprints202510.2178.v1

A key design decision within this layer concerns the modeling of the logical content within each clause. For the *sui:SpecClause* class, the ontology defines the *hasCondition:* property, a design decision within this layer concerning the modeling of the logical content for each clause. This means that the logical expression inside a clause (e.g., balance > 0) is stored as a literal string, a required and intentional level of abstraction to the layer's purpose. And that, because the goal of this layer is not to replicate the complex functionality of the Move Prover itself, a difficult endeavor in itself, and out of the scope of this article.

By modeling the type and target of specification clauses, the ontology creates a foundation for a new form of automated auditing procedure, focused on "verification health" or "specification coverage". An automated auditing tool, built upon this ontological foundation, that could execute queries against a contract's design, could help developers build robust and verifiably secure smart contracts. And through this design, our approach shifts the focus from the question: "Is this formal proof correct?", to a more equally important question, such as: "Is this code elements sufficiently specified to begin with?".

5. Validating the Framework

Following LOT methodology regarding evidence-based specification grounding, we performed an intermediate validation step between logical consistency (Section 5.1) and behavioral validation (Sections 5.2-5.4). Our aim was to validate whether our framework's components (the ontology, pattern library, and OC-DCR models) could accurately represent canonical, secure implementations of the four design patterns of the Sui Move language, instead of arbitrary formalisms.

5.1. Foundational Ontology Validation

The logical integrity of the proposed ontology (ensuring it is free of internal contradictions) was formally verified, using the integrated HermiT reasoner (version 1.4.3) of the Protégé Desktop environment (version 5.6.2). The reasoner successfully processed the entire ontology, computing all class, property, and instance inferences, reporting zero errors, thus no inconsistencies. To further confirm its logical integrity, the Protégé debugger was used as well, providing us with an extra confirmation that our ontology is both coherent and consistent. This formal verification ensures that the ontology's axioms do not lead to logical contradictions and that all defined classes are satisfiable.

5.2. DCR Pattern Model Validation

The workflow we followed for validating each of the four patterns (as described in the 3.4 subchapter), moving from formal specification to a visual model, is the following:

1. **Selection of a Canonical Implementation:** The identification of an implementation for each pattern was made from the Sui ecosystem, a critical methodological choice for ensuring our models are based on a verified codebase that works as the ground truth. The used codebases are the exact versions used at the time of evaluation and stored within our Github project repo:
    - **The Official Sui Framework:** We used the transfer_policy.move (Access Control), coin.move (Circuit Breaker), and package.move (Escapability patterns) modules, representing core codebases that define the network's foundational logic.
    - **The Official Sui Developer Documentation:** For the Time Incentivization pattern, we used the linear.move example, representing community-accepted best practices.
2. **Mapping to the Ontology:** We manually mapped the selected Sui Move code to the corresponding classes and properties in our ontological model, by creating instances that represent the specific code elements and defining the relationships between them (e.g., the transfer policy module implements the Access Control Technique).
3. **Process Analysis:** We analyzed the control flow and logic of the implementation to verify that the real-world code's behavior aligns with the behavioral rules defined in the corresponding OC-DCR model.

 doi:10.20944/preprints202510.2178.v1

4. **Executable Model Validation:** The ontology utilizes four primary DCR relations, with each DCR relation being mapped to a unique and consistent visual symbol (e.g., color and style of an arrow) provided from the online tool [2] that was found in the literature [53,73,74,89], and their semantics are defined as follows (Table 2):

Table 2. Visual Modeling.

Expression	Events Relations	Arrows Colors
B dcr:condition A	A must happen before B	Orange
A dcr:response B	If A happens, B becomes a required future action	Blue
A dcr:exclusion B	If A happens, B is permanently disabled	Red
A dcr:inclusion B	If A happens, it enables B and is available to happen	Green

5. **Graphs Testing:** The final step of the validation involves using an online simulation-assisted tool in order to test the created visual DCR graphs. By executing both the valid ("happy path") and invalid ("attack path") scenarios, we verify that the behavior of the executable visual model corresponds to the rules defined by the ontology.

5.3. Internal Validation

The internal validity of each DCR graph was confirmed through rigorous logical simulation, using the online tool, with the main objective to verify that the operational semantics of the created visual models are in alignment with the logical constraints that are defined in our ontology. The methodology involved executing events in the simulation engine to observe the state of the process model. For example, there are three events in the Access Control pattern: the ACGrantRole (creates a capability), the ACProtectedCall (a protected operation), and the ACRevokeRole (destroys the capability). Through the tool-assisted validation procedure, we validate two scenarios: i) We confirm that authorized access works correctly, in which by executing the ACGrantRole event, the ACProtectedCall is also activated; ii) We also confirm that the revocation functionality works, in which, after we provide access (executing the ACGrantRole event), we execute the ACRevokeRole event that triggers the dcrexclusion rule, permanently disabling the ACProtectedCall functionality. In this way, revocation works by removing the ability to execute the protected ACProtectedCall.

The same methodology was applied for validating all four patterns, and in each step, we verified that the set of enabled, disabled, and pending activities strictly followed our placed ontological rules, confirming a direct correspondence between the formal specifications and the executable graphs.

1. **Access Control Pattern:** The simulation confirmed that access is dependent on a revocable capability.
   - **Happy Path:** Executing AC_GrantRole enabled AC_ProtectedCall, validating the dcr:condition rule.
   - **Attack Path:** After executing AC_GrantRole, executing AC_RevokeRole disabled AC_ProtectedCall, validating the dcr:exclusion rule. The three-event model (AC_GrantRole, AC_ProtectedCall, AC_RevokeRole) was validated to ensure the revocation capability correctly disables protected operations, confirming the dcr:exclusion semantics.
2. **Circuit Breaker Pattern:** The simulation validated the model's ability to halt and resume key operations based on all specified rules.

[2] https://www.dcrgraphs.net/

 doi:10.20944/preprints202510.2178.v1

- **Happy Path:** Executing CB_Pause disabled CB_OperationalCall and CB_Pause itself, while enabling CB_Unpause, validating the dcr:exclusion and dcr:inclusion rules.
- **Attack Path:** With the process in the "paused" state, any attempt to execute CB_OperationalCall was blocked by the simulator.

3. **Time Incentivization Pattern:** The simulation confirmed the time-dependent logic central to the pattern.
   - **Happy Path:** Executing TI_Start enabled TI_Proceed and flagged it as a pending response, validating the dcr:response rule.
   - **Attack Path:** An attempt to execute TI_Proceed before TI_Start was blocked, validating the dcr:condition rule.
4. **Escapability Pattern:** The simulation validated the one-way authorization flow required for an escape hatch mechanism.
   - **Happy Path:** Executing ES_Authorize enabled ES_Escape, validating the dcr:condition rule.
   - **Attack Path:** An attempt to execute ES_Escape from the initial state was blocked, validating the dcr:condition.

5.4. Qualitative External Validation

The validation results for each of the four patterns are presented in this section, demonstrating the ability of the framework to model them, using authoritative codebases from the Sui ecosystem. The mappings were formally encoded as instances and property assertions within the ontology:

1. **Access Control Pattern**
   - **Canonical Implementation:** The pattern has been found to be implemented into the transfer_policy.move Sui Framework's codebase, allowing the creation of rules that restrict how certain assets can be transferred.
   - **Ontological Mapping:** The TransferPolicy struct is mapped to the StateObject class, and the TransferPolicyCap struct (the one that grants administrative rights) is mapped to CapabilityObject. The pattern's functions are then mapped to specific access control events:
     - The new() function maps to the AC_GrantRole event.
     - The protected withdraw() function maps to the AC_ProtectedCall event.
     - The destroy_and_withdraw() function maps to the AC_RevokeRole event.
   - **Process Analysis:** The implementation's logic aligns with our DCR graph:
     - The AC_ProtectedCall (withdraw) requires the TransferPolicyCap to be created by AC_GrantRole (new), satisfying the dcr:condition.
     - The feature to destroy the capability via calling the destroy_and_withdraw() function (AC_RevokeRole), satisfies the dcr:exclusion rule and ensures that access, once granted, is also verifiably revocable.
2. **Circuit Breaker Pattern**
   - **Canonical Implementation:** The pattern has been found to be implemented into the coin.move Sui Framework's codebase, implementing the DenyCapV2 mechanism for providing built-in circuit breaker capabilities.
   - **Ontological Mapping:** This pattern's capabilities inside the above codebase are defined by several key structs and functions:
     - The TreasuryCap<T> struct (for minting/burning) is mapped to CapabilityObject.
     - The DenyCapV2<T> struct (for pause operations) is mapped to the CircuitBreakerTechnique class.
     - The deny_list_v2_enable_global_pause() function maps to the CB_Pause event, while the deny_list_v2_disable_global_pause() maps to CB_Unpause.
     - Protected operations like mint() and burn() map to the CB_OperationalCall event.

 doi:10.20944/preprints202510.2178.v1

- **Process Analysis:**The implementation is a dual-layer control mechanism, aligning with our DCR graph. Initially, the CB_OperationalCall event is permitted, but with the execution of deny_list_v2_enable_global_pause() function (CB_Pause in our ontological mapping), the system transit to a paused state, thus satisfying two key conditions:
  - Implementing the dcr:exclusion rule disables the CB_OperationalCall.
  - The CB_Pause event enables the CB_Unpause event (similar to our dcr:inclusion), allowing the system to be restored.

3. **Time Incentivization Pattern**
   - **Canonical Implementation:** The pattern has been found to be implemented into the linear.move codebase from the Sui Vesting Framework, which defines a linear token vesting schedule to incentivize long-term holding.
   - **Ontological Mapping:** The key components of the Time Incentivization pattern are mapped to our ontology as detailed below:
     - The state of the pattern that includes the vesting schedule is held in the Wallet struct, and mapped to the StateObject class.
     - The claimable() function, which calculates available tokens, is mapped to the TI_Start event.
     - The claim() function, which allows withdrawal, is mapped to the TI_Proceed event.
     - The vesting completion state is mapped to the TI_Timeout event.
   - **Process Analysis:**
     - The TI_Proceed (claim) event enforces the dcr:condition by yielding tokens proportional to the time elapsed since TI_Start.
     - The linear formula (self.balance.value() * elapsed) / self.duration) calculates a yield proportional to the holding period, thus rewarding long-term holding.
4. **Escapability Pattern**
   - **Canonical Implementation:** The pattern has been found to be implemented into the package.move Sui Framework's codebase.
   - **Ontological Mapping:** The UpgradeCap struct, which grants upgrade authority, is the CapabilityObject. The authorize_upgrade() function maps to the ES_Authorize event, and the commit_upgrade() function maps to the ES_Escape event.
   - **Process Analysis:** The control flow follows our DCR graph's rules. The ES_Escape action (commit_upgrade) requires the prior completion of the ES_Authorize function call (authorize_upgrade). Thereby, the dcr:condition relationship is satisfied, ensuring that contract upgrades are deliberate and authorized.

6. Limitations and Future Work

The primary contribution of this paper is the design and validation of a foundational ontological framework, and that is why our evaluation methodology centered on demonstrating its capability to accurately model canonical implementations of known security design patterns. But this limitation of this foundational study creates a significant direction for future work: evaluating the framework's ability to detect faulty implementations. The absence of quantitative simulation metrics reflects a deliberate research focus rather than a methodological oversight. With that in mind, our main goal was to provide the semantic foundation for the development and empirical evaluation of a pattern-aware static analysis tool for such a task, a tool that represents a substantial research effort designated as the next phase of this research.

The design patterns that are included in the ontological framework, while justified as representative solutions that were found in the academic literature to address security vulnerabilities, are not exhaustive. Validating a sample of four patterns, while sufficient to demonstrate representational adequacy for this foundational study, still represents a small sample size, and future work should

 doi:10.20944/preprints202510.2178.v1

expand this library to cover a wider array of patterns. Similarly, the target language and platform that our model is based on are often upgraded, resulting in the need for possible upgrades into the core ontology, in order to reflect the new features or syntax changes, thus representing a standard maintenance consideration.

The modeling process we followed relies on manual curation of patterns from the official sources, and while this provides a high degree of accuracy, it does not easily scale to discover novel or less common patterns from a large corpus of contracts. This was a deliberate trade-off in our methodology, focusing on prioritizing depth and correctness for our foundational model rather than creating a fully automated discovery process that is still in an early stage due to the lack of code opacity.

Based on these findings, several opportunities that could use our work as a baseline remain open for future research, in order to enhance the understanding and practice of secure and efficient smart contract development:

1. A promising future research direction is the formalization of "pattern compositions" as first-class concepts, allowing tools to reason about how patterns interfere with, or reinforce one another, within a single module-contract.
2. The ontological framework could provide developers with real-time, semantically-aware feedback through its integration into Integrated Development Environments (IDEs). For instance, an IDE could recognize when a developer is implementing a vesting schedule and guide them to ensure that its logic correctly matches the Time Incentivization pattern.
3. The ontology's Verification layer could be used and leveraged for building tools that i) could perform an analysis of a project's verification structure (e.g. automatic identification of functions that lack certain blocks) thus pinpoint unverified components or ii) by upgrading it in order to create an automated procedure of auditing the formal specifications themselves, and not just checking the presence of a spec clause (moving from describing code and its specifications to analyzing and validating the relationship between them).

7. Conclusions

A multi-layered ontological framework was created and validated in this paper for the Sui Move programming language. It is designed to bridge the semantic gap between high-level architectural intent and low-level code in Sui Move smart contracts, addressing the challenge of detecting Business Logic Flaws (BLFs) inside smart contracts, which are not often detected by code-centric analysis tools.

By creating and leveraging a formal ontology as an "object-centric layer," we showed that it is possible to systematically integrate a collection of individual, pattern-specific DCR graphs into a single, connected OC-DCR model. A set of explicit ontological properties that formally link abstract behavioral models to concrete code artifacts were used in order to satisfy the defining features of the OC-DCR formalism. The framework's representational adequacy was confirmed through a qualitative validation against canonical implementations of four security design patterns, used from authoritative Sui Framework and documentation sources. This process verified our model's capacity to accurately capture a range of security, economic, and temporal logic.

Through this robust, machine-readable model of intended behavior, this research provides the essential semantic foundation toward engineering more verifiable and resilient decentralized applications, offering a structured approach to ensure that smart contract implementations faithfully follow their intended design.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Adisa, O.; Ilugbusi, B.S.; Obi, O.C.; Awonuga, K.F.; Adelekan, O.A.; Asuzu, O.F.; Ndubuisi, N.L. Decentralized Finance (DEFI) in the US economy: A review: Assessing the rise, challenges, and implications of blockchain-driven financial systems. *World Journal of Advanced Research and Reviews* **2024**, *21*, 2313–2328.

 doi:10.20944/preprints202510.2178.v1

2. Vazquez Melendez, E.I.; Bergey, P.; Smith, B. Blockchain technology for supply chain provenance: increasing supply chain efficiency and consumer trust. *Supply chain management: An international journal* **2024**, *29*, 706–730.
3. Zhu, G.; He, D.; An, H.; Luo, M.; Peng, C. The governance technology for blockchain systems: a survey. *Frontiers of Computer Science* **2024**, *18*, 182813.
4. Szabo, N. Formalizing and securing relationships on public networks.
5. Eshghie, M.; Artho, C.; Stammler, H.; Ahrendt, W.; Hildebrandt, T.; Schneider, G. Highguard: Cross-chain business logic monitoring of smart contracts. In Proceedings of the Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 2378–2381.
6. Metin, B.; Wynn, M.; Tunalı, A.; Kepir, Y. Business Logic Vulnerabilities in the Digital Era: A Detection Framework Using Artificial Intelligence. *Information* **2025**, *16*, 585.
7. Kannengiesser, N.; Lins, S.; Sander, C.; Winter, K.; Frey, H.; Sunyaev, A. Challenges and common solutions in smart contract development. *IEEE Transactions on Software Engineering* **2021**, *48*, 4291–4318.
8. Kushwaha, S.S.; Joshi, S.; Singh, D.; Kaur, M.; Lee, H.N. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *Ieee Access* **2022**, *10*, 6605–6621.
9. Mishra, D.; Phansalkar, S. Blockchain Security in Focus: A Comprehensive Investigation into Threats, Smart Contract Security, Cross-Chain Bridges, Vulnerabilities Detection Tools & Techniques. *IEEE Access* **2025**.
10. Alazab, M.; Alhyari, S.; Awajan, A.; Abdallah, A.B. Blockchain technology in supply chain management: an empirical study of the factors affecting user adoption/acceptance. *Cluster Computing* **2021**, *24*, 83–101.
11. Gao, S.; Li, Y. An empirical study on the adoption of blockchain-based games from users' perspectives. *The Electronic Library* **2021**, *39*, 596–614.
12. Peng, Z.; Yin, X.; Ying, C.; Ni, C.; Luo, Y. A Preference-Driven Methodology for High-Quality Solidity Code Generation. *arXiv preprint arXiv:2506.03006* **2025**.
13. Ghaleb, A.; Pattabiraman, K. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In Proceedings of the Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, 2020, pp. 415–427.
14. Durieux, T.; Ferreira, J.F.; Abreu, R.; Cruz, P. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In Proceedings of the Proceedings of the ACM/IEEE 42nd International conference on software engineering, 2020, pp. 530–541.
15. Piantadosi, V.; Rosa, G.; Placella, D.; Scalabrino, S.; Oliveto, R. Detecting functional and security-related issues in smart contracts: A systematic literature review. *Software: Practice and Experience* **2023**, *53*, 465–495.
16. Chaliasos, S.; Charalambous, M.A.; Zhou, L.; Galanopoulou, R.; Gervais, A.; Mitropoulos, D.; Livshits, B. Smart contract and defi security tools: Do they meet the needs of practitioners? In Proceedings of the Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.
17. Mysten Labs. The Sui Smart Contracts Platform. https://docs.sui.io/paper/sui.pdf, 2022. Accessed: 2025-09-19.
18. Welc, A.; Blackshear, S. Sui move: Modern blockchain programming with objects. In Proceedings of the Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, 2023, pp. 53–55.
19. Van Tonder, R. Verifying and displaying move smart contract source code for the sui blockchain. In Proceedings of the Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, 2024, pp. 26–29.
20. Chen, E.; Tang, X.; Xiao, Z.; Li, C.; Li, S.; Wu, T.; Wang, S.; Chalkias, K.K. SuiGPT MAD: Move AI Decompiler to Improve Transparency and Auditability on Non-Open-Source Blockchain Smart Contract. In Proceedings of the Proceedings of the ACM on Web Conference 2025, 2025, pp. 1567–1576.
21. Wenhua, Z.; Qamar, F.; Abdali, T.A.N.; Hassan, R.; Jafri, S.T.A.; Nguyen, Q.N. Blockchain technology: security issues, healthcare applications, challenges and future trends. *Electronics* **2023**, *12*, 546.
22. Wijesekara, P.A.D.S.N.; Gunawardena, S. A review of blockchain technology in knowledge-defined networking, its application, benefits, and challenges. *Network* **2023**, *3*, 343–421.
23. Kavita, S.; Shinde, S. A comprehensive survey of consensus protocols, challenges, and attacks of blockchain network. In Proceedings of the 2024 Fourth International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT). IEEE, 2024, pp. 1–6.
24. Wang, Z.; Chen, X.; Zhou, X.; Huang, Y.; Zheng, Z.; Wu, J. An empirical study of solidity language features. In Proceedings of the 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 2021, pp. 698–707.

 doi:10.20944/preprints202510.2178.v1

25. Patrignani, M.; Blackshear, S. Robust safety for move. In Proceedings of the 2023 IEEE 36th Computer Security Foundations Symposium (CSF). IEEE, 2023, pp. 308–323.
26. Garfatta, I.; Klai, K.; Gaaloul, W.; Graiet, M. A survey on formal verification for solidity smart contracts. In Proceedings of the Proceedings of the 2021 Australasian Computer Science Week Multiconference, 2021, pp. 1–10.
27. Tolmach, P.; Li, Y.; Lin, S.W.; Liu, Y.; Li, Z. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)* **2021**, *54*, 1–38.
28. Chen, K.; Luo, J.; Lu, Y.; Zhang, Z.; Zhang, W.; Wang, X.; Li, P.; Zhao, J. The Formal Verification of Aptos Coin. In Proceedings of the International Conference on Information Security. Springer, 2024, pp. 3–22.
29. Zhu, H.; Yang, L.; Wang, L.; Sheng, V.S. A survey on security analysis methods of smart contracts. *IEEE Transactions on Services Computing* **2024**.
30. Bartoletti, M.; Crafa, S.; Lipparini, E. Formal verification in Solidity and Move: insights from a comparative analysis. *arXiv preprint arXiv:2502.13929* **2025**.
31. Perez, D.; Livshits, B. Smart contract vulnerabilities: Does anyone care. *arXiv preprint arXiv:1902.06710* **2019**, pp. 1–15.
32. Tang, X.; Zhou, K.; Cheng, J.; Li, H.; Yuan, Y. The vulnerabilities in smart contracts: A survey. In Proceedings of the International Conference on Artificial Intelligence and Security. Springer, 2021, pp. 177–190.
33. Zhou, H.; Milani Fard, A.; Makanju, A. The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. *Journal of Cybersecurity and Privacy* **2022**, *2*, 358–378.
34. He, D.; Wu, R.; Li, X.; Chan, S.; Guizani, M. Detection of vulnerabilities of blockchain smart contracts. *IEEE Internet of Things Journal* **2023**, *10*, 12178–12185.
35. Hejazi, N.; Lashkari, A.H. A comprehensive survey of smart contracts vulnerability detection tools: Techniques and methodologies. *Journal of Network and Computer Applications* **2025**, p. 104142.
36. Tan, T.M.; Saraniemi, S. Trust in blockchain-enabled exchanges: Future directions in blockchain marketing. *Journal of the Academy of marketing Science* **2023**, *51*, 914–939.
37. Babel, K.; Chursin, A.; Danezis, G.; Kichidis, A.; Kokoris-Kogias, L.; Koshy, A.; Sonnino, A.; Tian, M. Mysticeti: Reaching the limits of latency with uncertified dags. arXiv 2023. *arXiv preprint arXiv:2310.14821*.
38. Danezis, G.; Kokoris-Kogias, L.; Sonnino, A.; Spiegelman, A. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In Proceedings of the Proceedings of the Seventeenth European Conference on Computer Systems, 2022, pp. 34–50.
39. Giatzis, A.; Papangelou, S.; Georgiadis, C.K. A Comparative Study of Solidity and Sui Move: Advancing Smart Contract Development. *IEEE Access* **2025**.
40. Zhong, J.E.; Cheang, K.; Qadeer, S.; Grieskamp, W.; Blackshear, S.; Park, J.; Zohar, Y.; Barrett, C.; Dill, D.L. The move prover. In Proceedings of the International Conference on Computer Aided Verification. Springer, 2020, pp. 137–150.
41. Wedyan, F.; Abufakher, S. Impact of design patterns on software quality: a systematic literature review. *IET Software* **2020**, *14*, 1–17.
42. Hu, B.; Zhang, Z.; Liu, J.; Liu, Y.; Yin, J.; Lu, R.; Lin, X. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. *Patterns* **2021**, *2*.
43. Di Sorbo, A.; Laudanna, S.; Vacca, A.; Visaggio, C.A.; Canfora, G. Profiling gas consumption in solidity smart contracts. *Journal of Systems and Software* **2022**, *186*, 111193.
44. Huang, M.; Chen, J.; Jiang, Z.; Zheng, Z. Revealing hidden threats: An empirical study of library misuse in smart contracts. In Proceedings of the Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–12.
45. Azimi, S.; Golzari, A.; Ivaki, N.; Laranjeiro, N. A systematic review on smart contracts security design patterns. *Empirical Software Engineering* **2025**, *30*, 1–40.
46. Ebrahimi, A.M.; Adams, B.; Oliva, G.A.; Hassan, A.E. A large-scale exploratory study on the proxy pattern in ethereum. *Empirical Software Engineering* **2024**, *29*, 81.
47. Zou, W.; Lo, D.; Kochhar, P.S.; Le, X.B.D.; Xia, X.; Feng, Y.; Chen, Z.; Xu, B. Smart contract development: Challenges and opportunities. *IEEE transactions on software engineering* **2019**, *47*, 2084–2106.
48. Olivieri, L.; Arceri, V.; Chachar, B.; Negrini, L.; Tagliaferro, F.; Spoto, F.; Ferrara, P.; Cortesi, A. General-purpose Languages for Blockchain Smart Contracts Development: A Comprenhensive Study. *IEEE Access* **2024**.

 doi:10.20944/preprints202510.2178.v1

49. Salzano, F.; Marchesi, L.; Antenucci, C.K.; Scalabrino, S.; Tonelli, R.; Oliveto, R.; Pareschi, R. Bridging the Gap: A Comparative Study of Academic and Developer Approaches to Smart Contract Vulnerabilities. *arXiv preprint arXiv:2504.12443* **2025**.
50. Liao, Z.; Hao, S.; Nan, Y.; Zheng, Z. Smartstate: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis. In Proceedings of the Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 980–991.
51. Ghaleb, A.; Rubin, J.; Pattabiraman, K. Achecker: Statically detecting smart contract access control vulnerabilities. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023.
52. Majd, N.E.; Hinojosa, A.; Fisher, C.; Landeros, F.; Baldimtsi, F. An Analytical Performance Evaluation on Sui Move Object-Centric Models. In Proceedings of the 2025 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). IEEE, 2025, pp. 1–3.
53. Xu, Y.; Slaats, T.; Düdder, B.; Troels Hildebrandt, T.; Van Cutsem, T. Safe design and evolution of smart contracts using dynamic condition response graphs to model generic role-based behaviors. *Journal of Software: Evolution and Process* **2025**, *37*, e2730.
54. Giuffrida, S.; Salim, S.; Ullah, A.; Vaccargiu, M. A Move Sui library for secure, certified and trusted supply chain ownership management. In Proceedings of the 2025 IEEE/ACM 7th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). IEEE, 2025, pp. 50–56.
55. Singh, A.; Parizi, R.M.; Zhang, Q.; Choo, K.K.R.; Dehghantanha, A. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security* **2020**, *88*, 101654.
56. Sun, Y.; Wu, D.; Xue, Y.; Liu, H.; Wang, H.; Xu, Z.; Xie, X.; Liu, Y. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In Proceedings of the Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
57. Lin, X.; Xie, Q.; Zhao, B.; Tian, Y.; Zonouz, S.; Ruan, N.; Li, J.; Beyah, R.; Ji, S. PROMFUZZ: Leveraging LLM-Driven and Bug-Oriented Composite Analysis for Detecting Functional Bugs in Smart Contracts. *arXiv e-prints* **2025**, pp. arXiv–2503.
58. Rameder, H.; Di Angelo, M.; Salzer, G. Review of automated vulnerability analysis of smart contracts on ethereum. *Frontiers in Blockchain* **2022**, *5*, 814977.
59. Soud, M.; Nuutinen, W.; Liebel, G. Soley: Identification and automated detection of logic vulnerabilities in ethereum smart contracts using large language models. *arXiv preprint arXiv:2406.16244* **2024**.
60. Zhang, Z.; Zhang, B.; Xu, W.; Lin, Z. Demystifying exploitable bugs in smart contracts. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 615–627.
61. Zhou, L.; Xiong, X.; Ernstberger, J.; Chaliasos, S.; Wang, Z.; Wang, Y.; Qin, K.; Wattenhofer, R.; Song, D.; Gervais, A. Sok: Decentralized finance (defi) attacks. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP). IEEE, 2023, pp. 2444–2461.
62. Carpentier-Desjardins, C.; Paquet-Clouston, M.; Kitzler, S.; Haslhofer, B. Mapping the DeFi crime landscape: an evidence-based picture. *Journal of Cybersecurity* **2025**, *11*, tyae029.
63. Studer, R.; Benjamins, V.R.; Fensel, D. Knowledge engineering: Principles and methods. *Data & knowledge engineering* **1998**, *25*, 161–197.
64. Van Heijst, G.; Schreiber, A.T.; Wielinga, B.J. Using explicit ontologies in KBS development. *International journal of human-computer studies* **1997**, *46*, 183–292.
65. Noy, N.F.; McGuinness, D.L.; et al. Ontology development 101: A guide to creating your first ontology, 2001.
66. Strmečki, D.; Magdalenić, I.; Kermek, D. An Overview on the use of Ontologies in Software Engineering. *Journal of computer science* **2016**, *12*, 597–610.
67. Diatta, B.; Basse, A.; Ouya, S. PasOnto: Ontology for learning Pascal programming language. In Proceedings of the 2019 IEEE Global Engineering Education Conference (EDUCON). IEEE, 2019, pp. 749–754.
68. Nongkhai, L.N.; Wang, J.; Mendori, T. Developing an Ontology of Multiple Programming Languages from the Perspective of Computational Thinking Education. *International Association for Development of the Information Society* **2022**.
69. Bella, G.; Cantone, D.; Longo, C.; Nicolosi Asmundo, M.; Santamaria, D.F. Blockchains through ontologies: the case study of the Ethereum ERC721 standard in OASIS. In Proceedings of the International Symposium on Intelligent and Distributed Computing. Springer, 2021, pp. 249–259.
70. Cano-Benito, J.; Cimmino, A.; García-Castro, R. Toward the ontological modeling of smart contracts: a solidity use case. *IEEE Access* **2021**, *9*, 140156–140172.
71. Van Woensel, W.; Seneviratne, O. Semantic Interoperability on Blockchain by Generating Smart Contracts Based on Knowledge Graphs. *arXiv preprint arXiv:2409.12171* **2024**. Accessed: 2025-10-19.

 doi:10.20944/preprints202510.2178.v1

72. Dominguez, J.A.; Gonnet, S.; Vegetti, M. The role of ontologies in smart contracts: A systematic literature review. *Journal of Industrial Information Integration* **2024**, *40*, 100630.
73. Awad, A.; Awaysheh, F.; López, H.A. BEST: A Unified Business Process Enactment via Streams and Tables for Service Computing. *arXiv preprint arXiv:2501.14848* **2025**.
74. Eshghie, M.; Ahrendt, W.; Artho, C.; Hildebrandt, T.T.; Schneider, G. Capturing smart contract design with dcr graphs. In Proceedings of the International Conference on Software Engineering and Formal Methods. Springer, 2023, pp. 106–125.
75. Xu, Y.; Slaats, T.; Dudder, B.; Hildebrandt, T.T. Adding generic role-and process-based behaviors to smart contracts using dynamic condition response graphs. In Proceedings of the 2023 IEEE/ACM International Conference on Software and System Processes (ICSSP). IEEE Computer Society, 2023, pp. 70–80.
76. Christfort, A.K.; Rivkin, A.; Fahland, D.; Hildebrandt, T.T.; Slaats, T. Discovery of object-centric declarative models. In Proceedings of the 2024 6th International Conference on Process Mining (ICPM). IEEE, 2024, pp. 121–128.
77. Garfatta, I.; Klai, K.; Gaaloul, W. Integrating Business Process Context into Solidity-to-CPN Formal Verification. In Proceedings of the 2024 32nd International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). IEEE, 2024, pp. 68–73.
78. Bartoletti, M.; Lipparini, E.; Pompianu, L. LLMs as verification oracles for Solidity. *arXiv preprint arXiv:2509.19153* **2025**.
79. Poveda-Villalón, M.; Fernández-Izquierdo, A.; Fernández-López, M.; García-Castro, R. LOT: An industrial oriented ontology engineering framework. *Engineering Applications of Artificial Intelligence* **2022**, *111*, 104755.
80. Labs, M. Move Language Tree-sitter Grammar. https://github.com/MystenLabs/sui/blob/main/external-crates/move/tooling/tree-sitter/src/grammar.json, 2025. Accessed: 2025-07-11.
81. Labs, M. Sui: A Next-Generation Smart Contract Platform. https://github.com/MystenLabs/sui. Accessed: 2025-07-19.
82. Move Book Contributors. The Move Book: A Guide to the Move Programming Language and Sui Blockchain. https://move-book.com/. Accessed: 2025-07-19.
83. Labs, M. Sui Documentation. https://docs.sui.io/. Accessed: 2025-07-19.
84. Labs, M. Introduction to the Sui Book. https://intro.sui-book.com/. Accessed: 2025-07-19.
85. Six, N.; Herbaut, N.; Salinesi, C. Blockchain software patterns for the design of decentralized applications: A systematic literature review. *Blockchain: Research and Applications* **2022**, *3*, 100061.
86. Seneviratne, O. The Feasibility of a Smart Contract" Kill Switch". In Proceedings of the 2024 6th International Conference on Blockchain Computing and Applications (BCCA). IEEE, 2024, pp. 473–480.
87. Guarino, N.; Sales, T.P.; Guizzardi, G. Reification and truthmaking patterns. In Proceedings of the International Conference on Conceptual Modeling. Springer, 2018, pp. 151–165.
88. Guizzardi, G.; Guarino, N. Explanation, semantics, and ontology. *Data & Knowledge Engineering* **2024**, *153*, 102325.
89. Debois, S.; Hildebrandt, T.T.; Marquard, M.; Slaats, T. The DCR Graphs Process Portal. In Proceedings of the BPM (Demos), 2016, pp. 7–11.