frontiers | Frontiers in Blockchain

TYPE Methods
PUBLISHED 08 January 2024
DOI 10.3389/fbloc.2023.1276233



OPEN ACCESS

EDITED BY
Marcelle Michelle Georgina Maria Von Wendland,
Bancstreet Capital Partners Ltd., United Kingdom

REVIEWED BY
Marino Miculan,
University of Udine, Italy
Muhammad Asghar Khan,
Hamdard University, Pakistan

*CORRESPONDENCE
Iqra Mustafa,
iqra.mustafa@mycit.ie

†These authors have contributed equally to this work

RECEIVED 11 August 2023
ACCEPTED 29 November 2023
PUBLISHED 08 January 2024

CITATION
Mustafa I, McGibney A and Rea S (2024), Smart contract life-cycle management: an engineering framework for the generation of robust and verifiable smart contracts.
*Front. Blockchain* 6:1276233.
doi: 10.3389/fbloc.2023.1276233



# Smart contract life-cycle management: an engineering framework for the generation of robust and verifiable smart contracts

Iqra Mustafa*†, Alan McGibney† and Susan Rea†

Nimbus Research Centre, Munster Technological University, Cork, Ireland

The concept of smart contracts (SCs) is becoming more prevalent, and their application is gaining traction across many diverse scenarios. However, producing poorly constructed contracts carries significant risks, including the potential for substantial financial loss, a lack of trust in the technology, and the risk of exposure to cyber-attacks. Several tools exist to assist in developing SCs, but their limited functionality increases development complexity. Expert knowledge is required to ensure contract reliability, resilience, and scalability. To overcome these risks and challenges, tools and services based on modeling and formal techniques are required that offer a robust methodology for SC verification and life-cycle management. This study proposes an engineering framework for the generation of a robust and verifiable smart contract (GRV-SC) framework that covers the entire SC life-cycle from design to deployment stages. It adopts SC modeling and automated formal verification methodologies to detect security vulnerabilities and improve resilience, extensibility, and code optimization to mitigate risks associated with SC development. Initially, the framework includes the implementation of a formal approach, using colored Petri nets (CPNs), to model cross-platform Digital Asset Modeling Language (DAML) SCs. It also incorporates a specialized type safety dynamic verifier, which is designed to detect and address new vulnerabilities that can arise in DAML contracts, such as access control and insecure direct object reference (Idor) vulnerabilities. The proposed GRV-SC framework provides a holistic approach to SC life-cycle management and aims to enhance the security, reliability, and adoption of SCs.

KEYWORDS

**formal verification, smart contracts, model-driven engineering, knowledge base graphs, DAML model**

## 1 Introduction

Blockchain, as a distributed platform, allows for the deployment of a software code called smart contracts (SCs) that can be used to create next-generation decentralized applications across different industrial sectors and stakeholders. SCs are executable pieces of code that reside on a blockchain network to automate digital workflows by containing self-executing business logic. Many currently deployed SCs handle a large amount of virtual currency worth millions in fiat currency, making the monetary incentives easily high enough to attract

adversaries and cyber-criminals. Therefore, SCs are subjected to external threats and attacks, just like any other software program (Luu et al., 2016a; Magazzeni et al., 2017).

A minor mistake in the definition and coding of the SC logic can introduce security vulnerabilities that can be exploited and potentially incur significant economic loss (millions $) or penalties. For instance, arithmetic bugs, exceptions, re-entrancy, and flash loan attacks are some examples of SC vulnerabilities (Kaur, 2023). To address this, numerous tools have been developed to detect vulnerabilities. According to a study reported in Luu et al. (2016b), in which Oyente, a symbolic execution tool ran on 19,366 SCs from the first 1,460,000 blocks in the Ethereum network, resulted in 8,833 SCs, with at least one security vulnerability. However, 340 SCs were found to have a re-entrancy handling vulnerability. This vulnerability resulted in one of the highest-profile hacks on SCs, the decentralized autonomous organization (DAO) attack deployed on Ethereum. The hackers exploited this vulnerability in which $70 million worth of ether (ETH) was siphoned off from BC (Popper, 2016). Moreover, in 2017, the Bithumb attack was deployed by breaching the South Korean Bithumb cryptocurrency exchange system, and the attacker was able to steal 32,000 users' data and money. Cumulatively, all these attacks happened due to poor programming practices.

Additionally, in designing and deploying secure and reliable contracts, the SC language selection is, in fact, an integral step that best addresses the individual specific business needs (Dwivedi et al., 2021). Therefore, when choosing an SC language, programmers should take into account the following aspects: security, usability, accessibility to tools and libraries, community support, scalability, and interoperability.

However, errors can still occur at any point across the following three phases of a SC life-cycle: modeling, pre-deployment, and network-deployment stages. Delmolino (2016) and Atzei et al. (2017) have documented particular execution weaknesses exploited by attacks described above in Solidity-based contracts that utilize the Ethereum execution environment. However, these methods are not automated, and a developer must manually check their program against each of the identified vulnerabilities as part of the creation process. As the reach of SCs expands to new application domains such as the Internet of Things (IoT), enterprise, maritime, cloud, artificial intelligence (AI), and medical field, the need for reliability and confidence in contract execution is coming to the forefront (Huynh-The et al., 2023; Kordestani et al., 2023; Wang et al., 2023). This has prompted the research community to investigate topics such as the analysis and reporting of SC vulnerabilities and bugs, auditing standards, identification of security strategies, model-driven engineering, and a need to verify domain-specific properties.

One of the promising approaches to address these challenges is the use of formal verification methods. Formal verification using formal methods for specifying, designing, and verifying programs has been a long-proven method to ensure the correctness of safety-critical systems. SCs are ideally suited for comprehensive formal verification as they are compact and time-bounded (Lin et al., 2022). From an academic perspective, several tools and approaches have, therefore, arisen to support the development of secure and robust SCs and to assist in the analysis of already-deployed contracts (Kaur, 2023; Silviu, 2023). This research includes approaches that use non-formal methods (static and dynamic analysis methods) to detect failures under specific execution circumstances and other methods based on formal techniques with the aim of automatic formal verification (FV) of SCs. Since non-formal methods can only test a particular request under specific scenarios, they cannot prove the correctness of SCs in general, making detecting complex patterns challenging.

Likewise, machine learning (ML)-based technologies such as sequence learning, minimum intermediate representation learning, supervised ML, and deep learning models (Tann, 2018; Liao, 2019; Momeni et al., 2019; Lesimple and Martin, 2020; Li et al., 2020; Wang et al., 2020) have been applied to detect vulnerabilities in SCs. These technologies leverage the power of ML algorithms to analyze codes and identify potential security issues. However, these existing tools and techniques target only a small percentage of vulnerabilities compared to the number of reported vulnerabilities in Silviu (2023); Dingman et al. (2019). In addition, some approaches significantly rely on a set of expert-defined rules/patterns, which can be error-prone and result in significant false-positive or false-negative errors. For this reason, recent works have explored formal verification, which has proven to be effective in achieving such correctness objectives, although it is very costly and harder to automate (Singh et al., 2020). Much of the research is focused on one aspect, and currently, limited approaches incorporate verification of SCs across the entire life cycle (specification, design, testing, and deployment). This can be attributed to the complexity and broad range of causes (e.g., poor coding and logical errors) for potential vulnerabilities as such further research is required to integrate automated modeling and formal verification techniques across each step in the development of SCs.

The majority of existing verification solutions are platform-dependent and language-specific; for example, Ethereum and Hyperledger Fabric are two separate platforms with distinct applications and security requirements[1]. As a result, the approach used to solve Ethereum SC issues will not work on Fabric (Zheng et al., 2020). Following a detailed review of the literature, it is our view that "the potential risk associated with poorly performing and insecure decentralized applications (dApps) will become a primary barrier to their uptake, and the absence of a complete framework for managing the entire contract life-cycle development only exacerbates the existence of this barrier."

SCs have unique properties that require a structured and controlled approach to their development and deployment. Hence, all stages of the contract creation and deployment process, including design, coding, testing, verification, deployment, and maintenance, should be incorporated by such a framework. Furthermore, the framework needs to be adaptable enough to support various SC applications and offer a single-stop solution for managing the contract's complete lifespan. Without a holistic approach to life-cycle management, SCs may be designed and deployed in an *ad hoc* manner, which can lead to security and reliability issues.

Hence, this study proposes an initial step toward addressing these challenges by developing an iterative verification mechanism

1 https://101blockchains.com/ethereum-vs-hyperledger-fabric/

for enterprise SCs to increase trust in blockchain applications, namely, the generation of robust and verifiable smart contract (GRV-SC) framework. The GRV-SC framework consists of several stages that cover the entire SC life cycle. These stages include requirement specification, modeling, verification, pre-deployment testing, network deployment testing, and execution and completeness using SC modeling and automated formal verification approaches. By following this comprehensive framework, developers can ensure that their SCs are robust, secure, and resilient throughout their entire life cycle. Based on the aforementioned needs, this study provides the following contributions:

- Exploring the motives and requirements to facilitate a formal SC life-cycle approach and how their absence can restrict contract security, reliability, robustness, and adoption.
- Conceptualizing a holistic life-cycle management approach for the GRV-SC framework.
- Initial implementation of a formal approach for cross-platform Digital Asset Modeling Language (DAML) SCs, i.e., colored Petri nets (CPNs) for DAML to address new vulnerabilities.
- Code generation engine that offers translation from the CPN model to secure DAML templates.
- Implementation of a dedicated type safety dynamic verifier for detecting DAML vulnerabilities.

The remainder of the paper is structured as follows: Section 2 provides a review of the current state-of-the-art with respect to SC modeling and verification. Section 3 offers an overview of the proposed GRV-SC framework. Section 4 covers the methodology of the proposed framework. Section 5 provides experimental results along with a case study. Section 6 draws conclusions with respect to the proposed work and identifies the next stage in the continuation of this work.

# 2 Research problem

In this section, we shed light on the existing research gap that poses a challenge to the SC ecosystem in terms of enhancing its security and robustness.

## 2.1 Requirements

The development life cycle for SC is typically divided into three phases: creation, deployment, execution, and completeness (Jani, 2020):

1) *Smart contract creation:* Generally, after multi-party negotiation, software developers define and convert the natural language contract specification to a computer program using platform-specific programming languages including but not limited to Solidity, Go, Kotlin, Java, or C++.
2) *Deployment:* The developed contract is then deployed across the blockchain network and is accessible to all parties involved. Any change to the contract post-deployment will require the creation of a new contract as a consequence of the immutable nature of blockchain systems.
3) *Execution and completeness:* Contractual conditions have been evaluated, following the deployment of SCs. Once these predefined conditions are met, the functions will be executed automatically. Following the execution of SC, all parties involved are updated with new states (i.e., written to the ledger). As a result, the blockchain keeps track of both the transactions and modified states during the SC execution, for example, resulting in digital assets being transferred from one party to another without the need for any human intervention.

The three key characteristics of SCs, i.e., automation, decentralization, and trustlessness, make the development extremely challenging due to a high degree of variability (for instance, the level of uncertainty associated with contractual agreements can be quite high; additionally, the programming language and platforms can also change frequently). Therefore, creating a SC is often just confined to converting a natural language agreement into a programming language. To address this complexity, it is proposed to expand the creation stage to create a formal design and validation process to minimize the possibility of poorly constructed SCs, as shown in Figure 1. It is proposed to introduce two essential steps in the SC life cycle process:

- First, the need to formalize the definition of SC logic utilizing model-based techniques.
- Second, formal verification and ML mechanisms are used to auto-validate and verify the robustness of the SC code at the pre-deployment and network-deployment phases.

By including these two steps, developers can embed more robust controls and create SCs that are less error-prone, more secure, and reliable.

SC design based on the use of a modeling mechanism is of utmost importance in order to verify the correctness prior to deployment, and the validation of the contract model aims to avoid deploying vulnerable contracts on-chain. This can be done using a variety of formal verification methods such as model-checking (Nehai et al., 2018), theorem-proving (Sen et al., 2017; Bhargavan, 2016; Amani, 2018), or Petri nets (Zupan, 2020; Liu and Liu, 2019; Duo et al., 2020). These methods use mathematical reasoning and logic to prove the functional correctness of the underlying SC model. This phase ensures contract security and privacy at the pre-deployment stage. The contract can then be subsequently deployed on test networks to ensure that it performs as expected on real ledger networks. Therefore, in SC life-cycle management, design (contract modeling), validation, and deployment should be an iterative process to ensure a high-quality code. To support this process, it is essential to have access to developer tools and frameworks that can automate and streamline key process steps. By adopting this systematic approach, developers can ensure that their SCs are secure, reliable, and trustworthy. This can help promote the long-term success of blockchain-based applications and transactions, which enhances the overall stability and growth of the blockchain ecosystem.

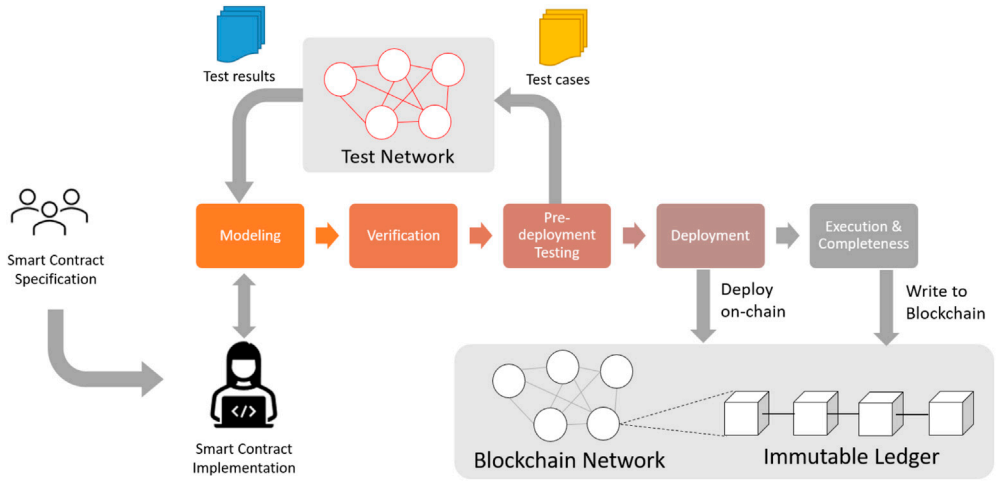


FIGURE 1
Extended smart contract development life cycle.

## 2.2 Related work

This section will cover the current literature in these two key areas, modeling and formal verification.

### 2.2.1 Smart contract modeling

Various engineering approaches have been developed and utilized to ensure the reliability of software systems. These include, for example, code only, code visualization, round-trip engineering, model-driven, and model-centric approaches. SCs that form part of a decentralized application (DApp) can be complex and difficult to understand, especially for non-specialist programmers. They embed business rules that are transparent when programmed; however, understanding the flow and contract itself is non-trivial due to the decentralized deployment pattern. To address this challenge, SC engineering must shift focus from a code-centric to a model-centric approach known as model-driven engineering (MDE). MDE not only reduces the coding complexity but it also promotes knowledge re-usability. The model-driven architecture (MDA), an example of MDE, comprises three models: computationally independent, platform-independent, and platform-specific models. These models can abstract the contract implementation process by offering different levels of behavioral, functional, and technical details (Boogaard, 2018). Moreover, the adoption of a model-centric or MDE approach aids in reducing the number of errors, increases the quality of the code, and empowers developers, particularly when working collaboratively to develop the solution. A model-centric approach for SC development creates a higher level of abstraction that allows practitioners to focus on the business problems rather than the technology required to implement the solution. This should lead to reusable platform-independent models and improve the development efficiency across blockchain ecosystems. Modeling also allows for more emphasis on security and error mitigation prior to deployment on-chain. Although the engineering approaches for SCs are still maturing, a code-centric approach is most commonly used; however, there have been several techniques developed to enable the transition toward a model-centric approach. Boogaard (2018) provided an overview of the common approaches utilized for SC modeling. This review can be extended to include additional approaches that have also been utilized including system modeling language (SysML) (Zupan, 2020), unified modeling language (UML) (Garamvölgyi, 2018), and Digital Asset Modeling Language (DAML) (DigitalAsset, 2019).

Zupan (2020) presented a modular architecture that utilizes SysML activity diagrams to facilitate an expert-friendly approach to review SC workflows. This allows for a sophisticated security review process within the framework before deploying SC on the BC network. The authors aim to leverage Petri net (PN) to support formal verification such that they implement a translation from SysML to Petri net modeling language (PNML), which can be used to verify the contract logic and auto-generate the SC code. The outcome is the generation of a secure SC template, and this still requires a developer to add codes manually, which, in turn, requires expert review and a testing process.

The UML macro programming approach has been explored by Peter et al. (Garamvölgyi, 2018) for generating SCs from UML state charts, in the domain of cyber–physical systems (CPS). Model elements and Solidity SC constructs generally have a direct mapping. The proposed study is incomplete, and the mapping does not follow a common operational semantics-based technique and as such is only partially automated; therefore, the development phase is still manually executed. Furthermore, it is domain-centric and only provides an abstract view of functionality (inner workings of code), which is a crucial demand for robust SC design.

To reduce the need for manual coding, Mavridou and Laszka (2017) have used a platform-independent model (PIM), namely, a finite state machine (FSM) approach for designing SCs. In this method, SCs act as state machines where a SC is in the initial state and a particular temporal transaction or external input is responsible for the SC transition from one state to the next state. This approach reduces the semantic gap by providing a formal model with clear semantics that offers capabilities to connect to formal analysis tools. Ultimately, the code generator feature helps developers to

implement a contract with minimal manual coding effort. However, the FSM transformation to Solidity is only partially automated, requiring manual coding to ensure the quality of the contract code. In addition, the properties that cannot be modeled in FSM can be added by plugins that aim to implement patterns and fix vulnerabilities. The disadvantage is that FSM is a complex model compared with other modeling approaches; in addition, objects and roles are not explicitly defined in this method. Due to this, it is difficult to manage the development of complex contracts without any design ideas. However, by using patterns, known vulnerabilities can be countered and these limitations can also be addressed.

Business Process Modeling Notation (BPMN) provides a graphical representation model for the specification of a business process. A number of works have utilized this approach as the basis for defining the SC logic. Frans et al. (Panduwinata and Yugopuspito, 2019) provided an example of mapped BPMN using three Hyperledger Composer concepts, namely, assets, participants, and transactions based on microservices for a reservation-based parking system. Similarly, Weber et al. (2016) have utilized BPMN as a PIM for supply chain use cases to address trust in collaborative business processes (CBPs). In both approaches, BPMN is translated into a Solidity code. The linear business process makes this approach the best fit for supply chain use cases but does not align well for recursive contracts.

Agent-based modeling (ABM) uses a structural natural language approach. Frantz and Nowostawski (2016) used ABM to decompose a SC model into rule-based statements that are then compiled into a structured formalization, leveraging a grammar of institutions [as introduced by CRAWFORD and Elinor (2005)].

In ABM, processes are modeled as dynamic systems that provide insights into the interactions and behavior of agents modeled by a set of statements (Van Dyke Parunak et al., 1998). In this methodology, the statement is composed of five components, abbreviated as ADICO. "A" stands for the "attribute" (properties of the agent), "D" stands for "deontic" (statement nature as permission, compulsion, and exclusion), "I" stands for "aim" (an action that governs the proposition), "C" denotes "conditions" (which are contextual conditions under which the proposition holds), and "O" denotes "or else" (describes the meanings associated with non-compliance with the statement). These five components can be used to describe the execution of a SC, linking the institutional functions using logical operators to generate rule sets. By modeling the relationships between agents in a rule-based manner, the ABM approach helps in detecting dependencies among the activities of users/participants in a system. The set of prescriptions is then transformed into a SC skeleton that can be completed manually by the developer. A domain-specific language (DSL) was utilized that allows the mapping of instructions to the SC skeleton. It has been argued that the SC skeleton requirements separate the specification task from implementation, thereby ensuring that no critical functionality is overlooked. Furthermore, it was suggested that institutional grammar facilitates the SC development process for non-technical users. However, in comparison to an easily readable structural natural language, this model holds high-level programming language characteristics, and the outcome of the conversion demands manual coding. The deontic ADICO component makes the behavioral aspect of SC explicit as it indicates what a participant should or should not do; however, it does not explicitly state how this component needs to be monitored or implemented.

For private-permissioned BCs, DAML goes beyond a SC modeling language; it is a complete platform for building full-stack applications. SCs can work with many distributed ledger technologies (DLTs) and databases, such as VMware, Hyperledger Sawtooth, Amazon Aurora, and Amazon QLDB. In this context, the key difference between DAML and any other platform is its sub-transaction privacy, which ensures that each participant involved in a transaction only sees the parts of the transaction that they are authorized to see. On the surface, the DAML language allows users to specify which parties view which parts of a transaction, and the Canton protocol allows the SC to plug into the consensus layer of different distributed ledgers and databases while preserving privacy guarantees. This is particularly relevant for financial institutions and others who require transaction confidentiality (DigitalAsset, 2019).

The language, in particular, adds a layer of security by including privacy and authentication as language features. The following are some of the characteristics of DAML:

1) It is an open-source functional language with Haskell-like syntax.
2) It provides a means to write contracts and automates some functions so developers can focus on the business logic.
3) Usable in a private execution environment, the information in contracts written in DAML is only accessible to authorized parties.
4) Being human and machine-readable, it provides an opportunity for non-experts to focus on business needs rather than coding complexities.

In summary, each modeling technique serves a distinct purpose: some are appropriate for specifying business workflows, such as BPMN, while others are suitable for visualization or manual inspection of SCs, and some are effective for formulating the requirements of contracts. BPMN, UML, and SysML offer modeling approaches that are generic in nature, but their semantics may not be well-established. To validate these modeling approaches, formal models such as PN/CPN and Behavior, Interaction, and Priority (BIP) frameworks are commonly used.

Hu et al. (2020) have discussed eight attributes that a SC should satisfy, i.e., "*legality*, *probativeness*, *consistency, customizability*, *observability*, *verifiability, self-enforceability*, *and access-controlling*." *Legality* refers to a code's legitimacy in accordance with the basic rules of a contractual agreement. This includes the consideration of legal, technical, and business aspects, as well as the ownership and control of assets involved in the contract. *Probativeness* means the secure storage of input data and event outcomes that can be used as evidence in court. *Consistency* means that legitimate/legal authorities should analyze the contract before publishing it to ensure that it does not conflict with existing rules and regulations. *Customizability* is about a contract tailored to meet the needs of the parties involved. A complex or intricate contract can be made by combining multiple simple contracts. *Observability* necessitates the use of interfaces to monitor the state of contracts, including the contract itself, its performance, and anything else related to it. *Verifiability* is about the verification of

contract execution logic. *Self-enforceability* is to guard the contract against breaches and third parties using cryptographic keys, and *access-controlling* means that only authorized individuals have access to contract information such as knowledge, control, and performance.

All of the above-mentioned attributes discussed by the authors are important considerations during the design of a SC. Each attribute contributes to the overall quality and reliability of a SC, and neglecting any of them can lead to problems and failures in contract execution. However, it should be noted that addressing all these aspects adds significant complexity. Therefore, the research focuses on a subset of these attributes to provide a practical approach to designing and verifying SCs that can be effectively implemented in real-world scenarios. This includes legality, consistency, customizability, and most importantly verifiability.

Enterprise-focused cross-platform languages like DAML inherit some of the characteristics described above as part of contract modeling, such as observability, self-enforceability, and access-controlling, which, in turn, reduces the modeling complexity of contract for formal verification (verifiability). However, the use of other modeling approaches, such as SysML and BPMN, involves an additional step of model transformation [e.g., convert to Solidity or chaincode such as *SysML* → *PN* → *SCtemplate* (Zupan, 2020), *UML* → *SCtemplate* (Garamvölgyi, 2018), *BPS* → *FactoryContract* (Weber et al., 2016; Panduwinata and Yugopuspito, 2019), and *nADICO-statements* → *SCskeleton* (Frantz and Nowostawski, 2016)] to ensure that the generated contract inherits the aforementioned attributes. DAML is proposed to look beyond language-specific vulnerabilities, reduce modeling complexity, and the risk of logical vulnerabilities to make SC development easier, suggesting its advantage over other techniques.

### 2.2.2 Smart contract verification

To minimize the risks associated with designing and deploying SCs, it is critical to ensure that SC properties are verified against design specifications and to identify any potential flaws that could have a detrimental impact on the DApp once deployed on-chain. For this reason, numerous works have focused on SC privacy and security assurance using different techniques such as static and dynamic analyses, ML, and formal methods, and these are briefly discussed in the following sections.

(A) ***Static and dynamic analyses for verification:*** Static analysis techniques are used to examine the SC code in a non-runtime environment. SC tools based on static analysis techniques identify vulnerability patterns, errors, and also assess the code behavior that is expected at runtime. Static code analysis is run on the i) source code or ii) bytecode. A static code analyzer will check all the possible paths of execution for specific (expected) vulnerabilities using a control flow graph (CFG). Therefore, to run the static analyzer, the source code must be parsed using lexer tools, which results in an abstract syntax tree (AST). After a CFG is extracted from the AST, a static analyzer runs different analyses based on different vulnerability types, i.e., symbolic and taint analysis. Symbolic execution includes reasoning in relation to code behavior for different inputs. Taint analysis identifies the flow of user input through a system to understand the security implications of the system design. There is no general static analysis tool as each code analyzer is dedicated to a particular set of vulnerability types. As such, to generalize an approach, the typical approach is used to combine multiple analyzers, which takes time and resources due to the extensive pre-processing usually required. Some commonly used static analysis tools include Zeus, Oyente, Vandal, and Securify (Brent, 2018; Kalra, 2018; Luu et al., 2018; Tsankov, 2018). Dynamic analysis techniques examine the SC code in a runtime environment. Although both approaches have their advantages and limitations, one should not be considered being preferable to the other. Static analysis will uncover vulnerabilities at the design phase and can identify risks that would not be detected during runtime. Dynamic analysis, on the other hand, is a functional assessment during runtime that is used to expose security risks but is restricted to the code that is executed during runtime. The most widely used dynamic analysis tools in relation to SCs are ContractFuzzer (Jiang et al., 2018) and MAIAN (Nikolić, 2018).

Many static and dynamic testing tools have been developed so far, with the majority of these being built for Ethereum SC testing. The usability of these testing tools and frameworks varies considerably. Testing analysis tools such as the Oyente framework[2], which uses a Docker image to deploy SC as it includes all the pre-requisite dependencies, can only detect semantically related security errors but cannot detect the flaws associated with logic. EThir is a framework designed for high-level analysis of Ethereum bytecode[3], and it leverages the Oyente framework's control flow graph (CFG) algorithm for EVM bytecode analysis, but it makes little progress in terms of improving the CFG algorithm's recovery capability. Securify[4] is an online scanning tool that uses Datalog solvers to scan for SC security flaws before deployment. Zeus uses a language interpretation methodology for Ethereum SC verification and can be extended to support other BC platforms for SC validation. Vandal[5] employs the same language interpretation methodology as EVM, but it analyzes the SC bytecode created by EVM using the decompiler. To detect gas (Ethereum transaction fee costs) expensive patterns, the Gasper tool has been devised and is capable of catching seven costly gas patterns. Other costly gas patterns may exist in more complicated SCs, but they have yet to be discovered. Furthermore, programs such as Oyente, Mythril[6], and Securify rely heavily on symbolic execution or symbolic analysis. As a result, the process becomes quite time-consuming and computationally intensive as it requires a study of all the executable paths in a SC or analysis of associated dependency graphs. Hence, such a methodology may not be suitable for detecting batch vulnerabilities.

Secure programming techniques and the use of common patterns can undoubtedly help in reducing vulnerabilities, but the

2 https://github.com/enzymefinance/oyente
3 https://github.com/costa-group/EthIR
4 https://github.com/eth-sri/securify2
5 https://github.com/usyd-blockchain/vandal
6 https://github.com/ConsenSys/mythril

tools and techniques discussed above have their limitations in terms of effectiveness. The primary reason is that they are mostly dependent on a developer understanding and applying them correctly, which is open to being error-prone. Second, these techniques are limited to identifying specific types of vulnerabilities. Automated vulnerability detection tools, like ContractFuzzer (Jiang et al., 2018) and MAIAN (Nikolić, 2018), take into consideration generic qualities/specifications that cannot capture contract-specific requirements; therefore, they are good at discovering common flaws but may not be effective at detecting unique/uncommon vulnerabilities. As a result, automated vulnerability detection tools are not always accurate and can result in false positives. Hence, both static and dynamic tools normally require the involvement of security experts to specify security properties and patterns at a low-level/byte-code level.

(B) ***Machine learning for verification:*** ML makes the vulnerability detection process faster and more reliable. Currently, there is significant research interest in utilizing ML for both SC verification and vulnerability detection within the research community.

To move beyond the limitations of static and dynamic methods, recent years have witnessed a shift toward the adoption of ML for detecting bugs and flaws in SCs. The problem here is that, until now, the solutions available have used supervised ML algorithms that need historical data for learning and detecting specific patterns (Sun et al., 2023). For instance, two prominent frameworks, namely, ContractWard (Wang et al., 2020) and SoliAudit (Liao, 2019), are based on supervised ML. ContractWard is used to detect six types of Ethereum SC security vulnerabilities using ML classification methods, whereas SoliAudit uses ML to discover vulnerabilities in Ethereum SCs by integrating static and dynamic analyzers.

This approach is reasonable for specific issues and offers assistance in many circumstances to automate the classification tasks. Other techniques, such as the long-short-term memory model from Tann (2018), ensemble learning-based SC vulnerability prediction (SCVDIE-ENSEMBLE) mechanism by Zhang et al. (2022), the predictive model by Momeni et al. (2019), and the deep learning-based approach from Lesimple and Martin (2020), are capable only of detecting those vulnerabilities for which they have been trained, and as such, they cannot identify other potential vulnerabilities, which, in turn, limits their applicability. ML algorithms are susceptible to the zero-shot learning problem; if your ML model is not properly trained, you cannot detect patterns and flaws accurately, and in the case of sparse data sets, ML model training will be poor. Although these ML-based approaches are great for finding bugs, they cannot guarantee their absence.

To cope with the limitations of the existing ML-based SC verification techniques, there is a need to leverage Semantic AI[7] technology. This is based on combining AI methodologies such as ML, natural language processing (NLP), text mining, knowledge modeling, and Semantic Web. Semantic AI leverages the benefits of both statistical and symbolic AI strategies, in particular, semantic reasoning and neural networks (NNs). This combination requires less training data in the learning process because a semantic knowledge graph is used as the core element in a semantic-enhanced AI architecture, which offers automated data quality management. AmpliGraph[8] is one technique that is based on the Semantic AI concept and is based on knowledge graph embeddings (KGEs), where entities and relationships are embedded in low-dimensional vector spaces (called embeddings), which can be utilized to uncover hidden knowledge. When compared to standard deep learning (DL) representations, knowledge graph-derived representations like AmpliGraph have the following advantages: it does not inherit ambiguities because each entity (subject) has a relationship with its originator (object), making it useful for modeling reasoning and explainable systems. It offers a semantic layer to help with reasoning (Q&A) tasks, information retrieval, entity disambiguation, etc. KG representation can be used as an input to DL algorithms to link two far-apart worlds. Additionally, KGE is considered the best option for recommender systems in terms of improving performance and the explainability of the recommender system (PALMONARI and Pasquale, 2020). Hence, representing knowledge in a vector space serves three purposes if used in the context of contracts: 1) addresses the explainability issue; 2) predicts potential vulnerabilities; and 3) makes recommendations about existing templates, scripts, and test data. Additionally, SC validation and verification can be automated when Semantic AI techniques, such as AmpliGraph, are used in conjunction with formal verification.

(C) ***Formal methods for verification:*** SC correctness verification using theorem-proving and model-checking is particularly compelling because of its mathematical properties. The use of formal reasoning for SC development and the analysis of formal models provide insight in terms of finding a wide range of vulnerabilities. Alharby and Van Moorsel (2017) conducted a systematic survey on SCs, in which they highlighted the current SC open challenges and research gaps. They also classified the challenges into four categories, namely, security, privacy, codifying, and performance-related issues. Based on the detailed survey, they have suggested that formal verification is one of the most important solutions to address these issues. Moreover, in recent years, many formal methods have been used for SC quality assurance.

Formal verification tools are based on formal operational semantics and offer strong security verification guarantees at pre- and post-SC deployment stages. They allow for the formal specification and verification of contract properties, as well as the detection of both common and uncommon vulnerabilities that could result in a security violation. Formal verification for SCs is typically based on theorem provers like the F* framework (Bhargavan, 2016), Event-B, SMT Solver (Alt and Reitwiessner, 2018), and proof assistants such as FEther (Yang and Lei, 2019), Coq, and Isabelle/HOL (Amani, 2018). These approaches are

7 https://www.poolparty.biz/machine-learning-meets-semantics

8 https://github.com/Accenture/AmpliGraph/

remarkably expressive, but they come at a high complexity cost, which demands an extensive user interaction to guide the theorem prover, making it difficult to verify complex systems. Other techniques such as symbolic execution and model checking are also frequently used. These methods are typically used to ensure that SCs are functionally correct. One of the benefits of model-checking methods is that they can be automated, and if a specification is proved false, they offer counterexample paths (demonstrates a run of SC that violates the specified properties). However, model checking has the drawback of causing a state explosion problem if a system has a high degree of concurrency (when the number of state variables in the system increases the size of the system, state space also increases exponentially). It cannot verify a system having infinite states. Additionally, these methods have shortcomings when it comes to validating a contract's business logic, i.e., if the contract logic has concurrency problems, the vulnerability is most likely due to an irrational design (Duo et al., 2020). In addition, each formal verification technique and tool addresses a specific problem, for example, Le (2018) only verified SC termination and non-termination conditions. Similarly, Nielsen and Spitters (2020) searched for contract invocation verification. With the exception of Alqahtani (2020); Zupan (2020), most techniques are platform- and language-specific (i.e., EVM and Solidity contract code). Despite their expressiveness, the techniques are expensive and inadequate for sophisticated contract verification. Furthermore, none of the existing approaches are completely automated, and expert intervention is still required.

Contrary to this, PN is a lightweight and graphical-oriented modeling approach to formal verification methods that is economical and scalable. CPN is a hybrid of classical PNs and high-level programming languages where the primitives for a process interaction, the ability to model concurrency, communication, and synchronization are facilitated by PNs, and the programming language supports the definition of data types and the manipulation of data values. CPN offers an intuitive graphical interface with well-defined syntax and semantics that allows for transparent conceptual modeling of complex and large systems by introducing the new element "color."

Conclusively, the majority of the theorem-proving and model-checking approaches discussed here are only applicable to Ethereum-based applications. Furthermore, they are complex to learn, complicated to implement, and harder to automate. Moreover, as SC business logic becomes more complicated, the approaches outlined do not include the necessary extension to address a range of high-level vulnerabilities associated with business applications or alternatively low-level vulnerabilities relating to properties such as re-entrancy, transaction-ordering dependence, or mishandled exceptions. For example, CPN, on the other hand, is one of the most advanced behavior modeling techniques. It can support the SC life cycle from requirements to implementation and has a better concurrency model (e.g., "true concurrency") (Duo et al., 2020). Defining distinct color sets can also improve contract description and abstraction. Users can analyze the state of each step of the contract execution using CPN, not just through simulation but also on a formal basis, making it easier to spot weaknesses in SCs and avoid excessive penalties that may be incurred as a consequence of poorly performing SCs.

# 3 Contribution: GRV-SC framework

Based on the review of the studies presented in Section 2.1, six critical challenges have been identified that developers face while trying to ensure correctness when writing SCs: 1) language dependency; 2) complexity of programming languages; 3) verification of contract logic; 4) failure to change or terminate SCs; 5) lack of provision to detect underperforming SCs (Zhang, 2016); and 6) data feed-derived characteristics (Zhang et al., 2019). Although the concept is generalizable for other SC languages, the proposed framework will focus on the DAML SC platform based on the rationale presented in Section 2.2.1. It is worth noting that no well-established modeling and formal verification frameworks or tools are currently available for DAML SCs. Although there is a static analysis tool available, namely, DAML-lf-verifier, according to the pull request (PR), DAML-lf-verifier is no longer supported and is discontinued. In addition, users can test DAML contracts using DAML scripts[9]; however, this is a basic method that might not be sufficient for extensive testing of the contracts. Hence, as a result, by providing a modeling and verification framework, namely, the GRV-SC framework, this study will explore ways to evaluate the security of DAML contracts and ensure their resilience to vulnerabilities and attacks at the pre-deployment phase.

Before delving into the methodology, the upcoming discussion explores the DAML system model and the principles of data modeling. This exploration aims to provide the reader with a better understanding of DAML's capabilities and limitations. Such knowledge serves as a prerequisite for comprehending the proposed study.

## 3.1 DAML system model

As shown in Figure 2, the DAML ledger serves as the foundation for decentralized applications (dApps) built using the DAML language. It maintains a shared state of information across multiple parties and ensures data integrity and consistency through a consensus mechanism. The DAML ledger system model distinguishes between the shared rules governing dApp and the individual strategies employed by its users (Bernauer, 2023; Mustafa, 2023). Shared rules, implemented exclusively in DAML, define the legal and technical rules governing business processes and transactions within dApp. These rules are universally applicable and ensure consistent behavior across all users. On the other hand, each user has the autonomy to define and maintain their own approach for interacting with the ledger, known as their user strategy. This strategy component, encompassing the dApp frontend, is implemented using a prevalent programming language like Java or TypeScript.

- **User interaction with the DAML ledger:** Users interact with the DAML ledger by transmitting DAML commands to a ledger client, referred to as a participant. The participant acts as an intermediary between the user's dApp and the DAML

9 https://docs.DAML.com/2.0.1/DAML/intro/12Testing.html

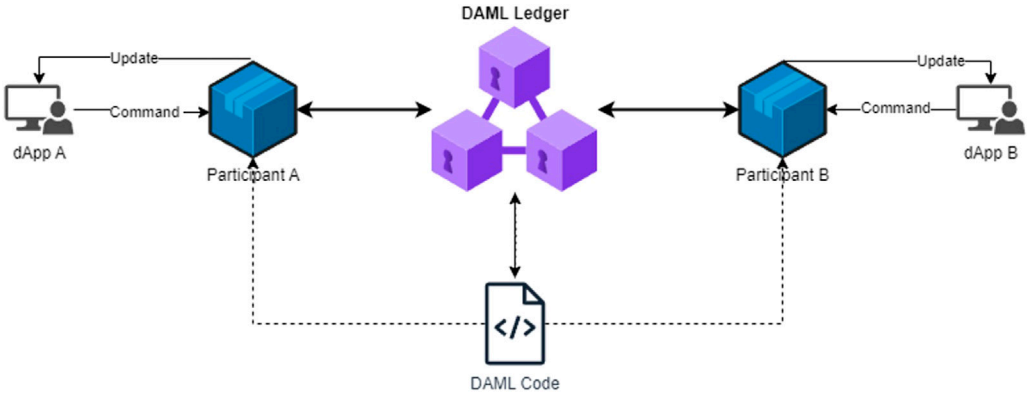


FIGURE 2
DAML ledger system model.

**TABLE 1 DAML contract terminologies.**

| Keyword | Description |
|---|---|
| template | *template* is akin to a *class* which offers the highest level of nesting. It contains |
| | • contract data (e.g., date, description, parties involved, etc.) |
| | • roles (signatory, controller, and observer) |
| | • choices and their respective controllers (who gets to do what) |
| party | Party data-type represents a legal entity. It has four sub-types: signatories, observer, controller, and maintainer |
| signatories | Can view and initiate SC |
| observer | Can only see the SC on the ledger but cannot create it |
| controller | Can view the contract and is also responsible for executing the choices in the contract |
| choice | Choices of a SC template specify the rules on how and by whom contract data can be changed. It can be consuming, non-consuming, pre-consuming, and post-consuming |
| exercise | Choice is exercised on a given SC by the contract id |

ledger, interpreting DAML commands and translating them into actions that can be executed on the ledger.

- **Participant's role:** Upon receiving a DAML command from a user, the participant deciphers the command by executing the corresponding DAML SC code. This execution results in a ledger update, which represents a change to the shared state of information maintained by the DAML ledger.
- **Validation and notification:** The DAML ledger plays a crucial role in ensuring the integrity and consistency of the shared state. It performs two critical functions:

1) Validation: The DAML ledger scrutinizes ledger updates to ensure that they conform to the defined DAML semantics. This validation process safeguards against invalid transactions and maintains adherence to the shared rules governing dApp.
2) Notification: Upon successful validation of a ledger update, the DAML ledger informs the affected participants. These participants, in turn, notify their respective user dApps, ensuring that all users are made aware of the changes to the shared state.

- **Integration with user strategies:** The DAML ledger's validation and notification mechanisms provide seamless integration with user strategies. By receiving notifications about ledger updates, users can adapt their strategies accordingly, ensuring that their actions remain aligned with the evolving state of dApp.

Table 1 outlines the terminology associated with DAML, followed by a section that presents the DAML system model. This aims to enhance the clarity of the proposed study.

## 3.2 DAML data modeling

An introduction to DAML begins with an exploration of consensus modeling 1. The consensus model incorporates three main templates: proposal, decision, and ballot.

**Listing 1.** DAML consensus model.

```
module Main where
import DA.List (sortOn, head)
template Proposal
 with
  proposer : Party
  acceptor : Party
  proposal : Text
  created : Time
 where
  signatory proposer
  observer acceptor
  key (acceptor, proposer) : (Party, Party)
  maintainer key._1
template Decision
 with
  acceptor : Party
  decision : Text
  observers : [Party]
 where
  signatory acceptor
  observer observers
template Ballot
 with
  acceptor : Party
  proposers : [Party]
 where
  signatory acceptor
  observer proposers
  nonconsuming choice MakeProposal :
  ContractId Proposal
   with
    proposal : Text
    proposer : Party
 controller proposer
 do
  now <- getTime
  create Proposal with created = now, .
choice MakeDecision : ContractId Decision
 controller acceptor
 do
  res <- sequence [fetchByKey @Proposal
  (acceptor,p) | p <- proposers ]
  assertMsg "At_least_one_proposer_has_to_
  make_a_proposal"
  $ not . null $ res
  let sorted = sortOn (\(k,v) -> v.created) res
  let (_, firstProposal) = head sorted
  create Decision with decision =
  firstProposal.proposal, observers
  = proposers, .
```

1) **Proposal template:** It represents a proposal made by a party *proposer* (i.e., *Alice*) to another party *acceptor* (i.e., *Bob*). It contains details such as the proposal text, proposer, and the time it was created. The *proposer* is the signatory, and the *acceptor* is the observer. A key is defined based on the *proposer* and *acceptor*, with the maintenance of the key's first element. Notably, the DAML ***Party*** data type signifies an entity with the capability to engage with a ledger, performing actions such as signing contracts and submitting transactions. The exact cryptographic procedures employed to record, verify, and authenticate these actions differ based on the implementation. Generally, all DAML ledgers necessitate connecting each transaction to cryptographic evidence for authorization and non-repudiation (Bernauer, 2023).
2) **Decision template:** This template embodies the decision made by an *acceptor* in response to proposals, encapsulating the decision text and a list of observers. The *acceptor* is the signatory, and observers as *observer* are specified.
3) **Ballot template:** It represents a ballot conducted by an *acceptor* involving multiple proposers.
   - Lists the *acceptor* as the signatory and *proposer* as observers. It includes two non-consuming choices:
   - *MakeProposal:* Allows proposers to create a proposal. The proposer specifies the proposal text and acts as the controller.
   - *MakeDecision:* It enables the *acceptor* to make a decision based on proposals received. The *acceptor* is the controller.
   - Fetches proposals from proposers and ensures at least one proposal exists.
   - Sorts proposals based on creation time and selects the first proposal.
   - Creates a decision with the selected proposal's text and the list of proposers as observers.

This consensus model utilizes the observer and signatory clauses to define notification and authorization rules for contract instances. The key and maintainer clauses are used for managing contract keys and their maintenance. The model reflects a process of proposing, deciding, and balloting to achieve consensus among participants.

# 4 Methodology

To address the six issues identified at the beginning of Section 3, a cross-platform SC life-cycle management approach known as the *GRV-SC framework* is proposed. The GRV-SC framework is a reference pattern that encapsulates a set of functional blocks to support developers and designers in creating robust SCs. GRV-SC leverages a model-driven approach to automate the SC development workflow by incorporating the key life-cycle phases (modeling, pre-deployment, and post-deployment verification) through the implementation of three interconnected modules, as shown in Figure 3; 1) a SC designer to support the definition of a SC model and use case specification; 2) validation tools to support automated formal verification; and 3) a test execution environment to detect vulnerabilities that may arise during network deployment.

The following section provides an overview of the three key modules that encapsulate the SC development life cycle.

## 4.1 GRV-SC model designer module

A contract specification defines the rules that govern how a contract will function in accordance with its design. The GRV-SC

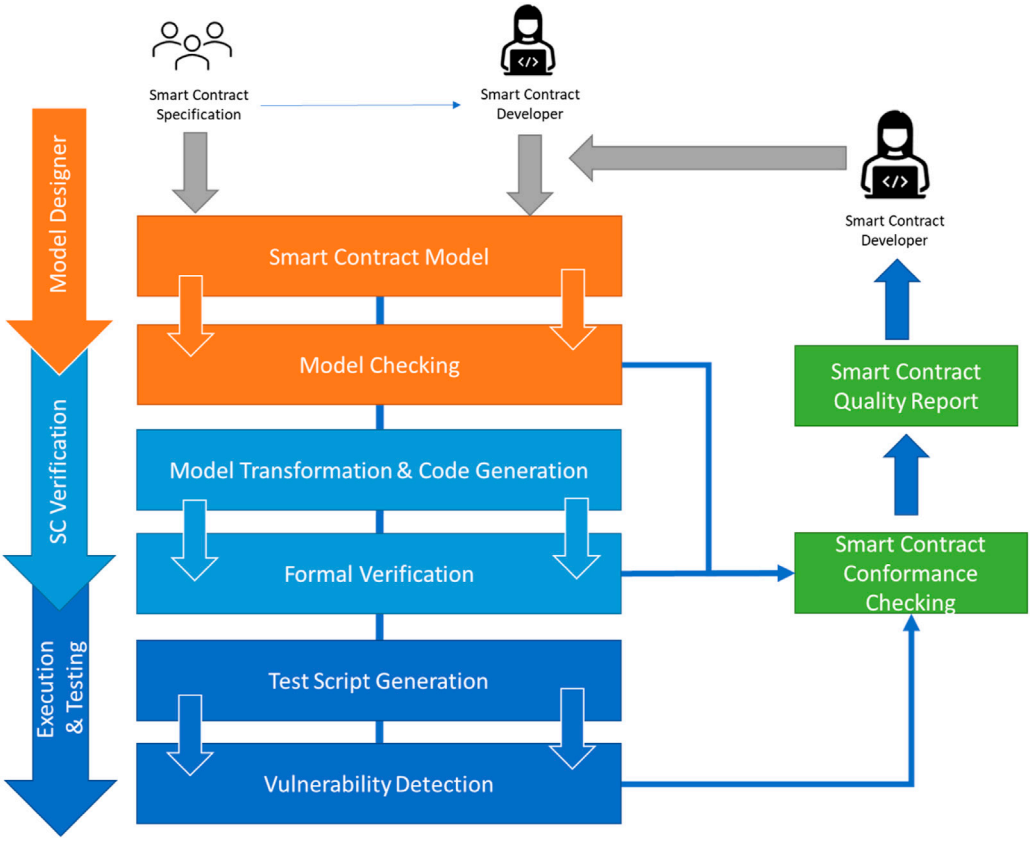


FIGURE 3
GRV-SC framework functional modules.

model designer is used by the developer to translate the SC requirement specification to a contract model and associated code. This is achieved by encapsulating the DAML development tools and modeling language as part of a user interface where the user can drag and drop the design controls from a pre-configured toolbox that defines the structure and business logic of SC. This designer also provides a mechanism to import and parse existing DAML contracts to visualize the data flow and interactions as a logical graph. The designer also offers model-checking functions (based on rule sets) for describing and evaluating the DAML SC model in a way that may be used to identify logical and run-time vulnerabilities using basic verification methods. This module is used to simplify the complex structure of contract code and to formalize numerous activities and tasks that encompass the software development life cycle (SDLC) (e.g., requirement definition, quality checking, and visual analysis of contract logic).

## 4.2 GRV-SC verification module

This module provides a library that allows for contract modeling and verification of a SC using CPN. The library is designed to allow the user to select a CPN model (*.cpn* file format) and policy files for a specific use case scenario to jointly build a DAML contract. Two classes of vulnerabilities have been identified as open issues in DAML, and addressing these is the focus of the verification module.

1) **Access control:** This DAML vulnerability arises when the programmer mistakenly grants a specific party the authority to exercise choices they are not permitted to. This vulnerability can only be identified during the modeling step as it is related to the DAML template structure. Hence, the CPN model can assist in analyzing the behavior of the parties (i.e., identifying valid/invalid access rights) in the context of the simulated dynamic interaction of the user's behavior.
2) **Insecure direct object reference (idor):** In DAML, every choice on a template is controlled only by the choice controller and can, in particular, be called publicly which causes an idor vulnerability. To avoid this, the controllers do not depend on the choice arguments but only on the template arguments. The signatories should have already validated the controllers when the contract was created. However, such checks are sometimes omitted, which may lead to the presence of this vulnerability. Hence, as part of the GRV-SC, a type safety dynamic verifier has been developed to catch the idor vulnerability. The type safety verifier checks whether a choice gets the contract ID and data on an existing DAML contract as arguments. If so, then it is the choice body's responsibility to check whether the contract ID indeed refers to given arguments, e.g., by fetching the contract from the ledger again. This concept can be derived as follows:

**Lemma 4.1.** *Let T and T′ be two distinct templates, C be a controller of the template T, x′ and y′ be T′ parameters, and 's' be a party to T. If T exercises choice on T′ parameters, then C should depend on T′ parameters x′ and y′ instead of choice arguments of T. Therefore, 's' should validate C at the time of the creation of T.*

**Proposition 4.2.** By using the propositional logic:

(a) $E(T, x', y')$: *T exercises choice on T′ parameters x′ and y′.*
(b) $D(C, x', y')$: *C depends upon T′ parameters x′ and y′.*
(c) $D(C, T)$: *C depends upon the choice arguments of T.*
(d) $I(v)$: *idor vulnerability arises.*

Propositions that are assumed to be true:

(a) $E(T, x', y')$
(b) $E(T, x', y') \rightarrow D(C, x', y') \wedge \neg D(C, T)$: *If T exercises choice on T′ parameters then C should depend on T′ parameters x′ and y′ instead of choice arguments of T*
(c) $\neg D(C, T) \rightarrow \neg I(v)$: *If C does not depend upon the choice arguments of T, the idor vulnerability does not arise.*

**Proof.** The proposition (*b*) can be re-written as $\neg E(T, x', y') \vee (D(C, x', y') \wedge \neg D(C, T))$ (using $p \rightarrow q \equiv \neg p \vee q$)

which can further be written as follows:$(\neg E(T, x', y') \vee D(C, x', y')) \wedge (\neg E(T, x', y') \vee \neg D(C, T))$ is true using distributive law: $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$.

If the above statement is true, then

$\neg E(T, x', y') \vee D(C, x', y')$ is true and

$\neg E(T, x', y') \vee \neg D(C, T)$ is also true. So this can be re-written as follows:

$\equiv E(T, x', y') \rightarrow \neg D(C, T)$ (using $p \rightarrow q \equiv \neg p \vee q$)

$\equiv E(T, x', y') \rightarrow \neg I(v)$ is true using above statement and ***(c)*** i.e., $p \rightarrow q, q \rightarrow r \equiv p \rightarrow r$.

The last statement says that if *T* exercises choice on *T′* parameters *x′* and *y′*, then the idor vulnerability does not arise. Hence, the statement was proved.

These vulnerabilities can have a significant adverse impact and can cause substantial financial, legal, and reputational losses to the organization that is a DAML client. The exploitation of these vulnerabilities can lead to several attacks, i.e., unauthorized data access may result from idor vulnerability whereas access control might result in privilege escalation, tampered transactions, and denial-of-service (DoS) attacks. Some of the examples are illustrated later in Section 5.3.

## 4.3 GRV-SC execution and testing module

The execution and testing module includes an auto-test execution environment for SC vulnerability detection at the functional level through test case generation based on existing contracts and script recommendations for reusability and optimization. This module consists of three steps: 1) feature engineering performed on the data model obtained at the transformation step to create a knowledge-based graph (KBG); 2) the use of ML models to uncover unidentified relations in KBG, and 3) the use of the named-entity recognition (NER) method for test script recommendations. Step 1 is dependent on the designer module output, i.e., the contract-to-data model transformation step (Section 5.3), where the obtained data model from the defined SC is subjected to feature engineering in order to extract the relevant features of DAML SC. With the use of these extracted features, neural embedding models, i.e., **TransE** and **ComplEx** for training, can be used to generate knowledge graph embeddings (KGEs), which reflect the syntactic and semantic structures of the SC function in metric space (by encoding concepts and links of a graph into low-dimensional vectors). The resulting embedding is then combined with model-specific scoring algorithms to anticipate unseen novel links and assist in the detection of possible SC security vulnerabilities that have been overlooked or for which the model has not yet been trained. The generated contract KBG along with the detected security vulnerability list will be stored as part of the conformance checking process and reused for future developed contracts.

### 4.3.1 NER for test script recommendation

To support the script recommendation, a NER feature is added. For instance, a DAML contract is an item that outlines the contract requirements (functions that distinguish one contract from another), i.e., [*SC_A = item_A*]. On the other hand, there may already be a number of relevant DAML scripts (test cases) stored from previous developments against each item index (SC), along with a list of vulnerabilities already identified. These scripts are known as test items, i.e., at $Index_0 \rightarrow$ [*Item_A*] = [*testitem_1, testitem_2*][*Vulnerabilities_list*].

For instance, if a user imports a new SC named "B" to reuse the existing scripts, the initial step is to transform *SC_B* into a partial KG, which is then matched to existing SC knowledge base graph (KBG) NER, which extracts the features (data points) with a high information gain for a recommendation. This kind of linkage would inherently contain more information, and it would also allow graph traversal to be used during the matching phase (to discover related SCs in the shortest amount of time). Later, for new contracts, the user can reuse the proposed test workflow KBGs. These KBGs produce a template that a user may use to test their contracts against previously developed and verified use cases with minimal effort, which makes the overall quality of SC creation better. In addition, if the user does not have initial test data for CPN simulation, the stored scripts (and associated test data) can be used in the verification step.

# 5 Experimental evaluation and results

The discontinuation of DAML-lf-verifier, as indicated in Section 3, means that it is no longer supported or actively maintained. Consequently, it is not possible to verify results using the latest version of DAML, making it difficult to perform cross-verification or conduct a comparative analysis with the proposed work. To overcome this challenge, two approaches were undertaken; first, a qualitative analysis was performed involving active members of the DAML community; in addition, a case study scenario was defined and used to evaluate the proposed study's methodology, workflow, and findings.

**TABLE 2** SWOT analysis of the GRV-SC approach and methodology.

| Strength (S) | Weakness (W) |
|---|---|
| • Aligned with well-established development practices for S/W testing | • Real-world validation limited to date |
| • Additional focus on security for smart contract development | • There are some limitations to static analysis tools |
| | • Challenges exist relating to contract modification and versioning in DAML not captured by the GRV-SC framework |
| **Opportunities (O)** | **Threats (T)** |
| • Extending existing practices with new tools for the verification of smart contracts | • The complexity of smart contracts makes it challenging to capture all facets of vulnerabilities |
| • Improved security tools for smart contract developers and testers | |
| • Enhancing static analysis of contracts and logic | |
| • Increased adoption and integration of DAML contracts through availability of support tools | • Resistance to change from developers with new tools and methodologies |
| • Awareness and education of the importance of securing smart contracts | |

**TABLE 3** Experimental environment for testing the GRV-SC framework.

| Name | Tool and version |
|---|---|
| Programming Language | DAML v2.2.0; .NET 6.0 |
| Network Environment and Testing | DAML HUB; CPN simulation |
| Editor | Visual Studio Code; Visual Studio 2022; CPN 4.0.1 |
| Operating System | Windows 10 |

## 5.1 SWOT analysis

To perform a strength, weakness, opportunity, and threat (SWOT) analysis of the GRV-SC framework, an exploratory qualitative study involving semi-structured interviews was undertaken. The objective of the interviews was to gain insights into the practices currently employed by industry professionals with significant experience in developing SCs. The analysis focused on the management of SC developed, deployed, and managed on a blockchain network. The primary objective of this analysis is to uncover the methods, tools, and strategies currently employed by developers to ensure the security and resilience of SCs, thus complementing the analysis carried out in Section 2.2. This also included an understanding of the importance of education, awareness, and the utilization of advanced tools in guaranteeing the security of SCs. Finally, this analysis facilitates the exploration of the potential benefits of the proposed GRV-SC framework and its alignment with requirements and current industry practices. The SWOT analysis included 8–12 participant interviews with various roles, including language engineers, developers, and enterprise users. Table 2 provides a summary of the interviews conducted in the form of a SWOT analysis. This covered the approach and tools proposed as part of GRV-SC. Generally, having such a framework was viewed positively; however, further validation in real-world scenarios is required to minimize resistance and promote its integration into the development life cycle. Further exploration of these outcomes will be carried out in further work.

## 5.2 Experimental environment

The experimental settings for evaluating the proposed GRV-SC framework are listed in Table 3: 1) DAML Parser; 2) DAML HUB for deployment and testing of the DAML contract. It is a cloud platform that enables DAML users to build simpler and more scalable serverless backends for their applications by allowing an interaction with live ledgers; 3) NET 6 used to develop the dynamic test verifier and CPN-DAML translator (code generation engine). To implement the experiment, the Windows 10 operating system with Visual Studio Code, Visual Studio 2022, CPN 4.0.1, and DAML v 2.2.0 has been used.

## 5.3 Case study for GRV-SC framework evaluation

To expand on the modules as described above, the following steps (from a developer's perspective) are depicted in Figure 5. So, to explore the effectiveness of each module, an auto-service center contract has been taken to illustrate the analytical process, as shown in Figure 4. The contract is designed to provide a secure and reliable car maintenance service to the car owner in exchange for a monetary sum. In this use case, three parties are involved:

1) The Bank: issues cash as contracts
2) CarOwner: who holds account in bank
3) CarShop: auto-service center, who offers repair service in exchange of cash

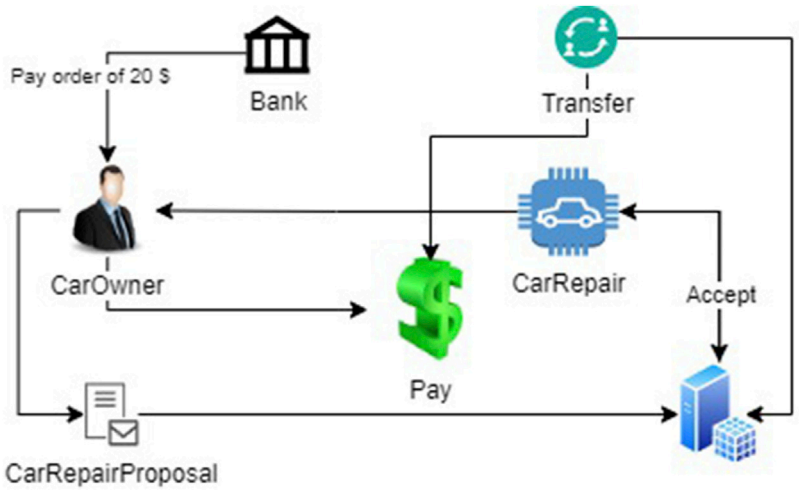


FIGURE 4
Auto-service center use case.

SCs must satisfy the following use case properties and attribute constraints:

- Create a template for the Bank and instruct the Bank to give cash to an account holder, the CarOwner.
- Add a choice/function to the Bank template to make this money transferable.
- Consider a bilateral service agreement, which calls for the CarOwner and CarShop to agree on a service and a fee. Without the other party's consent, neither party may enforce this agreement against the other.
- Utilize the propose–accept model to construct this bilateral agreement.
- Add a choice to CarOwner's payment to indicate their satisfaction with the service.
- Add constraints to guarantee that the money received satisfies the terms of the service contract.

1) **Design:** In order to construct a *Contract Model* of 4, the framework uses the designer module to construct a DAML template along with a logical graph, with relevant data structures defined for each of the control points. This model-driven approach aims to limit compile-time errors and syntax errors (i.e., if a contract has incorrect inputs, or any repeating transaction (due to coding errors), under-optimization, reduce the associated learning time for a new language). However, a user can also directly import the DAML contract code to the framework in order to generate a contract graph, which represents the contract flow.
2) **Transformation:** A DAML Parser is then used to transform a defined contract model/imported contract model into a *Data Model* to extract data points (see Figure 5, transformation block). These data points are distinguishing characteristics of SC that help to uniquely identify it from other contracts. These can be any of the DAML entities (controllers, signatories, and so on), as well as calculated data points such as relations from an observer or signatory. This transition accomplishes two goals: first, the retrieved data points are utilized to build a contract graph. Second, they were subjected to feature engineering before being fed into the ML model (see Figure 5, conformance testing and recommendation block). This transformation is needed to convert the DAML contract into a common data structure for data exchange and interoperability (e.g., list and dict) because the proposed framework modules (such as designer, verification, and execution) must be synchronized.
3) **Modeling and verification:** In order to generate a secure DAML SC template for the auto-service center, the above-mentioned contract properties (i.e., authorization, control flow, and functional/choice correctness) are verified by the CPN model. This CPN model (defined as an *.cpn* file) is then automatically transformed into a secure DAML template using the GRV-SC framework. The generated DAML template is executable on both Hyperledger Fabric and the DAML ledger platform.

Furthermore, in this study, we are focusing on two interesting classes of vulnerabilities:

1) **Access control:** The vulnerability occurs when the controller of the template is chosen incorrectly and authorized it to conduct certain actions, i.e., in the CarServiceCenter template when the CarShop will be the controller of the *Pay* choice, instead of CarOwner. So, at the modeling stage, party authorization (i.e., signatories, observers, and controllers) and their behavior concurrency are chosen to be the target states. The backward reachability approach (Bouali et al., 2009) has been applied to carry out control flow and data flow analysis in order to determine the position of the transition guard and the number of tokens at each place. It can also be used to verify the behavior of the parties or legal entities, i.e., who is authorized to do what (who creates a contract or exercises choices) at the pre-deployment stage. Finally, we obtained a place data structure, i.e., *STRING*REAL*CURRENCY*BOOL* = (*signatory*, *amount*, *currency*, *state*), which represents the information regarding the party, balance, currency, and the state of the transaction, respectively.

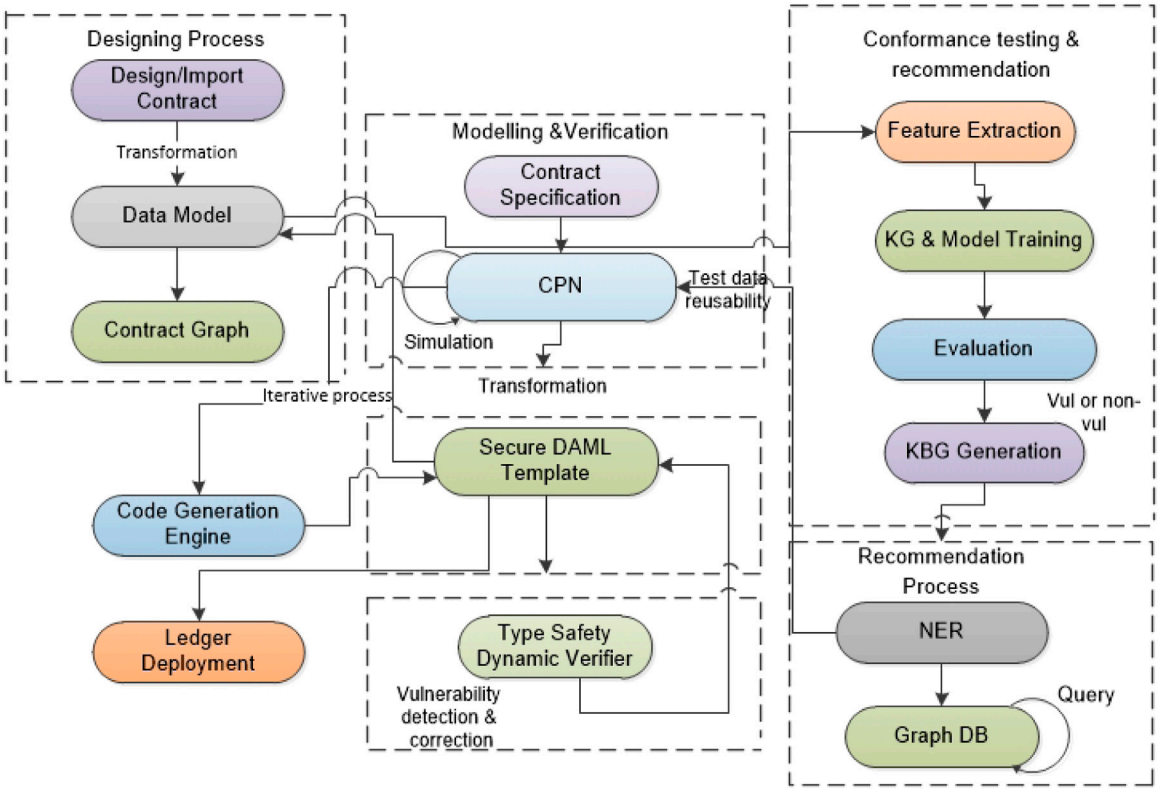


FIGURE 5
GRV-SC framework workflow.

As shown in Figure 6, modeling notations according to the DAML structure are defined as follows:

- **Places:** Places represent the variable and party's name.
- **Color set:** Color set represents the datatype of variable, i.e., compound color set: colset *currency* = with *USD*|*EUR*|*CAD*, string color set: colset *observer* = *string*. Initialize the color set with "owner name," i.e., assign the variable proposer a value. Here, currency as a record type has been created.
- **Transition:** Template name (class name) and choices (functions) are represented as transitions, i.e., blue color transition represents the DAML template name, and all places pointing toward this transition are members/parameters of that template in DAML, whereas green color transition represents **Choices,** i.e., as shown in Figure 6, CarRepairProposal, Bank, and CarServiceCenter represent the template, and each template has associated transitions such as *Accept* is the CarRepairProposal transition; the Bank has *Transfer* whereas the CarServiceCenter has the *Pay* transition. These green box transitions, i.e., *Accept*, *Pay*, and *Transfer* are the choices of these templates, while *create_proposal* is the event that occurs on the acceptance of the proposal by both parties marked as a red color transition. In addition, a place can be added to connect the transition, but naming the place is not required.
- **Arc:** CPN Arc inscription represents token flow and guard conditions.

Table 4 provides a summary of transitions in Figure 6. As shown in Figure 6, the Bank contract represents an amount of cash issued by a party named bank and owned by a party, i.e., the car owner in this case. Additionally, a bank must be a signatory in order to construct a contract in DAML. It is also observed that the Bank contract is independent of other contracts. However, the CarServiceCenter contract is bilateral, which requires authorization of the car owner and car shop before a contract is made. A single party, i.e., either the car owner or the car shop alone cannot create a CarServiceCenter, and it would result in a failed execution. For this reason, a propose–accept model is employed to build a CarServiceCenter contract with two signatories, where the car shop initiates the proposal step by creating a CarRepairProposal contract. The car owner then executes the choice Accept in the CarRepairProposal contract to accept the proposal. Additionally, once the job is finished by the CarServiceCenter, the agreed-upon amount is transferred to the car shop; however, if the owner attempts to exercise choice transfer before the job is actually finished, the execution will fail.

***Vulnerability impact***: For instance, changing the owner as a signatory in the Bank template could lead to several vulnerabilities, including the following:

1) **Malicious owner:** If an attacker gains control of the owner party, they could effectively take control of all Bank contracts associated with that owner. This could allow them to transfer funds out of the contracts or even archive them, preventing the rightful owner from accessing their funds.

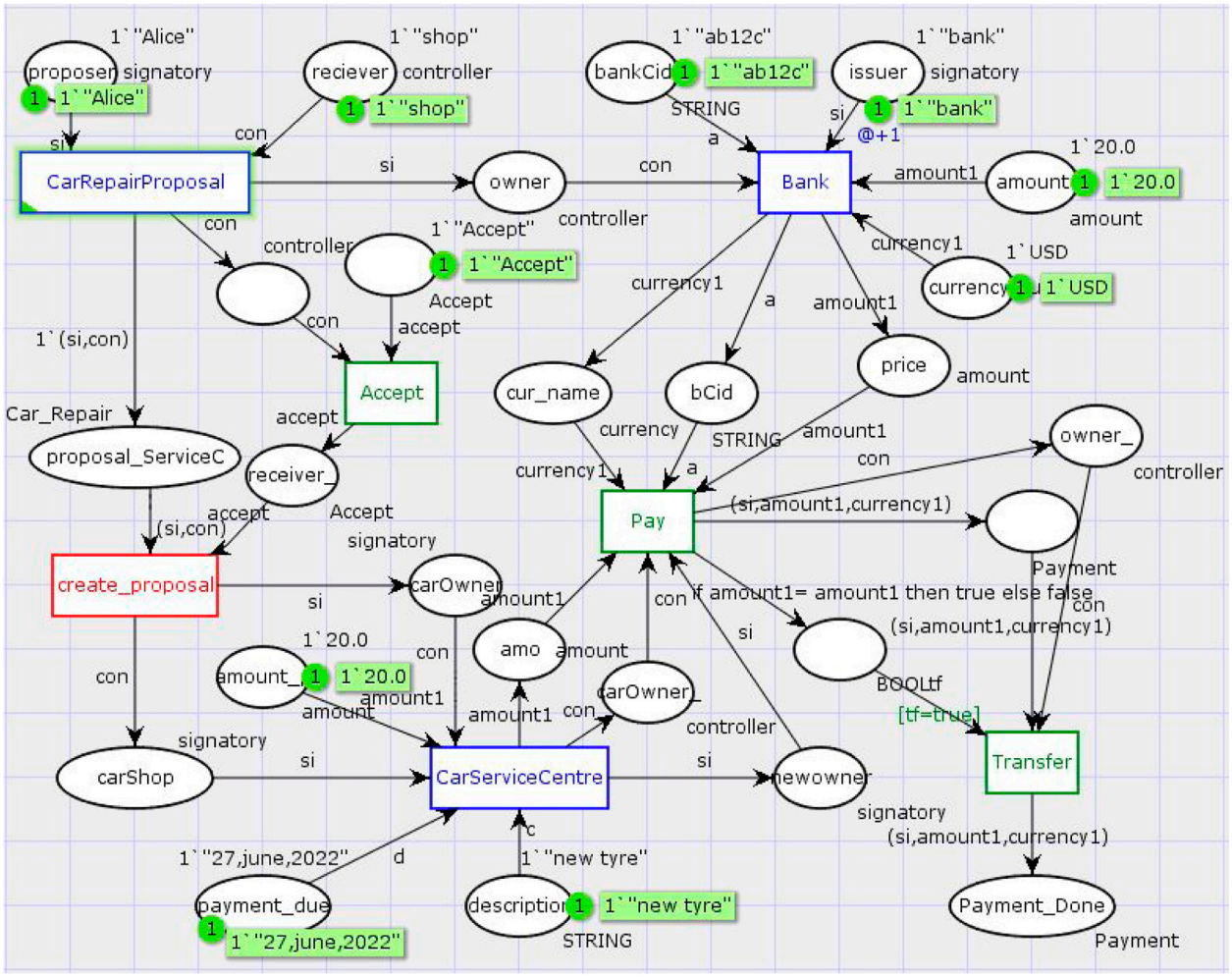


FIGURE 6
CPN model of the auto-service contract.

**TABLE 4 Summary of model transitions.**

| Model transition | Importance |
|---|---|
| Cash | Cash issued by a bank to the car owner |
| CarRepairProposal | Based on the propose–accept model, this contract will create another contract name CarServiceCenter |
| CarServiceCenter | Offers the maintenance service by the authorization of both parties, the car owner and car shop |
| Transfer | Action through which the car owner transfers money to the car shop |
| Pay | Car owner accepts service with the satisfaction which triggers the cash transfer from the car owner to carshop by exercising the choice Pay |
| Accept | Car owner accepts the proposal by exercising the Accept choice |
| create_proposal | CarShop initiates the proposal step by creating a CarServiceProposal contract. The car owner then accepts the proposal by executing the choice Accept in this CarServiceProposal contract |
| Constraints/guard conditions | Price required in the CarServiceCenter contract and the amount specified in the Cash contract should be the same |

2) **Unauthorized transfers:** By being able to sign transactions on behalf of the Bank, the owner could potentially initiate unauthorized transfers, moving funds to their own accounts or those of other unauthorized parties.

3) **Denial of service:** An attacker with control of the owner party could simply archive all Bank contracts associated with that owner, effectively denying the rightful owner access to their funds.

Leaving these vulnerabilities unaddressed could have significant impacts, including the following:

- **Financial losses:** Unauthorized transfers or the loss of access to funds could result in substantial financial losses for the rightful owner.
- **Reputational damage:** If customers become aware of security vulnerabilities in the Bank template, it could damage the reputation of the organization and erode customer trust.

Thus, to mitigate these risks, it is crucial to implement robust security measures, such as employing strong access control mechanisms.

2) **Insecure direct object reference (idor):** The implementation of the Pay choice in Figure 7A introduces a logical error (idor). In DAML, the contract ID and the contract payload are not interchangeable. Thus, by doing *bank = Bank* creates a new template on the ledger that did not exist previously and, as such, requests the user to re-enter all the template arguments again. As the field, *bank* in the *Pay* choice has no relation to the field *bankCid*. The values in these two fields can represent two completely different contracts. The value of the *bank* field can be a DAML record of type *Bank*, which does not exist on the ledger. Suppose the contract ID the user passes to the *bankCid* field refers to a *Bank* contract representing 20 *EUR*. Then, according to the business logic of the implemented contract, the exercise of the *Pay* choice should fail because the currency and the amount of the *Bank* contract do not match the currency and the price of the CarServiceCenter contract, which is, for instance, 200 *USD*.

**A**

```
nonconsuming choice Pay : ContractId Bank
        with
          bankCid: ContractId Bank
          bank: Bank
      controller carOwner
      do
      assert (bank.currency == USD && bank.amount == price)
exercise bankCid Transfer with newOwner = carShop
```

Vulnerable Contract

**B**

```
nonconsuming choice Pay : ContractId Bank
        with
          bankCid: ContractId Bank
        controller carOwner
        do
          bank <- fetch bankCid
          assert (
            bank.currency == USD &&
            bank.amount == price)
exercise bankCid Transfer with newOwner=carShop
```

Resilient Contract Code

FIGURE 7
CarServiceCenter contract. **(A)** Vulnerable contract. **(B)** Resilient contract code.

***Vulnerability impact:*** Due to this error in the Pay choice, a user enters an amount and currency into the *bank* field of the *Pay* choice that does not match those that were agreed upon. This could allow the user to pay for the car repair with a different amount or currency than was originally agreed upon or even to pay for the car repair with a different bank account altogether.

- **Untrusted payment processing:** This vulnerability could have a significant impact on the business logic of the CarServiceCenter contract. For example, if the user pays for the car repair with a different amount than was originally agreed upon, the CarServiceCenter may not be able to complete the repair. Or, if the user pays for the car repair with a different bank account, the CarServiceCenter may not be able to verify that the payment is valid.
- **Risk of incorrect fund allocation:** The vulnerability could enable unauthorized parties to manipulate payment amounts or currencies, leading to erroneous payments and misallocation of funds. This could result in financial losses for the CarServiceCenter or the car owner, and it could also disrupt business operations and erode customer trust.

The dynamic type safety verifier is used to detect such types of vulnerabilities by employing input validation and authorization checks. It ensures the contract ID must be passed to the choice in order to use a contract in a choice (such as to archive the contract or exercise a choice on it). Furthermore, if the contract data are used in the choice, the contract must be fetched using the contract ID, and the controller of that choice validates whether the contract ID indeed refers to the given arguments. Figure 7B represents the resilient approach. In this example, the *carOwner* is the controller of the *Pay* choice of the CarServiceCenter contract and will validate that it has indeed given their authority to exercise the choice, either by directly submitting the command that exercises the choice or by delegating this to a third party using another DAML contract.

For instance, if the verifier detects that a consuming choice has been exercised multiple times without appropriate authorization, it may identify this as a possible idor vulnerability. The initial implementation of the dynamic type safety verifier includes data-type checks, idor, and access control. It aids in detecting potential vulnerabilities in DAML contracts that may have been overlooked during static analysis or manual testing.

Overall, the framework offers a single-stop solution to generate secure DAML contracts that are executable on all DAML-supported platforms, which increase its adoption among new developers.

## 6 Conclusion and future direction

This study discusses the challenges programmers have with SC security and how security measures and tactics can be integrated within the SC development life cycle. The work aims to underline that SC security cannot be guaranteed by following a single-step process such as the root cause analysis of a contract's design, which is merely one step of a multi-step process that cannot ensure SC

security; user behaviors during contract execution must also be taken into account. As such, to minimize the associated risk and obtain a level of assurance of contract security, an iterative SC life-cycle management mechanism (Figure 1) is required. As a result, the GRV-SC framework is proposed, which provides a mechanism to incorporate MDE and formal verification into SC life-cycle management in order to make it easier for the developer to specify, code, test, and generate a reliable SC code. Following this methodology within the development process targets a reduction in the designing and verification overhead at later stages and aims to minimize the risk of poorly constructed code being deployed on a ledger. The framework comprises three components, namely, designer, verification, and execution. This study covers the implementation results for the first two components respectively and focuses on finding and fixing two significant classes of DAML vulnerabilities, i.e., access control and idor.

As future work, further development, and integration of the tools are ongoing, they will be evaluated in detail, considering a set of common vulnerabilities and coding issues observed in SC development. The process of formal verification will be improved by utilizing the "PIPE + VERIFIER" approach, which was introduced by Liu and He (2015). This approach allows for the analysis of high-level PNs using a state-of-the-art SMT solver Z3 as the backend engine. It enables the analysis of high-level CPNs (HL-CPNs) by translating HL-CPN models to SMT formulae.

## Data availability statement

The original contributions presented in the study are included in the article/Supplementary Material; further inquiries can be directed to the corresponding author.

## Author contributions

IM: conceptualization, formal analysis, methodology, validation, visualization, and writing–original draft. AM: conceptualization, investigation, supervision, and writing–review and editing. SR: conceptualization, funding acquisition, investigation, supervision, and writing–review and editing.

## Funding

The author(s) declare financial support was received for the research, authorship, and/or publication of this article. This research work is supported by the Science Foundation Ireland Centre for Research Training focused on Future Networks and the Internet of Things (AdvanceCRT), under the grant number 18/CRT/6222.

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Publisher's note



## References

Abdellatif, T., and Brousmiche, K.-L. (2018). "Formal verification of smart contracts based on users and BC behaviors models," in *2018 9th IFIP international conference on new technologies, mobility and security (NTMS)* (IEEE).

Abrial, J.-R. (2010). *Modeling in Event-B: system and software engineering*. Cambridge University Press.

Albert, E. (2020). "Synthesis of super-optimized smart contracts using max-SMT," in *International conference on computer aided verification* (Cham: Springer).

Alharby, M., and Van Moorsel, A. (2017). *Blockchain-based smart contracts: a systematic mapping study*. arXiv preprint arXiv:1710.06372.

Allamanis, M., Brockschmidt, M., and Khademi, M. (2017). *Learning to represent programs with graphs*. arXiv preprint arXiv:1711.00740.

Alqahtani, S. (2020). "Formal verification of functional requirements for smart contract compositions in supply chain management systems," in *Proceedings of the 53rd Hawaii international conference on system sciences*.

Alt, L., and Reitwiessner, C. (2018). "SMT-Based verification of solidity smart contracts." leveraging applications of formal methods, verification and validation," in *Industrial practice: 8th international symposium, ISoLA 2018, limassol, Cyprus, november 5-9, 2018, proceedings, Part IV 8* (Springer International Publishing).

Amani, S. (2018). "Towards verifying ethereum smart contract bytecode in Isabelle/HOL," in *Proceedings of the 7th ACM SIGPLAN international conference on certified programs and proofs*.

Atzei, N., Bartoletti, M., and Cimoli, T. (2017). "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust* (Berlin, Heidelberg: Springer).

Beckert, B. (2018). Formal specification and verification of Hyperledger Fabric chaincode. *Proc. Int. Conf. Formal. Eng. Methods.*

Bernauer, A. (2023). *Daml: a smart contract language for securely automating real-world multi-party business workflows*. arXiv preprint arXiv:2303.03749.

Bhargavan, K. (2016). "Formal verification of smart contracts: short paper," in *Proceedings of the ACM workshop on programming languages and analysis for security* Vienna, Austria.

Bliudze, S. (2015). "Formal verification of infinite-state BIP models," in *Automated technology for verification and analysis: 13th international symposium, ATVA 2015* (Shanghai, China: Springer International Publishing). October 12-15, 2015, Proceedings 13.

Boogaard, K. (2018). A model-driven approach to smart contract development. *MS thesis*.

Bouali, M., Rocheteau, J., and Barger, P. (2009). "Backward reachability analysis of colored petri nets," in *The European safety and reliability conference (ESREL'09)* (Taylor & Francis Group), 3, 1975–1981.

Brent, L. (2018). *Vandal: a scalable security analysis framework for smart contracts*. arXiv preprint arXiv:1809.03981.

Cavada, R. (2014). "The nuXmv symbolic model checker," in *International conference on computer aided verification* (Cham: Springer).

Crawford, S., and Elinor, OSTROM (2005). *A grammar ofInstitutions." understanding institutional diversity, pages137–174*. Princeton: Princeton University Press.

Delmolino, K. (2016). "Step by step towards creating a safe smart contract: lessons and insights from a cryptocurrency lab," in *International conference on financial cryptography and data security* (Berlin, Heidelberg: Springer).

DigitalAsset (2019). *DAMLScript SDK documentation*. Technical Report. Available at: https://docs.daml.com/ (Accessed December 2, 2020).

Dingman, W., Cohen, A., Ferrara, N., Lynch, A., Jasinski, P., Black, P. E., et al. (2019). Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework." *Int. J. Networked Distributed Comput.* 7, 121–132. doi:10.2991/ijndc.k.190710.003

Duo, W., Huang, X., and Ma, X. (2020). "Formal analysis of smart contract based on colored petri nets." *IEEE Intell. Syst.* 35, 19–30. doi:10.1109/mis.2020.2977594

Dwivedi, V., Pattanaik, V., Deval, V., Dixit, A., Norta, A., and Draheim, D. (2021). Legally enforceable smart-contract languages: a systematic literature review. *ACM Comput. Surv. (CSUR)* 54 (5), 1–34. doi:10.1145/3453475

Frantz, C. K. (2013). PRIMA 2013: principles and practice of multi-agent systems, 8291.

Frantz, C. K., and Nowostawski, M. (2016). From institutions to code: towards automated generation of smart contracts. *Proc. - IEEE 1st Int. Work. Found. Appl. Self-Systems, FAS-W* 2016, 210–215.

Garamvölgyi, P. (2018). "Towards model-driven engineering of smart contracts for cyber-physical systems." *2018 48th annual IEEE/IFIP international conference on dependable systems and networks workshops (DSN-W)*. IEEE.

Grishchenko, I., Maffei, M., and Schneidewind, C. (2018). "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of security and trust: 7th international conference, POST 2018, held as part of the European joint conferences on theory and practice of software, ETAPS 2018* (Springer International Publishing). Thessaloniki, Greece, April 14-20, 2018, Proceedings 7.

Hu, K., Zhu, J., Ding, Y., Bai, X., and Huang, J. (2020). Smart contract engineering. *Electronics* 9 (12), 2042. doi:10.3390/electronics9122042

Huynh-The, T., Thippa, R. G., Weizheng, W., Gokul, Y., Pasika, R., Quoc-Viet, P., et al. (2023). *Blockchain for the metaverse: a review*. Future Generation Computer Systems.

Jani, S. (2020). *Smart contracts: building blocks for digital transformation*. Indira Gandhi National Open University.

Jiang, Bo, Liu, Ye, and Chan, W. K. (2018). "Contractfuzzer: fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*.

Kalra, S., Seep, G., Mohan, D., and Subodh, S. (2018). "Zeus: analyzing safety of smart contracts," in NDSS, 1–12.

Kaur, G. (2023). "Smart contracts and DeFi security and threats," in *Understanding cybersecurity management in decentralized finance: challenges, strategies, and trends* (Cham: Springer International Publishing), 91–111.

Kordestani, A., Oghazi, P., and Rana, M. (2023). Smart contract diffusion in the pharmaceutical blockchain: the battle of counterfeit drugs. *J. Bus. Res.* 158 (2023), 113646. doi:10.1016/j.jbusres.2023.113646

Kristensen, L. M., and Christensen, S. (2004). "Implementing coloured Petri nets using a functional programming language." *Higher-order symbolic Comput.* 17, 207–243. doi:10.1023/b:lisp.0000029445.29210.ca

Le, T. C. (2018). "Proving conditional termination for smart contracts," in *Proceedings of the 2nd ACM workshop on blockchains, cryptocurrencies, and contracts*.

Lesimple, N., and Martin, J. (2020). *Exploring deep learning models for vulnerabilities detection in smart contracts*.

Li, X., Wang, L., Xin, Y., Yang, Y., and Chen, Y. (2020). Automated vulnerability detection in source code using Minimum intermediate representation learning." *Appl. Sci.* 10, 1692. doi:10.3390/app10051692

Liao, J.-W. (2019). "SoliAudit: smart contract vulnerability assessment based on machine learning and fuzz testing," in *2019 sixth international conference on Internet of Things: systems, management and security (IOTSMS)* (IEEE).

Lijie, C., Tao, T., Xianqiong, Z., and Schnieder, E. (2012). Verification of the safety communication protocol in train control system using colored Petri net. *Reliab. Eng. Syst. Saf.* 100, 8–18. doi:10.1016/j.ress.2011.12.010

Lin, S.-Yi, Zhang, L., Li, J., Ji, L. l., and Sun, Y. (2022). A survey of application research based on blockchain smart contract. *Wirel. Netw.* 28 (2), 635–690. doi:10.1007/s11276-021-02874-x

Liu, Su, and He, X. (2015). PIPE+ verifier-A tool for analyzing high level petri nets. *SEKE*.

Liu, Z., and Liu, J. (2019). "Formal verification of BC smart contract based on colored petri net models," in *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)* (IEEE), 2.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016b). Making smart contracts smarter. *ACM*, 254–269. doi:10.1145/2976749.2978309

Luu, L., Chu., D. H., Olickel, H., Saxena, P., and Hobor, A. (2018). "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (Vienna, Austria.

Luu, L., Chu., D. H., Olickel, H., Saxena, P., and Hobor, A. (2016a). "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 254–269.

Magazzeni, D., McBurney, P., and Nash, W. (2017). Validation and verification of smart contracts: a research agenda. *Comput. 50* 9, 50–57. doi:10.1109/mc.2017.3571045

Mavridou, A. (2019). "VeriSolid: correct-by-design smart contracts for ethereum," in *Financial cryptography and data security: 23rd international conference, FC 2019, frigate bay, st. Kitts and nevis* (Springer International Publishing). February 18–22, 2019, Revised Selected Papers 23.

Mavridou, A., and Laszka, A. (2017). *Designing secure Ethereum smart contracts: a finite state machine based approach*.

Momeni, P., Wang, Yu, and Samavi, R. (2019). "Machine learning model for smart contracts security analysis," in *2019 17th international conference on privacy, security and trust (PST)* (IEEE).

Mustafa, I. (2023). "Decentralized oracle networks (DONs) provision for DAML smart contracts," in *International congress on blockchain and applications* (Cham: Springer Nature Switzerland).

Nehai, Z., Piriou, P.-Y., and Daumas, F. (2018). "ModelChecking of smart contracts," in *IEEE international conference on BC* (Halifax, Canada.

Nielsen, J. B., and Spitters, B. (2020). ""Smart contract interactions in Coq." formal methods," in *FM 2019 international workshops: porto, Portugal* (Springer International Publishing). October 7–11, 2019, Revised Selected Papers, Part I 3.

Nikolić, I. (2018). "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th annual computer security applications conference*.

Palmonari, M., and Pasquale, MINERVINI (2020). Knowledge graph embeddings and explainable AI. *Knowl. Graphs Explain. Artif. Intell. Found. Appl. Challenges* 47, 49.

Panduwinata, F., and Yugopuspito, P. (2019). "BPMN approach in BC with hyperledger composer and smart contract: reservation-based parking system," in *2019 5th international conference on new media studies (CONMEDIA)* (IEEE).

Park, D. (2018). "A formal verification tool for Ethereum VM bytecode," in *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*.

Popper, N. (2016). *A hacking of more than \$50 million dashes hopes in the world of virtual currency*. (The New York Times), 17.

Sen, L., Luo, X., and Chen, Z. (2017). "Combining theorem proving and model checking in the safety-critical software development through translating event-B to SMV" in *MATEC web of conferences* (EDP Sciences), 128.

Silviu, O. (2023). "An overview of security issues in smart contracts on the blockchain." education, research and business technologies," in *Proceedings of 21st international conference on informatics in economy (IE 2022)* (Singapore: Springer Nature Singapore).

Singh, A., Parizi, R. M., Zhang, Q., Choo, K. K. R., and Dehghantanha, A. (2020). Blockchain smart contracts formalization: approaches and challenges to address vulnerabilities. *Comput. Secur.* 88, 101654. doi:10.1016/j.cose.2019.101654

Sun, T., and Wensheng, Yu. (2020). A formal verification framework for security issues of blockchain smart contracts. *Electronics* 9 (2), 255. doi:10.3390/electronics9020255

Sun, X., Tu, L., Zhang, J., Cai, J., and Wang, Y. (2023). ASSBert: active and semi-supervised bert for smart contract vulnerability detection. *J. Inf. Secur. Appl.* 73 (2023), 103423. doi:10.1016/j.jisa.2023.103423

Tann, W. J. (2018). *Towards safer smart contracts: a sequence learning approach to detecting security threats*. arXiv preprint arXiv:1811.06632.

Tsankov, P. (2018). "Securify: practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*.

Van Dyke Parunak, H., Savit, R., and Riolo, R. L. (1998). "Agent-based modeling vs. equation-based modeling: a case study and users' guide." multi-agent systems and agent-based simulation: first international workshop, MABS'98, Paris, France, . *Proceedings 1*. Springer Berlin Heidelberg.

Wang, S., Lei, Z., Wang, Z., Wang, D., Wang, M., Yang, G., and Gai, K. (2023). "Blockchain applications in smart city: a survey," in *Smart computing and communication: 7th international conference* Cham: Springer Nature Switzerland, 485–494.

Wang, W., Song, J., Xu, G., Li, Y., Wang, H., and Su, C. (2020). Contractward: automated vulnerability detection models for ethereum smart contracts. *IEEE Trans. Netw. Sci. Eng.* 8, 1133–1144. doi:10.1109/tnse.2020.2968505

Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., and Mendling, J. (2016). Untrusted business process monitoring and execution. *Int. Conf. Bus. Process Manag.*, 329–347.

Yang, Z., and Lei, H. (2019). *Fether: an extensible definitional interpreter for smart-contract verifications in coq*. IEEE Access.

Zhang, F. (2016). "Town crier: an authenticated data feed for smart contracts," in *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*.

Zhang, L., Wang, J., Wang, W., Jin, Z., Zhao, C., Cai, Z., et al. (2022). A novel smart contract vulnerability detection method based on information graph and ensemble learning." *Sensors* 22, 3581. doi:10.3390/s22093581

Zhang, L., Wang, Y., Hu, Y., and Au, M. H. (2019). A game-theoretic method based on Q-learning to invalidate criminal smart contracts. *Inf. Sci.* 498, 144–153. doi:10.1016/j.ins.2019.05.061

Zheng, Z., Xie, S., Dai, H. N., Chen, W., Chen, X., Weng, J., et al. (2020). An overview on smart contracts: challenges, advances and platforms. *Future Gener. Comput. Syst.* 105, 475–491. doi:10.1016/j.future.2019.12.019

Zhu, J. (2020). *Formal verification of solidity contracts in event-b*. arXiv preprint arXiv:2005.01261.

Zhuang, Y. (2020). *Smart contract vulnerability detection using graph neural networks*.

Zupan, N. (2020). "Secure smart contract generation based on petri nets," in *BC technology for industry 4.0* (Singapore: Springer), 73–98.