Modelling and Enforcing Access Control Requirements for Smart Contracts

Master Thesis of

Jan-Philipp Töberg

at the Department of Informatics
Competence Center for Applied Security Technology (KASTEL)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr. Bernhard Beckert
Advisor:	M.Sc. Frederik Reiche
Second advisor:	M.Sc. Jonas Schiffl

13. July 2021 – 13. January 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 13.01.2022

. . . . . .

(Jan-Philipp Töberg)

Abstract

Smart contracts are software systems employing the underlying blockchain technology to handle transactions in a decentralized and immutable manner. Due to the immutability of the blockchain, smart contracts cannot be upgraded after their initial deploy. Therefore, reasoning about a contract's security aspects needs to happen before the deployment. One common vulnerability for smart contracts is improper access control, which enables entities to modify data or employ functionality they are prohibited from accessing. Due to the nature of the blockchain, access to data, represented through state variables, can only be achieved by employing the contract's functions. To correctly restrict access on the source code level, we improve the approach by Reiche et al. [1] who enforce access control policies based on a model on the architectural level.

This work aims at correctly enforcing role-based access control (RBAC) policies for Solidity smart contract systems on the architectural and source code level. We extend the standard RBAC model by Sandhu, Ferraiolo, and Kuhn [2] to also incorporate insecure information flows and authorization constraints for roles. We create a metamodel to capture the concepts necessary to describe and enforce RBAC policies on the architectural level. The policies are enforced in the source code by translating the model elements to formal specifications. For this purpose, an automatic code generator is implemented. To reason about the implemented smart contracts on the source code level, tools like *solc-verify* and *Slither* are employed and extended. Furthermore, we outline the development process resulting from the presented approach.

To evaluate our approach and uncover problems and limitations, we employ a case study using the three smart contract software systems Augur, Fizzy and Palinodia. Additionally, we apply a metamodel coverage analysis to reason about the metamodel's and the generator's completeness. Furthermore, we provide an argumentation concerning the approach's correct enforcement.

This evaluation shows how a correct enforcement can be achieved under certain assumptions and when information flows are not considered. The presented approach can detect 100% of manually introduced violations during the case study to the underlying RBAC policies. Additionally, the metamodel is expressive enough to describe RBAC policies and contains no unnecessary elements, since approximately 90% of the created metamodel are covered by the implemented generator. We identify and describe limitations like oracles or public variables.

Zusammenfassung

Ein Smart Contract ist ein auf der Blockchain basierendes Softwaresystem zur dezentralen und unveränderlichen Behandlung von Transaktionen. Aufgrund dieser zugrundeliegenden Unveränderlichkeit können Smart Contracts nach ihrer initialen Veröffentlichung nicht mehr modifiziert werden. Selbst Softwarefehler oder Sicherheitslücken können nicht im Nachhinein behoben werden. Um die Korrektheit und Sicherheit der Contracts vor der Veröffentlichung festzustellen, müssen statische Analysen verwendet werden.

Aufgrund der zugrundeliegenden Blockchain Technologie muss eine Funktion des Smart Contracts genutzt werden, um eine Variable zu modifizieren. Um Zugriffe auf diese Variablen auf Implementierungsebene einzuschränken, erweitern wir in dieser Arbeit den Ansatz von Reiche u. a. [1], welcher Zugriffskontrollrichtlinien auf Architekturebene beschreibt und auf Implementierungsebene durchsetzt.

Diese Arbeit setzt sich als Ziel, modellierte rollen-basierte Zugriffskontrollrichtlinen (RBAC) für Solidity Smart Contracts auf korrekte Weise auf Implementierungsebene zu erzwingen. Dazu erweitern wir das Standard RBAC Modell von Sandhu, Ferraiolo und Kuhn [2] um unsichere Informationsflüsse zwischen Variablen und zusätzlichen Authorisierungseinschränkungen für Rollen. Um die daraus folgenden Richtlinien auf Architekturebene zu modellieren wird ein neues Metamodell entwickelt. Die so modellierten Richtlinien werden auf Implementierungsebene in formale Spezifikationen übersetzt, was die korrekte Umsetzung dieser gewährleistet. Diese Übersetzung wird automatisch in einem Generator implementiert. Um die generierte Implementierung anschließend zu überprüfen, werden Werkzeuge wie *solc-verify* und *Slither* angewendet und erweitert. Abschließend wird der Entwicklungsprozess skizziert, der sich aus diesem Ansatz ergibt.

Um diesen Ansatz zu evaluieren, und so Probleme und Limitierungen aufzuzeigen, wird eine Fallstudie mithilfe der drei Smart Contract Systeme Augur, Fizzy und Palinodia durchgeführt. Zusätzlich wird eine Metamodel Coverage Analyse eingesetzt, um die Vollständigkeit des Metamodels zu verifizieren. Abschließend argumentieren wir für die korrekte Durchsetzung der modellierten Zugriffskontrollrichtlinien bei der Verwendung unseres Ansatzes.

Wie die Evaluation zeigt, kann eine korrekte Umsetzung garantiert werden, solange bestimmte Annahmen eingehalten und Informationsflüsse nicht betrachtet werden. Um dies weiter zu evaluieren wurden Verletzungen der zugrundeliegenden Richtlinien in die Fallstudie eingefügt. Der vorgestellte Ansatz findet 100% dieser händisch hinzugefügten Probleme. Zusätzlich wird gezeigt, dass das entwickelte Metamodel vollständig genug ist, um rollen-basierte Zugriffskontrollrichtlinen zu beschreiben. Außerdem enthält es keine unnötigen Elemente, da ungefähr 90% des Metamodels vom Generator abgedeckt werden. Lediglich bestimmte Limitierungen wie öffentliche Variablen oder die Verwendung von Orakeln wird nicht abgedeckt.

Contents

List of Figures

List of Tables

List of Algorithms & Code Listings

List of Abbreviations

ABAC Attribute-Based Access Control.

ACM AccessControlMetamodel.

API Application Programming Interface.

AST Abstract Syntax Tree.

BHS Binary Hash Storage.

BoD Binding of Duty.

BPMN Business Process Model and Notation.

CFG Control Flow Graph.

CSM Caller-Specific Mapping.

DSL Domain-Specific Language.

EMF Eclipse Modeling Framework.

EVM Ethereum Virtual Machine.

FOL First-Order Logic.

FSM Finite State Machine.

GDPR General Data Protection Regulation.

GemRBAC Generalized Model for RBAC.

GQM Goal-Question-Metric.

IDE Integrated Development Environment.

JML Java Modeling Language.

JSON JavaScript Object Notation.

LoC Lines of Code.

M2M Model-to-Model (Transformation).

M2T Model-to-Text (Transformation).

MDSD Model-Driven Software Development.

MOF Meta-Object Facility.

OCL Object Constraint Language.

PCM Palladio Component Model.

PoS Proof of Stake.

PoW Proof of Work.

RBAC Role-Based Access Control.

SDP Software Distribution Platform.

SMT Satisfiability Modulo Theories.

SoD Separation of Duty.

UI User Interface.

UML Unified Modeling Language.

XACML eXtensible Access Control Markup Language.

XMI XML Metadata Interchange.

XML eXtensible Markup Language.

1. Introduction

Over the past decade, the influence of decentralized software platforms like blockchain increased steadily. They allow for a cryptographically safe linking of data, which makes them suitable for many areas of application, predominantly finance and economics [3]. One addition to these platforms is the usage of smart contracts, which are programs that can be deployed as a block on the blockchain instead of data. These programs are used to algorithmically enhance contracts between at least two parties to help with the automatic fulfillment of the contracts conditions. This approach is mostly used to manage the resources of the blockchain like cryptocurrencies, tokens and generally valuable assets but can also be employed to implement business workflows as well.

Due to the immutable nature of the blockchain platform, data cannot be changed after it is initially submitted. The same property holds true for smart contracts, which cannot be modified after their initial deployment. Therefore, upfront verification of the contract is necessary to detect vulnerabilities and malfunctions. Since no changes can be introduced to already deployed contracts, the contract's correctness needs to be validated as early in the development process as possible. One security aspect is access to the assets managed by the smart contract. These assets are represented using the contract's state variables. Due to the public nature of the blockchain, every address on the chain can access every public function and read all variables. However, changes to the state variables of a contract should be restricted to certain addresses instead of every possible blockchain user.

To address this problem, this thesis creates an approach where role-based access control (RBAC) policies for smart contracts are modelled on the architectural level. This model is employed as a foundation for the generation of formal specifications that correctly enforce these policies on the source code level. For this enforcement, formal verification tools are employed and extended. The original concept was introduced by Reiche et al. [1], who describe policies using the standard RBAC model in combination with the Palladio Component Model (PCM) on the architectural level. Here, *solc-verify* by Hajdu and Jovanović [4] verifies the resulting formal specifications on the source code level.

This thesis expands on the original concept and circumvents the approaches limitations. By employing the PCM as the metamodel to describe smart contracts and their RBAC policies, the authors identify drawbacks when modelling smart contracts and their complex data structures. Therefore, a new metamodel is created by employing the principles for model-driven software development (MDSD) and expanding the Eclipse Modeling Framework (EMF). Additionally, the underlying RBAC model is formally defined and extended to include role authorizations [5] and secure information flows [6]. To cover these policies on the source code level, formal specifications are automatically generated and verified using the tools *solc-verify* [4] and *Slither* [7].

The metamodel and automatic generation we create as well as the formal specifications we propose all follow the goal of correctly enforcing the modelled policies in the Solidity

source code. To achieve this goal, we collect appropriate information to model these policies on the architectural level. Additionally, we define the necessary formal specifications on the source code level. This includes the variables and data types it covers and the underlying formal model it employs. The fulfillment of this goal is evaluated by providing an argument for this correct enforcement as well as employing a case study using three Solidity software systems.

The remainder of this thesis is structured as follows: Chapter 2 begins by explaining the fundamental concepts necessary to understand the approach presented in the remainder of this thesis. These fundamentals include model-driven software development (MDSD), access control, blockchain, smart contracts and the Solidity programming language. We summarize and categorize similar approaches and related work in Chapter 3. Chapter 4 introduces the running example of an auction system. The necessary information to formally describe RBAC policies is collected throughout Chapter 5. Based on this information, the AccessControlMetamodel (ACM) is created in Chapter 6 and Chapter 7 describes how the formal model elements can be expressed on the source code level. In Chapter 8, insecure information flows are identified by employing the *Slither* framework presented in Section 2.4.2.1. To automatically translate ACM instances to the source code level, we implement a generator in Chapter 9. Chapter 10 outlines the process resulting from the application of our approach. To reason about the properties of the approach, we present an evaluation in Chapter 11. It consists of a case study, an argument regarding the enforcement's correctness and a metamodel coverage analysis. Lastly, we conclude the thesis and summarize future research directions in Chapter 12.

2. Foundations

The following chapter summarizes the fundamentals necessary to understand the approach presented in the remainder of this thesis. We begin by introducing MDSD principles in Section 2.1. For this, we describe the Eclipse Modeling Framework (EMF) as a basis for our metamodel in Section 2.1.1 and the addition of constraints using the Object Constraint Language (OCL) in Section 2.1.2. Additionally, we describe how source code can be generated using model transformations in Section 2.1.3. In Section 2.2 access control principles, the general terminology as well as the relevant access control models for our approach are summarized. Since our approach focuses on smart contracts, we introduce them and the underlying blockchain technology in Section 2.3. Afterwards, we talk about the development of smart contracts for the Ethereum blockchain using the programming language Solidity in Section 2.4. To do so, we first introduce relevant keywords and functionality in which Solidity differs to other programming languages like Java in Section 2.4.1. Additionally, we look at three verification tools that are relevant for the enforcement of our approach in Section 2.4.2.

2.1. Model-Driven Software Development

With MDSD, programmers raise the abstraction level for developing software even more in comparison to object-oriented languages. Here, models are handled as first-class artifacts and the center of the development process instead of supporting documents. MDSD allows developers to describe functionality or architecture on an abstract level, thus increasing the value of the developed primary artifacts as well as reducing the amount and rate of changes needed for these models [8]. To achieve this focus on models, MDSD proclaims that "everything is a model" [9, p. 8], which can be used as the basis for transformations between models and for code generation. Both concepts will be explained in more detail in Section 2.1.3.

One problem regarding this formal definition is the lack of a universal definition for *model*, since it is a term with many connotations that can be used in a wide range of fields and domains. One possible, often used definition comes from Stachowiak [10, p. 129], who describes a model as the portrayal or example of a real-world object. This is similar to the definition by Brambilla, Wimmer, and Cabot [9, p. 1-2], who define models as abstractions with two distinct features: reducing the amount of necessary information by focusing only on the relevant aspects and representing an original in an abstract way for systems or people to use. For our purpose the definition by Brambilla, Wimmer, and Cabot suffices and will be used in the remainder of this thesis, since the described abstraction helps us in creating a concise yet complete concept.

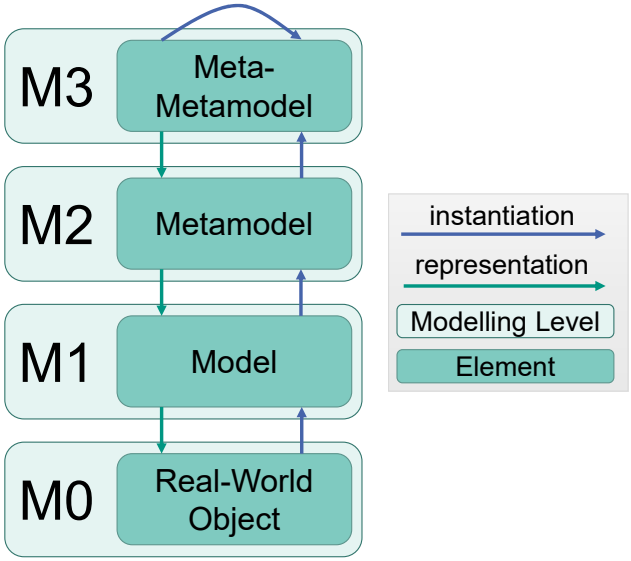

Figure 2.1.: The metamodel hierarchy using four hierarchical levels. Each model is an instance of the model on the next higher level, except for the meta-metamodel, which can describe itself. Simultaneously, each model represents the model from its underlying level. This figure is adapted from [9, Fig. 2.5].

In addition to the models that directly represent original objects, MDSD presents a hierarchy of models through the usage of so called *metamodels*. This hierarchy is visualized in Figure 2.1. A metamodel is a model that is used to represent another model, thus describing how new, valid models can be created. This can be made even more abstract with *meta-metamodels*, which are models that describe metamodels. Theoretically, this abstract hierarchy can be continued endlessly, but, according to Brambilla, Wimmer, and Cabot, practice has shown that most meta-metamodels can be used to describe themselves, thus eliminating the need for more abstract models [9, p. 15-17]. One example for such a meta-metamodel is the Meta-Object Facility (MOF), which is a standard defined by the Object Management Group [11].

To describe such a metamodel, a formal modelling language is used. According to Brambilla, Wimmer, and Cabot [9, p. 63-64], such a language consists of three parts:

1. *Abstract syntax*: The *abstract syntax* describes the general structure of the language. This consists of the different elements and constructs available to describe the domain the language wants to depict. This also includes the properties and relationships between the model elements.

2. *Concrete syntax*: With the *concrete syntax*, the language defines how the elements of the *abstract syntax* are represented. This can be either textual or visual, resulting in either source code like structures or diagrams. For each *abstract syntax*, at least one *concrete syntax* must to be defined.

3. *Static Semantic*: To give meaning to the elements from the language, the *static semantic* is defined. According to Stahl et al. [12, p. 30], the *static semantic* includes

all constraints and rules necessary to achieve well-formed models. This semantic focuses on constraints that cannot be expressed with the *abstract syntax*.

2.1.1. Eclipse Modeling Framework

One framework when applying MDSD principles is the EMF, which uses the support given by Eclipse to facilitate MDSD aspects like the model hierarchy. Eclipse is an open source software framework consisting of many different subprojects to allow for the creation of integrated development environments (IDEs) and other applications like simulators or generators. This creation is done by combining different *plug-ins*, which are the basic units for a component and provide a clearly defined functionality. This functionality can be invoked, reused or extended to create new plug-ins, complete IDEs or tools. To carry out the different plug-ins, Eclipse uses a runtime engine that allows for a flexible handling, deleting the necessity for restarts between the execution of different applications. Lastly, Eclipse provides an extra user interface, known as Platform UI, to enable a customizable UI structure of the developed applications by employing appropriate views, editors or perspectives [13, p. 3-9].

In addition to the Eclipse framework for developing IDEs, the EMF can be used to model the program to develop. It concerns itself with the problems developers face regarding their data model, which represents the data the developers want to work with (e.g. books in a library). This data is translated into programming logic through interfaces, serialized using formats like the Extensible Markup Language (XML) as well as documented and modeled using e.g. the Unified Modeling Language (UML). This separation of concerns would lead the developer to have three different representations of the same underlying data model, which they would need to keep consistent once changes occur. To circumvent this additional effort, EMF describes a central representation as a meta-metamodel called *Ecore*. *Ecore* is based on the essential subset of MOF mentioned in Section 2.1 and evolved to become its implementation [14].

This new model consists of a subset of the constructs used by UML class diagrams, thus focusing on one the most commonly used diagram types [15]. Additionally, *Ecore* uses XML Metadata Interchange (XMI), an XML based representation used to serialize metadata, for persistent storing and data serialization [13, p. 20]. All of these connections are visualized in Figure 2.2. They are necessary to develop models and generate code using the EMF.

If a developer wants to create a new *Ecore* model, they have three possible approaches at their disposal [13, p. 17-19]: First, they can create the required interfaces in Java and add annotations to them. These annotations mark which aspects are part of the *Ecore* model and which can be ignored, as well as specific information needed for more complex data types like lists. For the second option, the developer describes the domain model using an XML schema. Lastly, one can create the *Ecore* model based on an existing UML model. Since *Ecore* describes only a subset of UML, there are multiple ways of getting the *Ecore* model: The developer can either directly edit the UML like its *Ecore*, import the model using an EMF tool or export the model using an UML tool.

After the initial creation of the metamodel is finished and all subsequent changes are made, the *Ecore* model is employed to generate Java code. For the generation, an

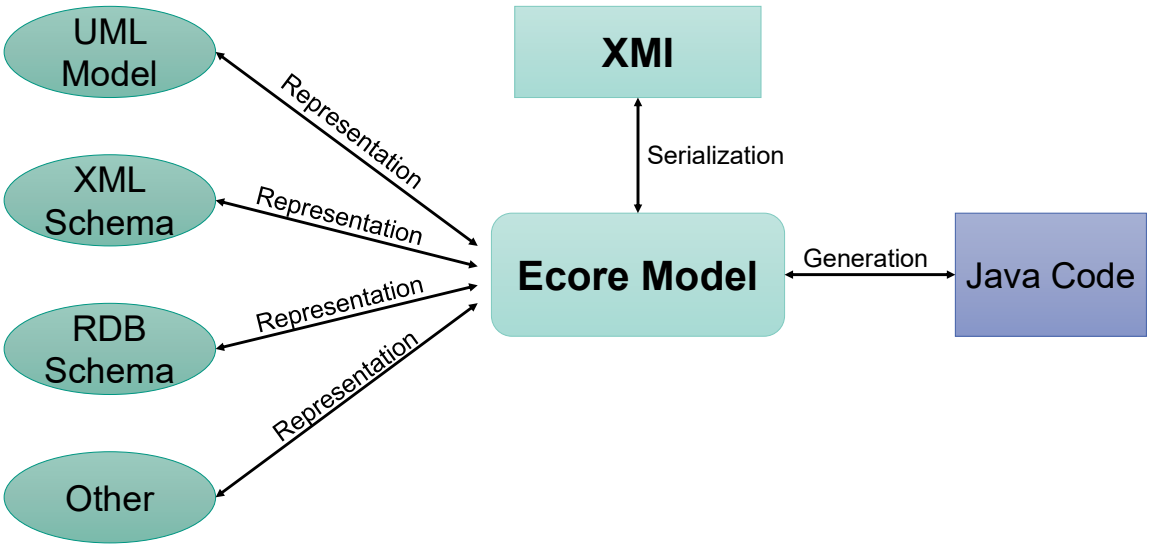

Figure 2.2.: The *Ecore* model and its connections to other model types. This figure is adapted from [13, Fig. 2.5].

additional generator model is needed. This model provides additional information like the necessary prefixes and the resulting file path by decorating the central *Ecore* model. Apart from this extra model, the code generation creates an interface for each *Ecore* class as well as the corresponding implementation, which allows for additional benefits like the generation of Factory methods. These comprehensive possibilities allow for the generation of complicated patterns like bidirectional references [13, p. 24-28].

2.1.2. Modelling Constraints and Conditions

As a developer, the usage of MDSD meta-metamodels can often be limiting due to the different capabilities and functionalities provided by the respecting technique. For example, the *Ecore* meta-metamodel described in Section 2.1.1 is similar to UML class diagrams in its notation and expressiveness. Both metamodels are focused on the static description of the system's structure. Therefore, describing constraints that rely on behavioural or dynamic aspects complicates the formulation of specific requirements.

One approach for circumventing these limitations is the Object Constraint Language (OCL) [16]. Originally developed by IBM as an extension to UML, it is widely integrated and optimized for different models or domain-specific languages (DSL), resulting in a broad area of application. OCL is employed to describe queries on the architectural level, express model manipulations or transform a model into another representation. In general, it is a textual and formal specification language that is statically verified, making it a fitting addition to the MDSD workbench [17]. OCL is used to describe additional rules, patterns or constraints that apply in the metamodel and thus must be adhered to in the resulting concrete instances. To do so, it utilizes *invariants, pre-* and *postconditions* to specify the constraints for the element. An *invariant* is bound to a class object and describes a single rule the element must follow [18]. *Pre-* and *postconditions* on the other hand are

```
context Driver
inv DriverOldEnough:
self.age >= 18
```

Listing 2.1: Example OCL constraint regarding the minimum driver's age

linked to an operation and are part of its method contract, describing the agreement between the methods caller and callee [19, p. 196-198]. *Preconditions* describe expectations regarding the execution state before a method execution, whereas *postconditions* formalize the expectations towards the state after the method. Fulfilling all pre- and postconditions enables formal verification of the methods behaviour as expected.

In general, OCL constraints are always connected to one model element, specified by the `context` keyword, which they mainly refer to. Additionally, a constraint can access all other elements referenced by its context element. As an example, a model includes an element symbolizing the driver of a car with attributes like name, age and address. However, the EMF does not provide the capabilities to add the constraint that the driver must be old enough to legally drive. To prevent a model with an underage driver from being consistent, we can add the constraint seen in Listing 2.1. Here, the `context` states the model element for which the constraint is defined. After that, a number of invariants (`inv`) or pre- and postconditions can be defined and named (`DriverOldEnough`). These can cover the elements attributes as well as its references or use static references to certain model elements.

2.1.3. Model Transformations and Source Code Generation

One key concept of MDSD are *model transformations*. These are used to convert one or more source models into one or more target models [20]. In general, model transformations can be sorted into two categories: *Model-to-model transformations* (M2M) are used to transform one model into another one, whereas *model-to-text transformations* (M2T) are used to translate a model into any kind of text (e.g. source code or documentation) [9, p. 123].

When using M2M transformations, the different models can adhere to the same or to different metamodels, resulting in either *exogenous* or *endogenous* transformations. An example for an exogenous transformation would be the creation of an artifact like the serialization based on the model, so both the source and the target model adhere to different metamodels. An endogenous transformation on the other hand is used to refactor or rewrite a model by changing its structure or contents by still adhering to the same metamodel [9, p. 124]. Both of these transformation types can be correlated to the model hierarchy visualized in Figure 2.1 since an endogenous transformation takes place on the same level of abstraction whereas an exogenous transformation may increase or decrease the model's abstraction.

Model-to-text transformations (M2T) are mainly employed to generate source code based on the source model. This generation does not necessarily cover the whole model or create a completely functional software system, so the concrete implementation is up to the developer. Theoretically, any programming language can be used to generate source

code in text form based on a serialized model [9, p. 141-143]. However, due to specific requirements, special DSLs have been developed to focus on model transformations and their implementation. One example that is used in combination with the EMF is called *Xtend*. This imperative and statically-typed programming language is based on Java. However, to better support M2T transformations, it extends Java's capabilities with concepts like operator overloading or functionally querying the model for information [9, p. 149]. Additionally, it uses template expressions to allow for readable string concatenations, embedding variable properties into static text templates in a structured way [21].

2.2. Access Control

With software systems becoming more complex and being accessible by a wider audience, the need for sound and correct access control is rising. This domain focuses on stopping unwarranted entities from accessing data or functionality they are not allowed to see, use or change. An access control request consist of a *subject*, which can be a user or anything that wants to access a resource, and an *object*, which is the resource or element that should be accessed. According to Samarati and Vimercati [22], a *policy* defines the rules describing which subjects are allowed to access which objects. Each policy is an instance of the *access control model*, which formalizes the aspects that can be considered in the policy. Additionally, the authors define a *mechanism* as a description of the low-level technical functions that are necessary to implement the restraints defined by the policy. This mechanism is implemented using a *system*.

Multiple models exist for the formalization of the required properties, which differ regarding the description of policies and how access is granted or prevented. One model is role-based access control (RBAC), where responsibilities are abstracted from single entities to roles. These roles are based on the organizational structure the system tries to represent. In general, the standard model for RBAC defined by Sandhu, Ferraiolo, and Kuhn [2] uses a layered structure, where new elements are added and capabilities are extended with increasing complexity. Beginning on the basic layer, three different sets are defined: *Users* $\mathbb{U}$ summarizes the subjects that want to access the system and its data, *Roles* $\mathbb{R}$ contains all available abstract roles and *Permission* $\mathbb{P}$ combines the accessible objects with the kind of access (e.g. read, write). To allow subjects to access objects, Definition (2.1) by Sabri [23] introduces the relation UserToRole (UtoR), connecting all users to the roles they currently have. Similarly, the relation RoleToPermission (RtoP) from Definition (2.2) connects the roles to their permissions and is based on the APR relation from [24].

$$\begin{gathered}\mathsf{UserToRole} \subseteq \mathbb{U} \times \mathbb{R} \\ (u, r) \in \mathsf{UtoR} \textit{ iff } \text{user } u \text{ is assigned to role } r\end{gathered} \tag{2.1}$$

$$\begin{gathered}\mathsf{RoleToPermission} \subseteq \mathbb{R} \times \mathbb{P} \\ (r, p) \in \mathsf{RtoP} \textit{ iff } \text{role } r \text{ has permission } p\end{gathered} \tag{2.2}$$

The combination of both allows users to access objects if they have a role with a fitting permission. This is formalized with the hasAccess predicate in Definition (2.3), which takes user u and permission p as input.

$$\mathsf{hasAccess}(u,p) \iff \exists r \in \mathbb{R} : (u,r) \in \mathsf{UtoR} \ \wedge \ (r,p) \in \mathsf{RtoP} \tag{2.3}$$

The first enhancement of this model adds a partial order in form of the hierarchical relation RoleHierarchy (RH) to the roles, which is formalized in Definition (2.4) and taken from [24]. This hierarchy allows for an easier structuring of the roles and can be represented using trees or inverted trees. These structures differ in the way permissions are inherited since it can either be done from the senior to the junior role or the other way round, depending on the concrete definition.

$$\begin{gathered}\mathsf{RoleHierarchy} \subseteq \mathbb{R} \times \mathbb{R} \\ (r_1, r_2) \in \mathsf{RH} \textit{ iff } \text{role } r_1 \text{ is superior to role } r_2\end{gathered} \tag{2.4}$$

The next layer adds additional constraints to roles based on the Separation of Duty (SoD) concept. This concept allows for the mutual exclusion of roles, restricting users from having both roles at the same time. This restriction can be enforced either statically using the model or dynamically by allowing each user to activate or deactivate their roles in a session. The general mutual exclusion is formalized with the relation MutualExclusiveRoles (MER) in Definition (2.5).

$$\begin{gathered}\mathsf{MutualExclusiveRoles} \subseteq \mathbb{R} \times \mathbb{R} \\ (r_1, r_2) \in \mathsf{MER} \textit{ iff } \text{role } r_1 \text{ is mutually exclusive to role } r_2\end{gathered} \tag{2.5}$$

Another more abstract access control model, which was developed more recently than RBAC, is attribute-based access control (ABAC) by Hu et al. [25], where each element holds a number of attributes. The access policy depends on conditions that consider the environment context as well as the attributes of the subject and object. Access is granted once all conditions are met. This can be interpreted as a generalization of the RBAC model, since roles are one example for an attribute that must match between subject and object.

One additional possibility to handle access control that is mainly used by firewalls or anti-virus software, is the application of a black- or whitelist [26]. Both concepts follow similar principles but work in opposite ways. A *whitelist* assumes that every access is forbidden but there are some exceptions. These exceptions are described in the whitelist, which is a central control that must be updated when new systems or user roles are added. The whitelist approach is used in the standard RBAC model, since the permissions always describe positive access [2]. A *blacklist* approach on the other hand assumes that every action and every access is allowed and only the exceptions marked by the blacklist are prohibited. This contains access to certain functions or by certain addresses or roles.

2.3. Blockchain and Smart Contracts

In its earlier stages, blockchain technology sprang from the idea of adding a digital timestamp to documents to have a better overview over the order in which different documents were created. This was soon extended to not only include the current timestamp but information about the preceding document as well. For this, each document, which is also called a *block*, contains a hash value of the previous block's content, chaining

all documents together in an extended linked list. Including this hashed value makes it impossible for anyone to add changes to the previous block, since these changes would also affect the hash values [27]. Due to this native immutability, which makes modifications to a deployed block impossible, blockchain technologies were soon employed as ledgers where transactions are saved in each block [3]. This does not only enable an accessible history of past transactions but also prevents malicious modifications to these transactions.

In their current nature, blockchains are used as the underlying data structure for cryptocurrencies like Bitcoin as well as distributed and decentralized peer-to-peer software systems. In these systems, each node of the network maintains a complete local copy of the main chain. Therefore, the addition of new blocks has to be communicated with the whole network. To handle the creation of new blocks, different consensus algorithms have been created to determine how a new block can be validated and appended [3]. The two main categories of algorithms are called Proof of Work (PoW) and Proof of Stake (PoS), which are explained in more detail in [27]. In general, a *miner* tries to solve a computationally expensive puzzle for which they are awarded the right to add the next block to the chain as well as some of the chains resources (e.g. cryptocurrencies or tokens). This process also mitigates risks, since the creation of a new block is more vulnerable for malicious behaviour then the already chained blocks.

All data that is saved on the chain is available to all nodes in the network, which depends on the deployed type of blockchain network. Access to *permissioned* blockchains is monitored, resulting in a private network used by companies or similar entities. One example is the Hyperledger Foundation [28] that provides open-source blockchain systems for private and public organisations alike, resulting in multiple possible blockchains like Hyperledger Fabric. *Permissionless* blockchains on the other hand are publicly accessible, increasing the anonymity of its users but also their need for cryptographic algorithms [29]. One example for this is Ethereum, which is described in more detail in Section 2.4.

Programs that are deployed on the aforementioned decentralized peer-to-peer software systems are called *smart contracts*. Smart contracts are employed to manage the resources handled by the blockchain, like cryptocurrencies or tokens, by describing a digital contract between two parties. Such a contract consist of execution statements that are stored on the blockchain as immutable transactions [3]. Due to the underlying, immutable blockchain technology, smart contracts cannot be modified once deployed. Since all transactions are duplicated for all nodes in the network, malicious modifications to the deployed contracts are not possible. However, this introduces the biggest difference between smart contracts and other software systems. Once a smart contract is deployed, changes and upgrades cannot be introduced. This increases the need for extensive static analysis of the contracts before their deployments, so possible security risks and erroneous behaviours are found in advance.

2.4. Solidity

Currently, the most prominent platform for smart contracts is Ethereum, a permissionless blockchain technology. According to Zheng et al. [3], Ethereum rewards miners with its own cryptocurrency called *Ether* for mining a new block. To handle the execution of

smart contracts, *Gas* is used as an internal price for executing transactions. Indicating how resource-intensive the execution is and how much Ether must be paid by the developer, gas can be used as an additional metric for developers to optimize.

For the development of smart contracts, Ethereum provides developers with a Turing-Complete instruction set utilized by programming languages like Solidity or Serpent [30]. Once a smart contract is deployed, it is compiled into low-level, stack-based bytecode that is executed on the Ethereum Virtual Machine (EVM). On the source code level, Solidity is an object-oriented programming language with a similar structure to modern Java, where a single contract can be compared to a Java class with the available transactions substituting the Java methods. Solidity's functional differences to other object-oriented programming languages as well as language features relevant for the remainder of this thesis are introduced in Section 2.4.1. Since the blockchain's immutability makes formal verification of smart contracts necessary, three different verification tools are introduced throughout Section 2.4.2.

2.4.1. Language Features

Due to Solidity being a programming language developed solely to implement smart contracts on the Ethereum blockchain, it comes with a bunch of keywords and features that are not provided by other programming languages. In this section, we present all of these capabilities that are relevant for our approach.

To begin, we look at the two keywords `require` and `assert`, which are both applied to verify the correct behaviour of the contract. If the condition specified in a `require` statement fails, the current function terminates and any changes to the contracts state are reversed to the state before the function call. The `assert` keyword works in a similar manner, except that all the remaining gas for this transaction is still consumed. Both keywords are provided to specify pre- and postconditions described in Section 2.1.2 for formal reasoning about the contracts behaviour. The `require` keyword should be applied like a precondition to check for valid inputs at the beginning of a function whereas `assert` describes the exceptions at the end of the function, similar to a postcondition [31].

Another important concept of Solidity are modifiers, which are employed to change or influence the behaviour of the function they annotate. There are two types available: Standard and Self-defined modifiers. Standard modifiers are expressed using special keywords and are helpful to summarize the behaviour of a function. If a function is annotated as `view` for example, it is not allowed to modify the contracts state variables. As the name suggests, a self-defined modifier is implemented by the developer to provide custom functionality. For this purpose, a new function is created with the `modifier` keyword instead of the `function` keyword, whose name can be annotated to the signature of any function in the source code. In the modifiers behaviour, the sequence `_;` marks the behaviour of the original function, so a modifier can be used to add behaviour before or after the original function [32, p. 68-70]. One example can be seen in Listing 2.2, where the `modifier` is used to constrain access to certain functions to the entity that created and thus owns the contract.

Since Solidity smart contracts are used on the Ethereum blockchain, they must be able to handle Ether, the cryptocurrency supported by Ethereum. The `payable` keyword marks

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ModiferExample {
    address payable owner;

    constructor() public {
        owner = payable(msg.sender);
    }

    modifier onlyOwner {
        require(msg.sender == owner, "Only the owner may access.");
        _;
    }

    // Additional functionality

    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}
```

Listing 2.2: Example contract showing the capability of the `modifier` keyword

variables and functions that are used to sent or receive ether. With this, an `address` variable can be marked as an address to which the contract can send money and functions can be marked as being able to receive money and change the contracts balance [32, p. 66]. If a function receives money, it can check the value using `msg.value`. To get the address of the function caller, developers can use the `msg.sender` keyword, which is not limited to work with `payable` functions but can also be used e.g. to check the credentials of a caller. Similarly, the `tx.origin` keyword is used to get the first entity (i.e. external account) starting the chain of transactions instead of the last entity calling a specific function. However, using `tx.origin` for authorization purposes is discouraged since it is easy to spoof [33].

The last property of Solidity we want to discuss is the location where data is stored. Since every variable that is committed to the blockchain consumes memory and gas, thus making the deployment of a contract more expensive, not all variables should be saved on the blockchain. For this, Solidity provides three different keywords that specify the location [32, p. 61-62]:

- `storage`: These variables are permanently saved on the global blockchain as part of the contract and can thus be accessed by every function and contract. All state variables of a contract are storage variables per default.

- `memory`: Variables marked as `memory` variables are only saved during a single transaction in the local memory, which makes them consume much less gas than storage variables [34]. Function parameters and return values are memory variables per default.

- `calldata`: The last type is the forced default for function parameters of external functions and has a constant gas cost. However, these variables are only saved temporarily like the memory variables.

2.4.2. Formal Verification Tools

Due to the immutability of smart contracts, different tools were developed by researchers and practitioners alike to analyze the behaviour and structure of these contracts as early in the development process as possible. By doing so, developers are supported in creating safe and secure smart contracts that mitigate the possibility for attacks and exploitation. These tools employ the capabilities of *formal verification* [19] to reason about the system's properties. For this purpose, the tools provide a formal argument to show that a system fulfills a certain property. Both the system as well as the property are formalized using a certain, underlying logic.

In the following sections, we introduce three of these tools that are especially relevant for our approach. We introduce *Slither*, a static analysis framework, in Section 2.4.2.1. This framework provides different functionalities like failure detection or information collection, thus supporting the developer in better understanding the smart contract and its dependencies. The other two tools, *solc-verify* and *SciviK*, verify the given smart contract using annotations and downstream Satisfiability Modulo Theories (SMT) solvers. We summarize their functionality in Sections 2.4.2.2 and 2.4.2.3. In Section 2.4.2.4, we compare the syntax and functionality of *solc-verify* and *SciviK*.

2.4.2.1. Slither

To reason about program correctness and additional properties, two main categories of approaches have been established and refined: static software analysis and dynamic testing [35]. The main difference between both is how they handle the program under inspection. For static analysis, the analyzer inspects the program without looking at its behaviour during the runtime by focusing on the programming code and all possible execution paths. In contrast, dynamic testing looks for errors and weaknesses by executing the code under certain conditions (e.g. different inputs or changes in the environment). This focuses on a single execution path but analyzes it more thoroughly. Both categories can be seen as complementary and should be used in tandem to achieve an optimal analysis of the inspected program.

One framework used for static analysis of solidity smart contracts is called *Slither*[1] by Feist, Grieco, and Groce [7] and can be used to detect vulnerabilities and code optimizations as well as supporting developers with understanding the source code. To begin the analysis, *Slither* takes the abstract syntax tree (AST) created by the solidity compiler

[1] `https://github.com/crytic/slither` - Last accessed: 04.01.2022

as an input. After inferring additional information from this AST, the source code is translated into an intermediate representation called *SlithIR*, which is based on single static assignment and thus requires that each variable is assigned only one time and defined prior to that. The framework utilizes this representation to run pre-defined detectors written in Python, where each detector focuses on a single weakness or possible optimization. In addition to these detectors, the previously collected information like contract inheritance or function calls are printed.

Currently, the framework provides developers with 74 different detectors and 18 different printers, in addition to a public python API enabling the development of new custom detectors and printers. How this API is structured and used is explained in more detail in Section 8.2.1, where it is applied to create a new custom printer. What all these utilities have in common are the three built-in analyses that provide the foundation for further detection or printing modules. The first analysis focuses on each variable and where they are read and written. This allows *Slither* to examine which variables are influenced by which function as well as finding the set of variables all modified by the same function. The second kind of analysis identifies functions that are not protected through access control patterns like ownership. To find such functions, the framework looks for the usage of the `msg.sender` keyword and checks if it is used directly in a comparison. Lastly, *Slither* analyzes the dependencies of each variable and if they are *tainted* and thus can be influenced by the user accessing the contract. With this, certain vulnerabilities like reentrancy, where a function is called repeatedly (e.g. through recursion) before the previous function invocation is finished, are detected. This vulnerability can lead to funds being transmitted repeatedly before the credit is updated, thus resulting in a possible exploit that allows for an easy drain of complete accounts [33].

2.4.2.2. solc-verify

One tool which allows for the formal verification of smart contracts is *solc-verify*[2] by Hajdu and Jovanović [4], which analyzes the correctness of Solidity smart contracts on a functional basis. To do so, the tool provides an annotation language which can be added to comments in the smart contract to express formal specifications. These can take the form of contract-level invariants, functional pre- and postconditions as well as loop invariants and are formulated by a combination of annotations in first-order logic (FOL) and the Solidity source code. For example, the `require` and `assert` keywords described in Section 2.4.1 are taken into consideration for the formalization as additional pre- and postconditions. One example for the possible annotations are modification specifiers that can be used to allow a specific function to change certain state variables. An example contract with *solc-verify* annotations can be seen in Listing 2.3, where the `increaseXByN` function is annotated with a modification specifier and a postcondition. The modification specifier in line 7 formally expresses that the function is allowed to make changes to the state variable `x`, whereas the postcondition in line 6 states that the value of `x` after the execution is the same as the value before the execution plus the given parameter `n`.

[2] `https://github.com/SRI-CSL/solidity` - Last accessed: 04.01.2022

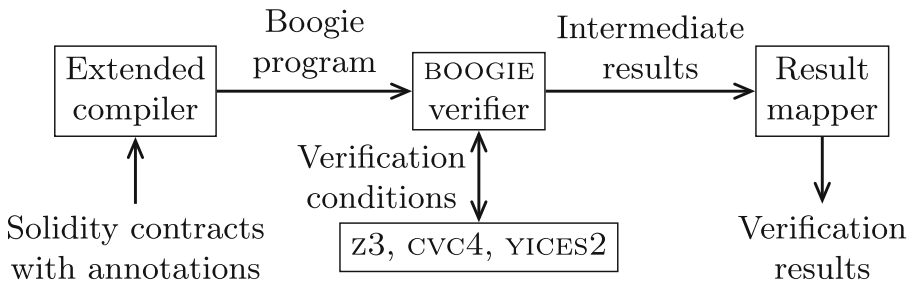

Figure 2.3.: Process of the solc-verify verification tool. This figure is extracted from [4, Fig. 3].

```
pragma solidity >=0.7.0 <0.9.0;

contract SolcVerifyExample {
      uint private x;

      /// @notice postcondition x == (__verifier_old_uint(x) + n)
      /// @notice modifies x
      function increaseXByN(uint n) public {
            x += n;
      }
}
```

Listing 2.3: Example contract showing *solc-verify*'s annotation language

To reason about these specifications, they are translated into the Boogie intermediate verification language. To start the reasoning process visualized in Figure 2.3, the authors extend the default Solidity compiler with their tool. This begins with taking the contract with its annotations and creating a Boogie program by traversing the AST. Based on this program, quantifier-free verification conditions are generated and parsed to external SMT solvers like z3 or CVC4. Afterwards, the results are mapped back to the original contract and communicated to the developer, allowing for fine-grained feedback where possible verification problems have been detected.

Since its original publication, the tool has gained a number of updates that increase its functional capabilities. In one update, the authors add features for managing memory and storage independently and in combination. To do so, they formalize the Solidity semantics into an SMT-based intermediate language that can encode most Solidity elements in a quantifier-free manner. This supports an efficient formalization of smart contracts, which enables deep copy assignments and non-aliasing [34]. In another update, they allow for the formal specification and verification of Solidity events. These events track a specified set of variables and are published, once any of the variables changes. By formalizing events and their influence on the behaviour of a transaction, events are employed to reason about functional correctness together with the already established parts of the Solidity contract [36].

2.4.2.3. SciviK

Similar to *solc-verify*, the formal framework *SciviK*[3] [37] allows for the formal verification of smart contracts through FOL annotations. Again, the annotations are added to the comments in the implementation and allow for the specification of formal contracts, loop and general invariants and checks for certain vulnerability patterns. With the last annotation type, the developer can mark a function to be checked for e.g. the reentrancy error. To reason about the specified properties, *SciviK* translates the Solidity file with its annotations after the compilation to the Yul intermediate representation, annotated with the EVM bytecode. In comparison to *solc-verify*, *SciviK* does not directly reason on this intermediate representation but translates it again to another representation called WhyML, where the annotations are directly inserted into the code. Lastly, this WhyML program is combined with the EVM model to generate the verification conditions that are checked by employing external SMT solvers.

2.4.2.4. Comparing solc-verify and SciviK

In general, both tools provide the same basic functionality, only differing in the used syntax to specify its annotation. These differences are summarized in Table 2.1. The differences between these two tools are the additional functionality and the focus the developers set during the tools development. For example, *solc-verify* was updated to include the specification of Solidity events whereas *SciviK* introduces a special annotation to check for often occurring error patterns like Reentrancy.

However, there are still some functional differences that differentiate both tools. For example, *SciviK* provides the capability to learn loop invariants and the usage of interactive theorem provers. To make use of the learned loop invariants, the developer can use an additional annotation that only specifies the variable(s) of interest. To learn this loop invariant, *SciviK* uses Continuous Logic Networks on the WhyML representation. For the interactive proving of the annotations, *SciviK* uses Coq as the downstream proving tool. However, this process is only started when a specifically marked annotation cannot be solved by an automatic SMT solver. Therefore, a manual prove using Coq is started instead [37].

Solc-verify on the other hand allows for a more detailed description when reasoning about collections of elements. For this, it allows the usage of a sum function to summarize integer collections and an equality predicate allowing for the comparison of complex data types like `mappings` or `structs`.

[3] As of the writing of this thesis, *SciviK* is not publicly available.

Table 2.1.: Differences and similarities in the syntax and the possible functionality of the annotation languages for the two tools *solc-verify* and *SciviK*.

Functionality	*solc-verify* [4]	*SciviK* [37]
Begin annotation	/// @notice <ANNOTATION>	/* <ANNOTATIONS> */
Precondition	precondition <EXPR>	@pre <EXPR>
Postcondition	postcondition <EXPR>	@post <EXPR>
Contract-Level invariant	invariant <EXPR>	@meta <EXPR>
Loop invariant (Manual)	invariant <EXPR>	@inv <EXPR>
Loop invariant (Learned)	**Not possible**	@learn <VARS>
Modification specification	modifies <VAR> [if <COND>]	**Not possible**
Sum over collection	__verifier_sum_(u)int()	**Not possible**
Old value	__verifier_old _<TYPE>() [4]	old <VAR>
Return value	<VAR>[5]	result
Equality predicate	__verifier_eq(<REF1>, <REF2>)	**Not possible**
Universal quantifier ∀	forall (<VARS>) <QUANTEXPR>	forall <VARS>, <QUANTEXPR>
Existential quantifier ∃	exists (<VARS>) <QUANTEXPR>	exists <VARS>, <QUANTEXPR>
Check for error patterns	**Not possible**	@check <PATTERN>
Interactive proving	**Not possible**	@coq <EXPR>
Specifying events	emits <EVENTNAME>	**Not possible**
Assumption	*require(<EXPR>)* [6]	@assume <EXPR>
Assertion	*assert(<EXPR>)* [6]	@assert <EXPR>

[4] As of version 0.5.17, this keyword does not cover all available data types. It is missing for `enum`, `struct`, `string` & `mapping`

[5] To access the return value of a function, the developer must give the returned variable a name in the function declaration. Then, the variable can be referenced by that name.

[6] *solc-verify* takes the `require` and `assert` statements implemented in the contract into consideration instead of providing this functionality with its annotation language [4].

3. Related Work

With the enforcement of modelled access control requirements, our approach can be located at the intersection of different computer science domains. One domain is the application of MDSD to describe the structure and behaviour of smart contracts on an architectural level. Another is the formal verification of smart contract properties and access control systems. Lastly, we consider the combination of access control policies and smart contract systems.

Many approaches overlap in only one of the aforementioned domains with our approach, whereas the number of related approaches that consider all domains we focus on is limited. To distinguish between these approaches, we sort similar work into four main categories that are looked at in more detail in the remainder of this chapter.

The category in Section 3.1 describes approaches that combine formal methods with access control. These approaches use techniques like model checking or theorem proving to reason about formally defined access control models and their properties. Mostly, they verify the formal correctness of the model and its implementation. However, other properties like soundness are also considered.

In Section 3.2 we describe approaches that apply the techniques of MDSD for the development of smart contracts. This takes the form of a metamodel that is introduced or expanded upon to describe smart contracts on an architectural level, generating smart contract code based on an already existing model or the combination of both.

Focusing more on smart contracts, Section 3.3 considers approaches that employ formal methods to evaluate smart contracts and their properties. These approaches may vary in the programming language and underlying blockchain technology that they support. However, most of them allow for the formal verification of smart contracts. Additionally, we describe approaches that reason about security properties other than access control on a source code level using formal specifications.

section 3.4 contains approaches that combine access control with smart contracts. This category can be divided into two subcategories. The category in Section 3.4.1 contains approaches that use the possibilities provided by smart contracts and the blockchain to develop access control systems for other areas of applications. This either results in a tamper-proof system or in a general approach that can be applied by other developers. Section 3.4.2 collects approaches that develop access control for smart contract applications, focusing on their unique properties like immutability. All of these approaches utilize a model to describe the access control requirements but only some of them also consider formal methods.

3.1. Enforcing Access Control Using Formal Methods

To reason about the correctness or security of access control models, many researchers have employed the possibilities provided by formal methods. They combine formal specifications with the elements given by the access control model of their choice to reason about them using theorem proving or model checking. Despite these similarities, the approaches differ in multiple ways. For example, they choose different access control models or focus more on the verification than the specification. Some examples for these approaches will be presented and compared to our approach in this section. Generally, the main difference regarding our approach is that we do not focus solely on the formal specification of the RBAC model we use, but use it as a stepping stone to reason about smart contract security. Additionally, most of the approaches presented here do either not use RBAC as their model or they provide a broader approach, not limited to the restraints provided by smart contracts.

In the approach by Abd-Ali, El Guemhioui, and Logrippo [38], a new metamodel paradigm that enables the description of general access control policies is created. The resulting metamodel elements, which are taken from the UML metamodel, are enhanced using formal semantics written in FOL. Based on these formal semantics, reasoning about security properties is simplified. Despite the creation of a new metamodel, which is also part of our concept, this approach does not focus on RBAC but provides a solution for more general access control policies. Additionally, the approach does not reason about the formal specifications but only provides the basis for other developers to reason about their models more easily.

In contrast to the previous approach, the formal framework developed by Jaidi, Labbene Ayachi, and Bouhoula [24] is mainly focused on the analysis and the risk assessment for RBAC policies. To detect possible vulnerabilities, RBAC models are formalized using sets for users, roles, objects, access modes and permissions as well as their combinations. These sets are employed to describe a policy, which is compared to its implemented realization by calculating their differences using set theory. By calculating these differences, special problems like renamed roles or hidden users are detected. Although the framework has a similar focus in reasoning about vulnerabilities in RBAC policies, it uses different methods to our approach since it relies on set theory. Additionally, the approach has a broader field of application, since it is not limited on a special kind of software system like ours. However, it focuses its assessment on the modelled RBAC policies on the architectural level whereas we also enforce the policies in the implementation on source code level.

Focusing more on the formal verification of RBAC than the previous approach is the solution provided by Mustafa, Drouineaud, and Sohr [39], which employs the Java Modeling Language (JML) to implement a role-based authorization engine. With this approach, the functional behaviour of the engine is specified using sets for users, roles, permissions and sessions as well as their combinations. Based on these sets, the approach utilizes complex equations based on the Z notation to describe functions for the developed engine. These equations are then translated into JML constraints. Despite the approach also using formal contracts to reason about the implementation, it does not cover RBAC policies directly but focuses on an engine for defining and enforcing these policies. Additionally, this engine is

applied to Java projects whereas our approach focuses on smart contracts implemented with Solidity.

A more practical approach was devised by Farah, Gadouche, and Tari [40], who created correct-by-design specifications for RBAC policies using the Event-B framework. These specifications and their declarative as well as behavioural aspects are focused on a health-care information system, allowing practical reasoning about correctness and security properties. Due to its application domain and the usage of the Event-B framework, this approach differs heavily from ours.

Another approach is the formulation of RBAC policies using formal theories like FOL or SMT and evaluating them and their properties using theorem proving. For example, the approach by Arkoudas, Chadha, and Chiang [41], the one by Kolaczek [42] and the approach by Sabri [23] all rely on a subset of FOL to specify RBAC policies. However, their main difference being the tool they use for their reasoning: Arkoudas, Chadha, and Chiang use the SMT solver *Athena*, Sabri uses the theorem prover *Prover9* whereas Kolaczek formalizes the policies using *PROLOG*. Another example relying on a state-based model and formal state transitions is the approach by Yuan et al. [43], which additionally focuses on the verification of special RBAC constraints like SoD from Section 2.2. To reason about the formal specifications, a theorem prover based on the Z specification language is employed. All in all, these approaches are quite similar to our approach in that they use either FOL or theorem proving to reason about RBAC policies and constraints. However, as mentioned earlier, they do not cover specific software systems like smart contracts.

Lastly, we look at the approach by Mondal and Sural [44] which uses timed automata to reason about time-based constraints for RBAC policies. The reasoning about these is done via model checking, focusing on security properties. The main difference to our approach is the focus on the time-based constraints when analyzing RBAC policies, which are only covered as additional conditions in our model. Also, we employ formal contracts and theorem proving to reason about our policies instead of model checking.

3.2. Modelling and Generating Smart Contracts

In addition to formal methods. our approach relies on concepts of MDSD. Therefore, this chapter describes related approaches that employ these concepts to model and implement smart contacts. Most of the described approaches generate a smart contract using M2T transformations. These generated smart contracts are compiled and often include behaviour specified by the corresponding models. This is the main difference between our approach and theirs, since we focus on the definition of a metamodel for correctly enforcing RBAC constraints in combination with formal specifications. We do not concern ourselves with the generation of complete smart contract source code that contains executable behaviour, only creating the smart contract structure with mostly empty functions.

One approach called *Caterpillar* by López–Pintado et al. [29] is an execution engine for smart contract applications. This approach allows developers to send the Business Process Model and Notation (BPMN) of their application through a REST API to be compiled and deployed on the Ethereum blockchain. In comparison to previous work regarding the

execution of BPMN models, *Caterpillar* can handle sub-processes as well as complex BPMN concepts like boundary events or multiple-instance activities. In comparison to our work, *Caterpillar* lacks the means for describing access control requirements as well as using BPMN as an existing notation for their input. This approach also focuses on the smart contract's behaviour instead of the structure and its relation to access control policies.

A framework with similarly exhaustive capabilities to *Caterpillar* is *FSolidM* by Mavridou and Laszka [45], which allows for the modelling of smart contracts using transition-based finite state machines (FSMs). Based on such a FSM, the approach can automatically generate Solidity source code. Additionally, the approach allows for an easy addition of design patterns to provide common functionalities like automatically timed transitions or diminishing the possibility of security vulnerabilities with e.g. access control based on *ownership*. This *ownership* principle saves a single entity for each contract as the owner who has additional permissions to access vulnerable parts of the contract. However, only a single entity has these permissions. The difference to our approach is the coverage of access control requirements since the approach is limited to a single role, the *owner*. Additionally, they employ finite state machines to model the contract and its behaviour, whereas we rely on a metamodel focused on the system's architecture and the RBAC policies.

Focusing on the legal aspects of smart contracts as programmatic enhancements of legal contracts, the unifying metamodel introduced by Ladleif and Weske [46] incorporates aspects like the legal state, legal actions and participants. All these concepts work in combination with each other to describe the operational as well as non-operational aspects of smart contracts. With their unifying model, the authors collect all essential requirements they identified in the literature, which also includes the previously presented approaches *Caterpillar* [29] and *FSolidM* [45]. Our approach differs from the one by Ladleif and Weske since they emphasize the legal nature of smart contracts, whereas we focus more on the technical and structural aspects. Additionally, despite including the concept of roles in the unifying model, it is used to describe the participants in a legal contract instead of a collection of permissions for access control as is done with the RBAC model.

Relying on models with more formalized foundations, Zupan et al. [47] developed a framework for describing and generating smart contracts based on petri nets. The approach focuses on security-by-design to reason about properties like soundness and safety. To create a new smart contract, developers can either start with an empty petri net or use an already existing SysML activity diagram. This model is verified to check for the correct adherence to the aforementioned security properties. Finally, the smart contract is generated by translating each transition into a function and by using tokens to generate variables. The main difference regarding our approach is the missing possibility to model and reason about role-based access control.

Mostly, the metamodels presented in this section rely on a visual representation to describe smart contracts. Another possibility, namely the textual description by applying a DSL, is used by Wohrer and Zdun [48]. Their *Contract Modeling Language* is based on an object-oriented abstraction of smart contracts, structuring each contract similar to an object-oriented class. State variables are described independent from the possible actions, but variables as well as actions are described using clauses that allow for granular description of properties and functionality. For the code generation, a parser creates an

AST, which is traversed to translate each node into a corresponding Solidity element. This approach does not use formal methods for the evaluation of the generated code nor does it consider access control, which are the main differences to our approach.

Similar to the previous example, the approach by Syahputra and Weigand [49] describes a smart contract model as well as an approach to generate source code using a M2T transformation. However, their approach utilizes profiles and stereotypes to extend UML diagrams to be able to describe smart contracts. With this, the authors provide a platform-independent description that is translated into either Ethereum or Hyperledger source code. To support this generation, a smart contracts is described on three levels: Essential, Infological & Datalogical. The main focus of this approach lies on the datalogical level, where an UML profile is applied to describe all elements available in smart contracts. In comparison to this approach, our approach focuses on describing and enforcing access control as well as modelling and generating the smart contract, whereas this approach focuses solely on the code generation. Additionally, this approach allows for the description and generation of smart contract behaviour, whereas our approach focuses on the structure and RBAC policies. Also, they describe smart contracts independent of the underlying blockchain technology, whereas we focus on the Ethereum blockchain.

3.3. Formal Verification of Smart Contracts

Due to the immutability of smart contracts, verifying their behaviour and identifying vulnerabilities is an important aspect. For this, the different approaches rely on formal methods like theorem proving or model checking to evaluate the given smart contracts. The resulting tools either collect a number of vulnerability detectors in a single application or provide developers with a framework for specifying properties that should hold in their contract. Other approaches focus on a single aspect or problem regarding smart contracts and their vulnerabilities.

Some examples for the first category are the tools we describe in more detail in Section 2.4.2, which use formal methods or static analysis to provide developers with the tools to automatically reason about their contract. Both *solc-verify* [4] and *SciviK* [37] create an annotation language enabling the developer to describe formal contracts for their implementation using FOL. However, despite being based on the same principles and ideas, both vary in the functionality they provide. These differences are explained in more detail in Section 2.4.2.4. In contrast, *Slither* [7] provides a static analysis framework enabling the detection of certain security vulnerabilities. Additional information about the implementation that can help developers to better understand the connections and dependencies in their contracts is collected and rehashed.

Another available tool with a different focus then the three mentioned before is *VeriSolid*[1] by Mavridou et al. [50]. This tool provides a formal verification framework using model checking on transition-based models. To allow for the correct-by-design development of smart contracts, the tool extends the *FSolidM* framework from Section 3.2 for modelling and generating contracts with capabilities to reason about the model's properties. The underlying model is based on FSMs, so the reasoning of *VeriSolid* is based on the specified

[1] `https://github.com/anmavrid/smart-contracts` - Last accessed: 05.01.2022

transitions and guards. The main properties it can validate are deadlock freedom, liveness and safety. In comparison to our approach, *VeriSolid* provides only limited coverage for RBAC policies since they support only a single role (*owner*). Also, this role cannot be included in the formal verification about the contracts properties. Our approach on the other hand focuses on the formal enforcement of RBAC policies. So our approach allows for more complete reasoning about RBAC, whereas *VeriSolid* focuses more on other security aspects.

In comparison to the previously presented tools, the approach by Bhargavan et al. [51] consists of *Solidity** and *EVM**. Both are prototypes employed by the authors to demonstrate the general applicability of their approach. These prototypes are based on the functional programming language F*, specifically developed for program verification. The tools are combined to reason about smart contracts on the source code as well as the bytecode level, allowing for different focuses. For example, *Solidity** can verify safety or functional correctness whereas *EVM** can identify bounds on the amount of gas required to complete a call or a transaction.

Another tool that is presented by Alt and Reitwiessner [31] is an extension to the standard Solidity compiler consisting of an SMT-based formal verification module. This module employs bounded model checking to reason about properties like over-/underflow, divisions by zero or assertion fails. This reasoning is done by translating the Solidity code into a combination of different quantifier-free theories like uninterpreted functions or linear arithmetics. Due to current limitations, a complete reasoning about all aspects of the Solidity language is not possible.

Since smart contracts and the blockchain technology are new technologies, there has not been as much research about their formal correctness in comparison to other topics in computer science. Approaches like [52, 53] try to use concepts from other domains to reason about smart contracts. Both approaches translate a given smart contract into a Java program with similar functionality and vulnerabilities and reason about its properties using JML and the verification approach KeY [19]. They both employ JML to specify formal contracts that describe properties and expected behaviour, which are verified using KeY. The difference between both approaches is that Ahrendt et al. [52] translate Solidity smart contracts to Java programs, using the JavaCard functionality of JML to model transactions. Beckert, Schiffl, and Ulbrich [53] on the other hand abstract the Hyperledger Fabric computation model by interpreting the smart contract architecture as a single core machine that employs an underlying database and relies on client requests.

The approach by Marjanovic and Milosevic [54] uses deontic and temporal constraints to model the behaviour of smart contracts, independent of a specific implementation language. To reason about these constraints, they visualize the constraints using newly developed diagram types called role windows and time maps.

Lastly, the approach by Schiffl et al. [55] formally proves the correctness of the Palinodia system from Section 3.4.2 by verifying each function individually using pre- and postconditions. To also prove the security of the whole application, the authors provide a manual proof using temporal constraints. In comparison to our approach, the authors focus on the formal verification of security properties using e.g. *solc-verify* as well. They also consider access control requirements in their proof. However, their reasoning is not based on a

previously defined model and is specific to the Palinodia system. We focus on providing a general approach and only employ Palinodia as a part of our case study in Section 11.3.

3.4. Access Control Combined with Smart Contracts

Due to their immutability and the decentralized technology they run on, smart contracts are used to facilitate access control enforcement. In general, the resulting approaches can be sorted in one of two categories: Approaches that use the cryptographic possibilities provided by blockchain technology and smart contracts to provide a new and better way to control access to applications that are not smart contracts itself. These approaches are described in Section 3.4.1. The other category is summarized in Section 3.4.2 and contains approaches that focus on handling access control for smart contract systems. For this, the approaches keep in mind the difficulties related to the anonymity and immutability of blockchain systems.

3.4.1. Access Control Based on Smart Contracts

The approach by Baby, Honnavalli, and Soman [56] implements a new identity and access management system using the blockchain. For this, smart contracts are combined with mobile devices to give individuals control over their own data. This decentralized approach focuses on privacy protection for individuals where all data is saved on their device and only provided to an interested organization if they permit the access. In addition, the shared data is chosen individually for each identification request, allowing users to only share the minimal amount necessary to use the services by each organisation.

A similar approach is proposed by Davari and Bertino [57], who develop a blockchain-based model for providing consent to the processing of personal data. In contrast to the previous approach, the model adheres to the constraints provided by the General Data Protection Regulation (GDPR). The GDPR is a law by enacted by the European Union in 2018. For this, the approach combines the usage of the Hyperledger Fabric blockchain with external data management systems like MongoDB, where the personal data is stored. Both systems communicate over a REST API that allows operations that are described using an extension to the Extensible Access Control Markup Language (XACML). This whole process complies to the aforementioned GDPR.

Focusing on a different aspect of access control that can be supported by smart contracts, Hirotsugu and Tetsuya [58] developed an approach to handle inference attacks based on a metamodel. During an inference attack, a malicious user can get information they are not allowed to have by inferring it from locations they are permitted to access. To prevent this kind of information leakage, the approach models the access control dependencies in the form of a hypergraph. A hypergraph is a generalization of a graph where a number of vertices can belong to a single edge. To scan this graph for information flows and possible leakages, the approach uses a coloring algorithm. Additionally, it saves the access history for each user encrypted on the blockchain. This enables the approach to evaluate which information the current user already possesses and if access to more information should be granted or not.

In comparison to our solution, the presented approaches differ mainly in the way that they do not provide access control or identity management for smart contract systems but rather use the possibilities given by this technology.

3.4.2. Access Control for Smart Contracts

With *DistU*, Khan et al. [59] present an extended usage control model for permissioned blockchains like Hyperledger Fabric. Their system focuses on two concepts: Decision continuity and attribute mutability. For the first concept, the approach constantly monitors the resources after access is provided to intervene when malicious behaviour occurs and enforce the access policies. With attribute mutability, the attributes of the accessing subject or the accessed object can be changed dynamically based on the granted permissions. This can be used to change the state of the accessed object so not all available information can be accessed. By implementing both of these concepts, *DistU* can be used for a dynamic enforcing of RBAC policies on permissioned blockchains. However, there are many differences regarding our approach. For one, this approach neither uses MDSD nor formal methods. It focuses solely on the security aspects of RBAC through constant monitoring and dynamic changes to the attributes, whereas our approach focuses on an enforcement based on a architectural model. Additionally, the system is limited to permissioned blockchain systems like Hyperledger Fabric, whereas our approach is developed for the permissionless Ethereum blockchain.

Similar to the previous example, the access control possibilities provided by the OpenZeppelin [60] framework do not rely on a predefined architectural model or formal methods. In their approach, the developers provide special classes with structures to check for the *ownership* of a smart contract or if other addresses possess a certain role necessary for the access. However, in comparison to our approach the developers use only a single role called *owner*, which is a limitation in comparison to the RBAC requirements we handle in our approach.

Another approach by Amaral de Sousa, Burnay, and Snoeck [61] describes *B-MERODE*. This is an extension to the *MERODE* method for designing and implementing interorganizational information systems. Their method employs MDSD and business processes to generate smart contracts concerning cross-organizational collaboration like supply chains. To do so, the approach extends the three layer architecture of the original *MERODE* approach with two new layers: One responsible for handling permissions and access control and one for managing all blockchain-specific operations. One advantage of this approach is flexibility, since it can manage participants and permissions dynamically and independent of the underlying blockchain technology. Our approach on the other hand focuses on the Ethereum blockchain. Despite this advantage, this approach does not use RBAC but an ownership-based permission system. Additionally, it does not use formal methods for the verification and enforcement of access control requirements, despite claiming that it would be possible.

One approach that also uses RBAC for describing access control requirements is *Blockchain Studio* by Mercenne, Brousmiche, and Hamida [62], which is an extension to the smart contract modelling and generation tool *Caterpillar* described in Section 3.2. The extension adds the possibility to model roles, users and organisations with their respective

connections. This means that each organization contains a list of users and roles in addition to a connection between users and their roles inside of the organization. Finally, each function is extended by describing the roles that are allowed to access it. These permissions are checked at runtime using an additional smart contract that maps addresses to users and roles. Despite providing a similar model to our approach, one important difference is the level of detail. This approach only considers access to functions whereas our approach explicitly models variables as well. Additionally, this approach is able to generate a running smart contract based on their model due to extending *Caterpillar* whereas our approach focuses on the formal verification of the access control requirements, thus only modelling and generating the contracts structure.

The approach by Stengele et al. [63] is called *Palinodia* and employs smart contracts to manage access control for binary integrity protection. With this, the authors enable users to verify binary files supposedly containing software updates. For this, the approach lets developers manage their software and maintainers manage their software distribution platform (SDP), while administering an identity management that verifies the role for each user inside of the system. Since we identified their approach as a proper use case for our work, we describe it in more detail in Section 11.3.3. This approach allows for the handling of smart contract access control on the source code level. However, it does not cover the architectural level or the enforcement of security properties like correctness or safety.

Lastly, the approach by Reiche et al. [1] focuses on the generation of formal specifications based on a previously defined model to realize RBAC policies. To describe the requirements on the architectural level, the approach extends the Palladio Component Model (PCM) to allow for the definition of Solidity functions and variables as well as their connection to the different roles. The approach generates a formal contract for each function in the annotation language of *solc-verify* from Section 2.4.2.2 as well as function stubs for later implementation of the remaining functionality. This work describes the original concept our thesis is based on. In our thesis, we circumvent the limitations and problems found by the authors. The main limitation is the extension of the PCM to describe RBAC policies on the architectural model. To mitigate this, we create a new metamodel specifically designed to support the goal of correctly enforcing the RBAC requirements. Additionally, the formal specifications they generate are not complete and e.g. evaluations regarding complex data structures are still missing.

4. Running Example: Auction

An auction system is used as a running example throughout this thesis. Originating in the Solidity documentation [64], this system is based on the real-world scenario of an auction where different entities bid on items that are for sale. In the documentation, two versions employing the underlying blockchain technology are distinguished: In the open version, all participants see who is bidding what amounts of money on which item. In the blind version on the other hand, the participants for the auction are kept secret.

The version introduced by the Solidity documentation [64] consists of a single smart contract representing the sale of a single item. During a predefined bidding period, every user can send money as their bid. In the open version, this money is sent directly and thus can be examined by every other user, whereas the blind version only sends a cryptographically safe hash of the money. If a new bid is identified by the system as higher then the currently highest bid, the highest bidder is updated. After the bidding period is finished, the money of the highest bidder is collected and all other bidders get their invested money back.

An extension to the version of the Solidity documentation by Reiche et al. [1] adds two more roles to the open version. By calling a publicly accessible function of an *Auction-Management* contract, any entity creates a new auction, represented by a *SingleAuction* contract. For this contract, the creator entity gets the *seller* role that defines the auctions runtime as well as collecting the money after the auction ends. The second new role is a *manager* that can shutdown all auctions in the case of an emergency (e.g. a security breach in the contract). Similar to the version from the documentation, all subjects with the role *bidder* are allowed to bid their money as long as the auction is still active.

To further specify the extension by Reiche et al. [1], we add an additional role marking the currently *highest bidder*. This role is the only one allowed to collect the sold item after the auction is finished. Additionally, the *bidder*s transfer their money directly by sending the amount with the *bid* transaction instead of staking the amount which is only collected at the end. To accommodate for this, any *bidder* can withdraw the money they are currently bidding. To add temporal constraints, the auction can only be closed by the *seller* a number of days after it was created. An earlier shutdown can only be started in the case of an emergency. This version is visualized in Figure 4.2 and a manual implementation is provided in Appendix A.1.

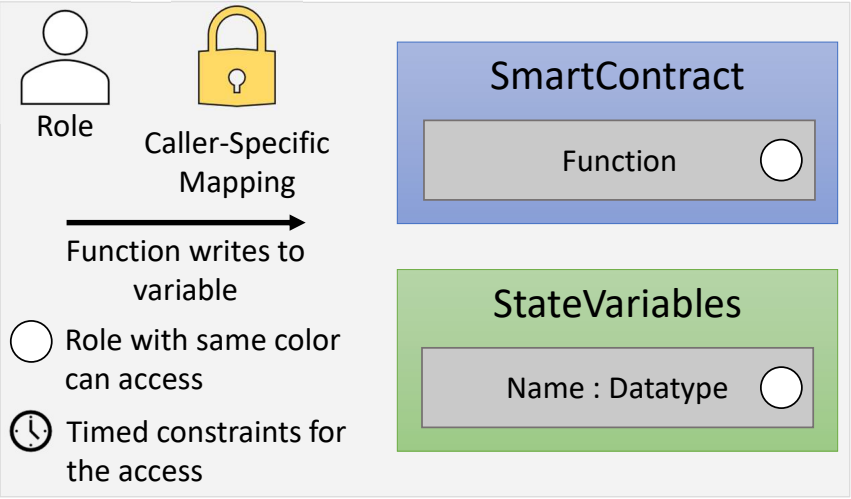

Figure 4.1.: Explanation of the elements employed to describe the roles, smart contracts, functions and state variables in the use case visualizations.

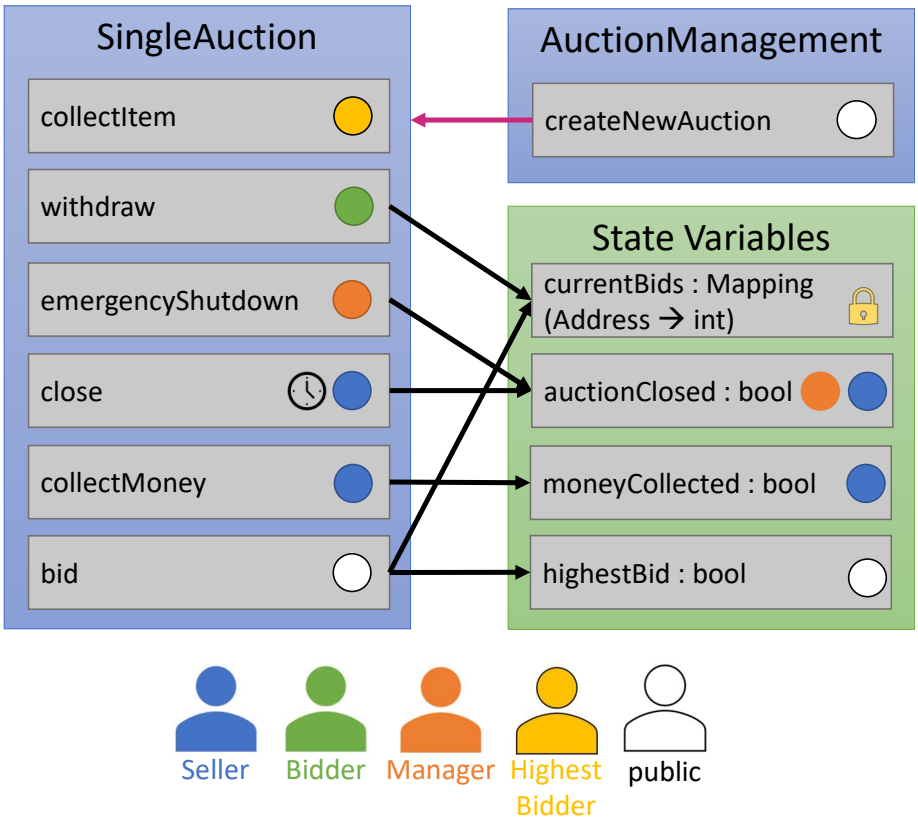

Figure 4.2.: Visualizing the two contracts *AuctionManagement* and *SingleAuction* for the auction use case. This figure employs the visual elements described in Figure 4.1.

5. Defining Role-Based Access Control Policies for Solidity Smart Contracts

Before an architectural model to describe smart contract RBAC policies is created, we summarize the necessary information for describing these policies. This includes a formal definition of the underlying RBAC model as well as adapting this formalization to cover elements from the smart contract domain like state variables. Additionally, this formal model is extended to enable a more concise description of RBAC policies.

Section 5.1 describes the differences our domain introduces to the standard RBAC model by Sandhu, Ferraiolo, and Kuhn [2] from Section 2.2. Additionally, the remaining model elements are expanded upon using the authorization constraints by Ben Fadhel, Bianculli, and Briand [5]. Section 5.2 defines the differences between modifying and influencing access to state variables. Also, it explains the resulting changes and additions regarding the RBAC model from Section 5.1. In addition to the formal RBAC model, Section 5.3 introduces Solidity and smart contract specific elements that are considered when discussing access control for smart contracts. This includes Solidity elements responsible for handling monetary assets as well as the `mapping` data structure.

5.1. Formal Specification of Role-Based Access Control Policies

Beginning with the underlying access control model, the domain we work in adds constraints to the standard flat RBAC model by Sandhu, Ferraiolo, and Kuhn [2] from Section 2.2. There is no need to explicitly model the *Users* $\mathbb{U}$ since we focus on smart contracts running on the Ethereum blockchain. As explained in Section 2.3, this blockchain is permissionless, meaning access to it is publicly available, thus the software architect cannot know all users trying to access their contract beforehand. In addition, we aim at creating an approach usable to statically verify and enforce the RBAC policies for smart contracts. This verification is performed before its deployment, so the dynamic handling of users is not relevant on the architectural level. However, in the generated parts on the source code level, the assignment of addresses to roles is still handled. We call anything represented by these addresses, which can be a subject or another smart contract, an *entity* in the remainder of this thesis.

Another aspect of the standard RBAC model by Sandhu, Ferraiolo, and Kuhn [2] that needs adaptation before it can be included is the *Permission* set $\mathbb{P}$. It describes how an object can be accessed (e.g. read, write). However, in the context of smart contracts, there is no way to restrict reading access to state variables, so keeping data secret relies on

encryption [30]. Therefore, restricting reading access to state variables is irrelevant for permissionless blockchains like Ethereum, so this thesis focuses on RBAC policies for modifying state variables. To accommodate for this, we cover access to state variables directly in the model with the set $\mathbb{S}$ from Definition (5.1), instead of relying on the abstraction described with the permissions set. These state variables can only be modified through transactions [30], which are represented by the functions of the contract. Therefore, the set $\mathbb{F}$ from Definition (5.2) summarizes all functions.

$$\mathbb{S} : \text{Set of all state variables} \tag{5.1}$$

$$\mathbb{F} : \text{Set of all functions} \tag{5.2}$$

To express the permission to call a certain function, we introduce the RoleToFunction (RtoF) relation in Definition (5.3). This relation is a restatement of Definition (2.2) by Jaidi, Labbene Ayachi, and Bouhoula [24] and connects one role from $\mathbb{R}$ to one function from $\mathbb{F}$.

$$\begin{gathered}\mathsf{RoleToFunction} \subseteq \mathbb{R} \times \mathbb{F} \\ (r, f) \in \mathsf{RtoF} \textit{ iff } \text{role } r \text{ is allowed to call function } \texttt{f}\end{gathered} \tag{5.3}$$

Similarly, a role is allowed to access a state variable if the corresponding tuple exists in the RoleToVariable (RtoS) relation from Definition (5.4).

$$\begin{gathered}\mathsf{RoleToVariable} \subseteq \mathbb{R} \times \mathbb{S} \\ (r, s) \in \mathsf{RtoS} \textit{ iff } \text{role } r \text{ is allowed to make changes to state variable } \texttt{s}\end{gathered} \tag{5.4}$$

Similar to the standard RBAC model [2], both the RtoF and the RtoS relation model access explicitly. Therefore, only permitted access is included and unwarranted access are modelled implicitly by the abstinence of corresponding tuples. However, if no role is allowed to access, functions and state variables are handled differently. A variable without any accessing roles should not be accessed by anyone, neither inside nor outside the contract. On the other hand, if no role is permitted to access a function, we define that function to be publicly available to any entity. This decision is based on the consideration that a function inaccessible to any entity is not usable in the smart contract domain.

Based on these sets and relations, Reiche et al. [1] introduce Formula (5.5). This formula applies FOL to describe the central constraint that the model must adhere to: If a role r is allowed to access the function `f` but is not allowed to access the state variable `s`, `f` may not modify `s`. For example, the system for the running example from Chapter 4 ensures that the *bidder* cannot access any function that changes the `auctionClosed` variable.

$$\forall f \in \mathbb{F}, s \in \mathbb{S}, r \in \mathbb{R} : ((r, f) \in \mathsf{RtoF} \ \wedge \ (r, s) \notin \mathsf{RtoS}) \implies \neg\mathsf{doesModify}(f, s) \tag{5.5}$$

To model and verify the doesModify predicate, we introduce the FunctionToVariable (FtoS) relation describing access to state variables by functions in Definition (5.6).

$$\begin{gathered}\mathsf{FunctionToVariable} \subseteq \mathbb{F} \times \mathbb{S} \\ (f, s) \in \mathsf{FtoS} \textit{ iff } \text{function } \texttt{f} \text{ is allowed to make changes to state variable } \texttt{s}\end{gathered} \tag{5.6}$$

With the FtoS relation, the doesModify predicate can be defined as seen in Definition (5.7). For the moment, this predicate solely depends on the direct changes made through function `f` to state variable `s`. However, Section 5.2 introduces additional constraints that extend the considerations collected in this predicate.

$$\mathsf{doesModify}(f, s) \iff (f, s) \in \mathsf{FtoS} \tag{5.7}$$

Further formalizing RBAC constraints introduces Formula (5.8). This formula states that a role r that is permitted to access a state variable `s` also must be able to access at least one function `f` that does modify `s`. For example, the *seller* is allowed to modify the `auctionClosed` variable to close the auction. However, if the *seller* is not allowed to access the `close` function, it may never conduct any changes to `auctionClosed`. By verifying this formula, possible gaps in the architectural model can be detected. These gaps show violations regarding the underlying RBAC model since permitted access is not enforced in the model. Therefore, resolving these violations increases the conciseness of the modelled system.

$$\forall r \in \mathbb{R}, s \in \mathbb{S} : (r, s) \in \mathsf{RtoS} \implies (\exists f \in \mathbb{F} : (r, f) \in \mathsf{RtoF} \ \wedge \ \mathsf{doesModify}(f, s)) \tag{5.8}$$

As the authors of the standard RBAC model [2] describe, increasing its expressiveness introduces additional constraints like role hierarchy and mutual exclusion into the model. These allow for a more distinct description of policies. However, as Ben Fadhel, Bianculli, and Briand [5] describe, not all constraints that can occur in real-world RBAC systems are covered by the standard model. To mitigate this limitation, the authors introduce the Generalized Model for RBAC (GemRBAC) as a unifying framework. This framework adds eight authorization constraints to the standard flat RBAC model. In the following, we describe these constraints and whether they are **relevant** or *irrelevant* for our domain:

1. **Prerequisite** - A role X can have another role Y as its precondition. For example, if an entity is assigned to the *highest bidder* role in our running example from Chapter 4, it must have the *bidder* role first. This constraint can be applied to our domain, despite the lack of formally represented users, since the role assignment on the source code level is managed as well. This enables a more complete generation, since the role assignment can be created once and should not be adapted by the developer in later stages of development. This constraint is formally defined with the Prerequisite (Pr) relation in Definition (5.9).

$$\begin{gathered}\mathsf{Prerequisite} \subseteq \mathbb{R} \times \mathbb{R} \\ (r_1, r_2) \in \mathsf{Pr} \textit{ iff } \text{role } r_1 \text{ is the prerequisite for role } r_2\end{gathered} \tag{5.9}$$

2. **Cardinality** - There can only exist a constant number of users assigned to a role at once. For example, only a single entity can be the *seller* for an auction from the running example in Chapter 4. Similar to the prerequisite constraint, this constraint can be applied to our domain to increase the extent of the role assignment on the source code level. Therefore, it is enforced on the source code level by creating the corresponding counting variable and checks during the generation.

3. *Precedence & Dependency* - These constraints restrict a users possibility to activate or deactivate a role at runtime. The underlying understanding from the GemRBAC is that each user can choose in each *session* which roles they want to use. Since sessions are not part of our domain and dynamic user to role assignments are not handled on the architectural level, we do not include these constraints.

4. **Role Hierarchy** - This constraint describes the hierarchical connection between roles and was already explained as part of the standard RBAC model [2]. This constraint is included in our domain and the formal specification from Definition (2.4) in Section 2.2 is applied. This relation is transitive, so a junior role also inherits its access permissions from the superior roles of its superior. This transitive connection is formalized in Formula (5.10).

$$\forall r_1, r_2, r_3 \in \mathbb{R} : (r_1, r_2) \in \mathsf{RH} \ \wedge \ (r_2, r_3) \in \mathsf{RH} \implies (r_1, r_3) \in \mathsf{RH} \tag{5.10}$$

5. **Separation of Duty (SoD)** - Roles, users and permissions can be mutually exclusive. Since our model is limited to roles, we only add SoD constraints for the mutual exclusion of roles. This can be viewed in Definition (2.5) in Section 2.2. An example for this constraint from the auction use case are the *seller* and *bidder* role, who are never assigned to the same entity.

6. *Binding of Duty (BoD)* - This complement to SoD constraints can only be defined for permissions. In that case, two permissions p_1 and p_2 are linked together, forcing any role assigned to p_1 to also be assigned to p_2. Since our access control model does not include permissions, this constraint is not included.

7. *Role Delegation & Revocation* - A user is permitted to assign any of its role to another user who meets the roles requirements. This assignment can also be revoked if necessary. Since we do not cover the assignment of roles on the architectural level, these constraints are not included.

8. **Context** - Specifying further restrictions regarding the access based on either the current time or the current location. Adding the possibility to add more access constraints to the model is sensible. However, from the two example types, only temporal constraints make sense in the blockchain domain.

5.2. Modifying and Influencing Access to Variables

In addition to the formalized constraints in Section 5.1, our domain is susceptible to violations to secure information flows defined by Terauchi and Aiken [6]. The authors define a secure information flow as a program, where the final values for the low-security variables do not depend on the initial values of the high-security variables. In the domain of smart contracts, we define these secure information flows in correspondence with the roles from the formal model. Here, an information flow is secure if all variables that cannot be modified by a role are independent from the variables a role can modify. So violations

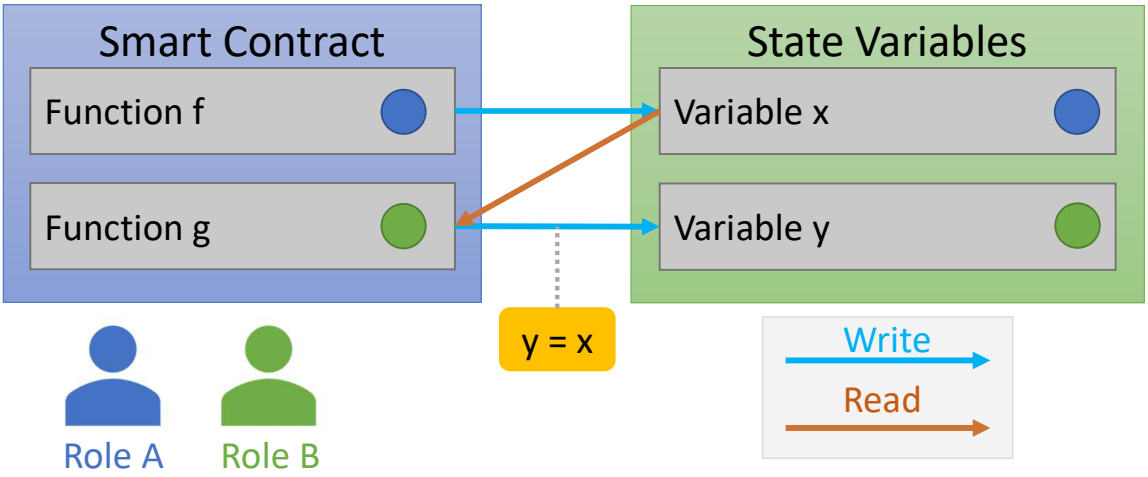

Figure 5.1.: Despite all access being modelled correctly (e.g. Role *A* cannot call function `g` and thus not modify state variable `y`), role *A* influences state variable `y` through state variable `x` despite not being permitted to access it directly, since the value of `x` is assigned to `y` throughout the execution of function `g`. This means that an insecure information flow exists where `x` *influences* `y`. This visualization employs the model elements introduced in Figure 4.1.

to these secure information flows allow an entity to indirectly modify the value of another variable it is not permitted to access. In the following, the connection between two variables leading to insecure information flows is also described as an *influence* relation.

Figure 5.1 visualizes an example containing an influence relation. In this scenario, two roles (*A* and *B*) and one smart contract are modelled. The smart contract consists of two state variables (`x` and `y`) and two functions (`f` and `g`). The model also defines that role *A* is allowed to call `f` and modify `x` and role *B* can access `g` and modify `y`. If `f` is the only function modifying `x` and `g` is the only function modifying `y`, then there are no violations to the modelled RBAC policies since no malicious behaviour can occur. However, connections between both variables on the source code level lead to an insecure information flow when `x` influences `y` in function `g` or vice versa.

Essentially, this influence can be achieved in multiple ways that are summarized in Listing 5.1. The *direct* form of influence occurs when the value of `x` is assigned to `y` in line 2. This enables role *A* to influence the value of `y` despite not being allowed to access it directly. Another way is *indirect* influence, where the value of `y` depends on the value of `x` through a condition instead of direct assignment. This type of influence can be examined in line 4. In addition, both kinds of influence can also be *transitive*, when the value of `y` depends on another variable `z` that is directly or indirectly influenced by `x`, as presented in line 6 and 7. The running example from Chapter 4 contains an indirect influence since the *seller* can close the auction after a certain point in time by modifying the `auctionClosed` variable. However, bidding and thus changes to the `currentBids` mapping are only permitted while the auction is still open.

To distinguish between modifying and influencing access in the formal RBAC model, we split the RoleToVariable relation from Definition (5.4) in Section 5.1 into two new

```
// direct influence
y = x;
// indirect influence
y = (x == 5? 10: -10);
// transitive influence
z = x;
y = z; // y = x
```

Listing 5.1: Example Solidity code showing the different kinds of influence

relations. The RoleModifiesVariable (RmS) relation from Definition (5.11) maps roles to the variables they are permitted to modify, whereas the RoleInfluencesVariable (RiS) relation from Definition (5.12) connects the roles to the variables they are allowed to influence. The connection between these two relations is specified in Formula (5.13). We define modifying access to be a stronger restriction compared to influencing access. So each role that is allowed to modify a variable is also allowed to influence it.

$$\begin{gathered}\mathsf{RoleModifiesVariable} \subseteq \mathbb{R} \times \mathbb{S} \\ (r, s) \in \mathsf{RmS} \textit{ iff } \text{role } r \text{ is allowed to modify the state variable } \mathtt{s}\end{gathered} \tag{5.11}$$

$$\begin{gathered}\mathsf{RoleInfluencesVariable} \subseteq \mathbb{R} \times \mathbb{S} \\ (r, s) \in \mathsf{RiS} \textit{ iff } \text{role } r \text{ is allowed to influence the state variable } \mathtt{s}\end{gathered} \tag{5.12}$$

$$\mathsf{RoleModifiesVariable} \subseteq \mathsf{RoleInfluencesVariable} \tag{5.13}$$

Additionally, we consider role hierarchy when checking for modifying or influencing access to state variables as well as functions. To do so, we introduce three new predicates that check if any superior role is allowed to access. All of them are defined recursively to consider the complete hierarchical structure instead of only the direct superiors. Definition (5.14) covers modification access to variables by considering direct access and access through a superior role. The Definition (5.15) does the same but regarding influencing access. For functions, Definition (5.16) provides the same check considering the RoleToFunction relation.

$$\begin{aligned}\mathsf{modifiesVar}(r, s) \iff & (r, s) \in \mathsf{RmS} \quad \vee \\ & (\exists r_h \in \mathbb{R} : (r, r_h) \in \mathsf{RH} \ \wedge \ \mathsf{modifiesVar}(r_h, s))\end{aligned} \tag{5.14}$$

$$\begin{aligned}\mathsf{influencesVar}(r, s) \iff & (r, s) \in \mathsf{RiS} \quad \vee \\ & (\exists r_h \in \mathbb{R} : (r, r_h) \in \mathsf{RH} \ \wedge \ \mathsf{influencesVar}(r_h, s))\end{aligned} \tag{5.15}$$

$$\begin{aligned}\mathsf{callsFunc}(r, f) \iff & (r, f) \in \mathsf{RtoF} \quad \vee \\ & (\exists r_h \in \mathbb{R} : (r, r_h) \in \mathsf{RH} \ \wedge \ \mathsf{callsFunc}(r_h, f))\end{aligned} \tag{5.16}$$

These predicates enable the update of Formula (5.8) from Section 5.1 to also cover function calls and indirect influence. The result can be examined in Formula (5.17). Here, for any role r that is permitted to modify a variable s, r must be able to call at least one function that implements this modification access.

$$\begin{aligned} \forall r \in \mathbb{R}, s \in \mathbb{S} : \ & \mathsf{modifiesVar}(r, s) \\ & \implies (\exists f \in \mathbb{F} : \mathsf{callsFunc}(r, f) \ \wedge \ \mathsf{doesModify}(f, s)) \end{aligned} \tag{5.17}$$

To permit certain variables to influence other variables, the relation in Definition (5.18) is introduced. It connects the first variable of the tuple with the second by declaring that the first is permitted to influence the second. The type of influence is irrelevant for this relation. The relation is also transitive, as described in Formula (5.19).

$$\begin{gathered} \mathsf{VariableToVariable} \subseteq \mathbb{S} \times \mathbb{S} \\ (s_1, s_2) \in \mathsf{StoS} \textit{ iff} \text{ an information flow from state variable } s_1 \\ \text{to state variable } s_2 \text{ exists} \end{gathered} \tag{5.18}$$

$$\forall s_1, s_2, s_3 \in \mathbb{S} : (s_1, s_2) \in \mathsf{StoS} \ \wedge \ (s_2, s_3) \in \mathsf{StoS} \implies (s_1, s_3) \in \mathsf{StoS} \tag{5.19}$$

By utilizing the VariableToVariable (StoS) relation from Definition (5.18), an additional constraint on the RBAC model is specified. If a role r is allowed to modify a state variable x but is not allowed to influence another state variable y, it is prohibited for x to influence y. This constraint is formalized in Formula (5.20).

$$\begin{aligned} \forall r \in \mathbb{R}, s_1, s_2 \in \mathbb{S} : & (\mathsf{modifiesVar}(r, s_1) \ \wedge \ \neg \mathsf{influencesVar}(r, s_2)) \\ & \implies (s_1, s_2) \notin \mathsf{StoS} \end{aligned} \tag{5.20}$$

To exploit unwarranted access to illegally influence state variables, a malicious role can use modification access to a variable by employing the VariableToVariable relation. Additionally, a function call to another function that is responsible for changing the prohibited variable can also lead to unwarranted influence. To formally describe these function calls, the FunctionToFunction (FtoF) relation in Definition (5.21) is added to the model. Similar to the VariableToVariable relation from Definition (5.18), the FunctionToFunction relation is transitive, as described with Formula (5.22). For each tuple, the first function describes the caller whereas the second expresses the callee.

$$\begin{gathered} \mathsf{FunctionToFunction} \subseteq \mathbb{F} \times \mathbb{F} \\ (f_1, f_2) \in \mathsf{FtoF} \textit{ iff} \text{ function } f_1 \text{ calls function } f_2 \end{gathered} \tag{5.21}$$

$$\forall f_1, f_2, f_3 \in \mathbb{F} : (f_1, f_2) \in \mathsf{FtoF} \ \wedge \ (f_2, f_3) \in \mathsf{FtoF} \implies (f_1, f_3) \in \mathsf{FtoF} \tag{5.22}$$

Integrating these function calls into our formal RBAC model allows for updates to the doesModify predicate from Definition (5.7). As we mentioned in Section 5.1, the predicate is extended to consider function calls. The updated version is represented in Definition (5.23) and is applied in Formula (5.17).

$$\begin{aligned}\mathsf{doesModify}(f,s) \iff & (f,s) \in \mathsf{FtoS} \quad \vee \\ & (\exists f_c \in \mathbb{F} : (f,f_c) \in \mathsf{FtoF} \ \wedge \ \mathsf{doesModify}(f_c,s))\end{aligned} \tag{5.23}$$

With all relevant relations defined, additional constraints can be specified. Similar to Formula (5.20), Formula (5.24) models that any role that can access a function `f` also must be permitted to access all functions called by `f`. In reality, a function call can depend on a case analysis, leading to cases where the accessing role would not need the permission to call all called functions. However, formally defining these cases leads to information from the source code level being modelled explicitly in the underlying formal RBAC model, thus violating the separation of concerns.

$$\forall r \in \mathbb{R}, f, f_c \in \mathbb{F} : (\mathsf{callsFunc}(r,f) \ \wedge \ (f,f_c) \in \mathsf{FtoF}) \implies \mathsf{callsFunc}(r,f_c) \tag{5.24}$$

Based on these new relations, Formula (5.5) by Reiche et al. [1] can be expanded to also cover indirect access by roles. The result is displayed in Formula (5.25). Here, the doesInfluence predicate is an extension of the doesModify predicate from Definition (5.23) to also consider variable influence and function calls.

$$\begin{aligned}\forall f \in \mathbb{F}, s \in \mathbb{S}, r \in \mathbb{R} : (\mathsf{callsFunc}(r,f) \ \wedge \ \neg\mathsf{influencesVar}(r,s)) \\ \implies \ \neg\mathsf{doesInfluence}(f,s)\end{aligned} \tag{5.25}$$

$$\begin{aligned}\mathsf{doesInfluence}(f,s) \iff & (f,s) \in \mathsf{FtoS} \quad \vee \\ & (\exists s_i \in \mathbb{S} : (f,s_i) \in \mathsf{FtoS} \ \wedge \ (s_i,s) \in \mathsf{StoS}) \quad \vee \\ & (\exists f_c \in \mathbb{F} : (f,f_c) \in \mathsf{FtoF} \ \wedge \ \mathsf{doesInfluence}(f_c,s))\end{aligned} \tag{5.26}$$

Formula (5.25) formalizes that for each role r that can access a function `f` and not influence a variable `s`, `f` is not allowed to influence `s`. This doesInfluence predicate checks for three cases in which a function `f` influences a variable `s`:

1. Checking if the tuple $(f,s) \in \mathsf{FtoS}$. With this check, direct access by the function to the variable is excluded. This was already considered in the previous definition for the doesModify predicate from Definition (5.7) in Section 5.1.
2. Validating indirect access through variable influence by searching for a single variable s_i that is modified by the function `f` and that influences `s`.
3. Analyzing the doesInfluence predicate recursively for all functions that `f` is allowed to call. This analysis is similar to the recursive definition of the upgraded doesModify predicate from Definition (5.23).

Throughout this section we introduced multiple definitions and formulae. However, not all of them are enforced directly by our approach since some describe supporting aspects. All in all, the presented definitions describe relations, sets and predicates that are combined to form the formulae the underlying RBAC model must fulfill. The relevant formulae our approach aims to enforce are Formulae (5.17), (5.20), (5.24) and (5.25).

5.3. Covering Solidity and Smart Contract Elements

In addition to the RBAC formalization from Section 5.1 and the differences between modifying and influencing access from Section 5.2, the smart contract domain entails additional aspects relevant for the correct enforcement of RBAC policies. However, handling the access control for these aspects differs from the formal model described throughout this chapter. This includes the `mapping` data structure and the handling of monetary assets through special balance variables.

The `mapping` data structure provided by the Solidity programming language enables developers to map a value to each possible key. Due to this property, the data structure is used in many smart contract applications to map specific entities to values that only they should access. This is employed to link a monetary value to the address that transferred the amount, which happens for example in the auction use case from Chapter 4. Here, the `currentBids` mapping saves the amount of Ether bid during the auction for each entity.

In the smart contract domain, `mappings` where the developer needs to verify that an entity can only access the value corresponding to its key are important aspects of the underlying programming language. This leads us to include every `mapping` that connects entities to entity-specific values in our approach. We call a `mapping` that fulfills these requirements *caller-specific*. For each caller-specific mapping (CSM) we must ensure that no changes are made to memory locations not associated with the current entity. In the auction use case this would mean that each bidder should only access their current bid and not increase the bid of another entity.

When using Solidity smart contracts, monetary assets (Ether) can be transferred between contracts to use as currency. To keep track of the amount of money a contract has at their disposal, it possesses a `balance` variable. This variable can only be changed by sending or receiving Ether and not through direct assignment. Due to the importance of balance modifications and their connection to monetary assets, access to these variables is restricted as well. To handle these balance modifications in our approach, each function can be explicitly permitted to make a certain type of change to the contract balances.

6. Description of Role-Based Access Control Policies on the Architectural Level

To achieve the goal of enforcing RBAC policies in the Solidity source code based on an architectural description, this architectural descriptions must be created. To enable the automatic generation of source code and annotations, the MDSD workbench is applied. Previous work by Reiche et al. [1] found the PCM to be inappropriate for describing the underlying smart contract architecture, since the PCM focuses on the description of component-based software architecture [65]. Due to the mismatch between smart contracts and components, smart contract specific aspects like addresses or complex data types cannot be modelled correctly. To elude these shortcomings, our approach creates a new metamodel based on the EMF by employing *Ecore* as its meta-metamodel. This new *AccessControlMetamodel* (ACM) covers the information collected throughout Chapter 5.

The ACM is created and presented in Section 6.1. Since the model still includes implicit assumptions regarding its elements, these assumptions are made explicit by OCL constraints, which are explained in Section 6.2.

6.1. Describing the AccessControlMetamodel

After collecting the underlying information in Chapter 5, the metamodel to describe RBAC policies and enable their correct enforcement is created. This AccessControlMetamodel (ACM) utilizes the EMF by implementing the *Ecore* meta-metamodel. Integrating the ACM in the metamodel hierarchy from Section 2.1, it is situated on the M2 layer. Concrete instances of the ACM describe the M1 layer and the real-world scenario the instances capture are placed on the layer M0. This structure is visualized in Figure 6.1.

The ACM is not only based on the *Ecore* meta-metamodel but also relies on the *SolidityMetaModel* [66] and the *Metamodel-Modeling-Foundations* [67]. Both metamodels are defined on the M2 layer. The *SolidityMetaModel* by Dietrich and Reiche [66] is an extensive metamodel for describing Solidity smart contracts using three packages: The *SolidityContracts* package is employed to describe the structure of a single Solidity Contract using e.g. StateVariables, Functions and data Types. To model multiple smart contracts in conjunction with each other, the *SoliditySystem* package is employed. In it, multiple Contract instances are connected to each other through their provided or required functions. Lastly, the *RBAC* package implements the approach by Reiche et al. [1] to enable the basic modelling of RBAC policies by connecting Roles to operations and variables. Since the ACM focuses on

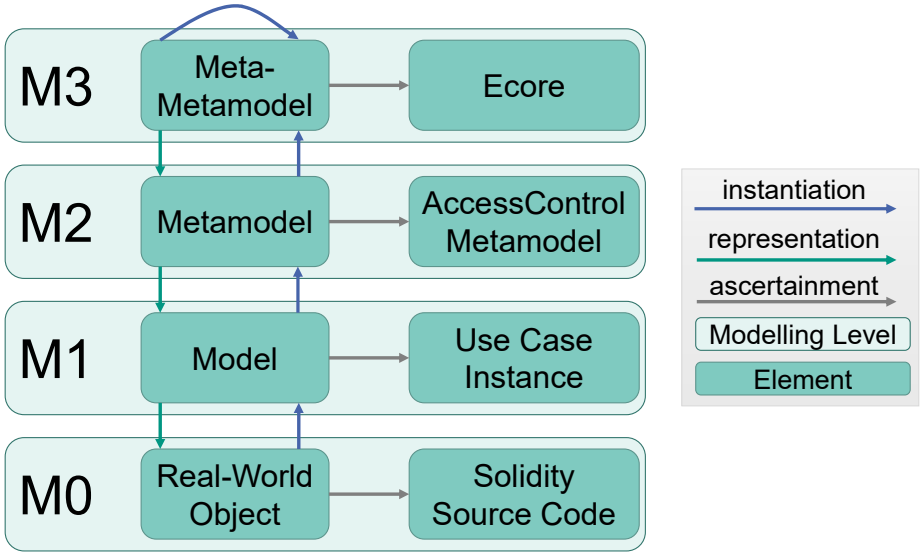

Figure 6.1.: Visualization of the metamodel hierarchy from Figure 2.1. Here, each level is linked to a concrete implementation of its abstract concept, centered on the AccessControlMetamodel.

describing RBAC policies and their source code enforcement, defining the basic structure of the smart contracts is done by employing the *SolidityMetaModel*.

Another referenced metamodel is the *Metamodel-Modeling-Foundations* by Krach and Seifermann [67], which is also employed in the *SolidityMetaModel*. The *Metamodel-Modeling-Foundations* collects commonly use elements that are required to model different use cases [67], like providing a unique id or giving model elements a name. By referencing the *Metamodel-Modeling-Foundations*, elements in the ACM are provided with a unique id and an attribute for defining a name simply by inheriting from the Entity element.

The ACM separates its capabilities into the packages *SmartContractModel* and *AccessControlSystem*. The *SmartContractModel* is employed to describe the structure of the smart contract and can be examined in Figure 6.2. With the *AccessControlSystem*, RBAC policies can be described based on their formal definition from Sections 5.1 and 5.2. This package can be viewed in Figure 6.3. By separating these concepts on the architectural level, the general approach for describing RBAC policies can be adapted for other programming languages than Solidity, increasing the model's reusability.

The *SmartContractModel* package imports the Function, StateVariable, Contract and Type element from the *SolidityMetaModel*. To expand the capabilities of the central Contract element, the AccessControlContract is created by inheriting from the Contract element. To access the created Functions directly through the AccessControlContract, a new containment is added. This simplifies the creation of new functions when modelling the smart contract by enabling their direct connection to the contract. Additionally, the AccessControlContract contains FunctionBalanceModification elements. These elements are employed to cover the handling of monetary assets by restricting changes to the `balance` variables as stated in Section 5.3. Each FunctionBalanceModification references a `function` and models, if that function is allowed to modify the balance of either the function caller, represented by

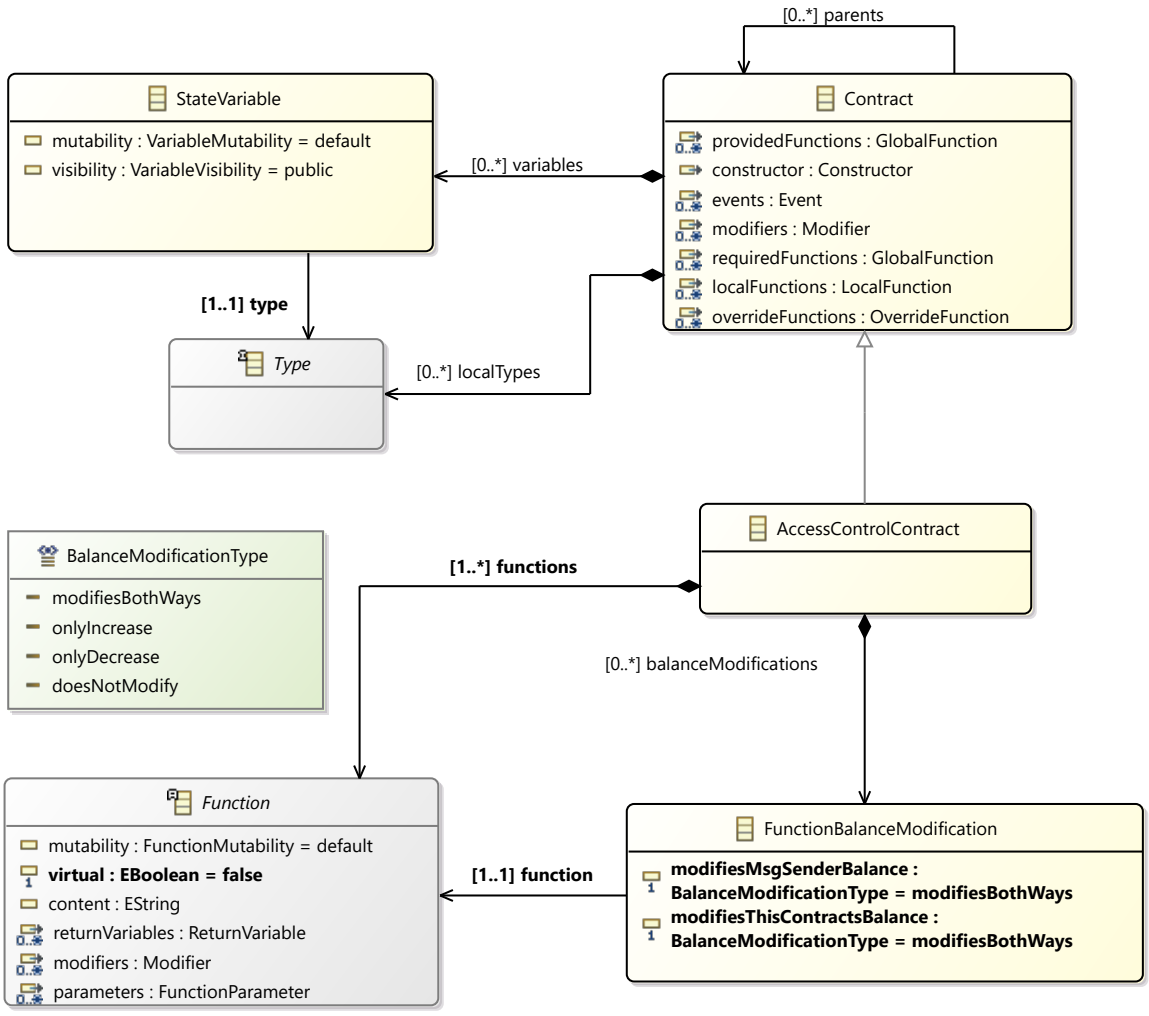

Figure 6.2.: Elements from the ACM to describe Solidity smart contracts. The elements Function, StateVariable, Contract and Type are imported from the *SolidityMetaModel* [66].

the `modifiesMsgSenderBalance` attribute, or the contract it belongs to, represented by the `modifiesThisContractsBalance`. The possible values for both attributes are summarized in the BalanceModificationType enumeration. The four possible values are as follows:

1. `modifiesBothWays` - The `function` is allowed to modify the balance completely. This includes increases as well as decreases.

2. `onlyIncrease` - The `function` is only permitted to increase the balance. This case occurs in the auction example from Chapter 4 for the `bid` function. This function transfers money to the *SingleAuction* contract as a bid, so the contracts balance is increased by that amount.

3. `onlyDecrease` - The `function` is only permitted to decrease the balance. Similarly to the `onlyIncrease` case, this occurs when a *bidder* withdraws previously bid money using the `withdrawMoney` function, decreasing the contracts balance.

4. `doesNotModify` - The `function` is prohibited from modifying the balance in any way possible. This `enum` value describes the default value.

Since `doesNotModify` is the default case, a `FunctionBalanceModification` does not need to be created for `Functions` that are prohibited from modifying the balances.

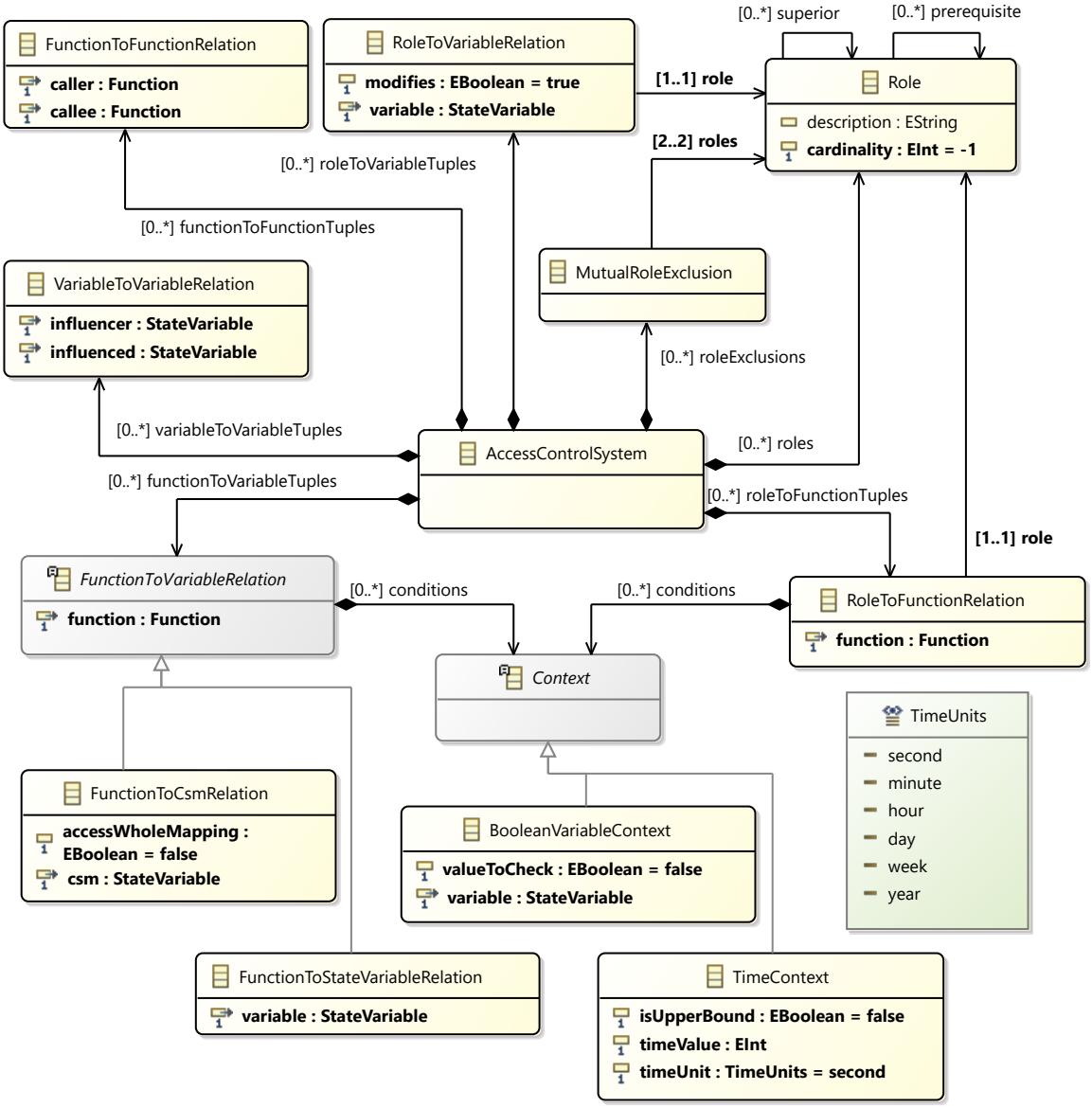

Figure 6.3.: Elements from the ACM to describe RBAC policies. These elements reference the Solidity elements from Figure 6.2 to define and enforce policies based on the formal definitions from Sections 5.1 and 5.2.

To describe and enforce RBAC policies for smart contracts based on the formal definitions from Sections 5.1 and 5.2, the *AccessControlSystem* package is employed. It relies on the *Metamodel-Modeling-Foundations* metamodel to add an identifier and a name to each element, but it does not import any elements from the *SolidityMetaModel*. Only the

ones imported into the *SmartContractModel* are referenced. The central element of the *AccessControlSystem* package is the AccessControlSystem, which describes the top-level container for all other model elements.

The Role element is contained in the top-level container and describes a single role in the system. To provide additional information about that role, a `description` can be added. To implement the authorization constraints described in Section 5.1, additional attributes are defined. The `cardinality` attribute denotes the maximum amount of entities that are assigned to that role at once. The default value of `-1` represents that no maximum number is specified. For example, the amount of *bidder*s in the auction use case from Chapter 4 is not restricted but there can only exist one *highest bidder*. To cover the role hierarchy from Definition (2.4) in Section 2.2, each role references an arbitrary amount of other roles through the `superior` attribute. Similarly, the `prerequisite` relation from Definition (5.9) in Section 5.1 references an arbitrary amount of required roles. To model the SoD constraints described in Section 5.1, the MutualRoleExclusion element is used. It is contained in the top-level container and references exactly two `roles`, which are marked as mutually exclusive. This element directly covers the relation from Definition (2.5).

To connect the three main elements (Functions, StateVariables and Roles), the formal model from Sections 5.1 and 5.2 introduces relations where each tuple connects two of these elements. These relations are directly modelled using the *Element*To*Element*Relation elements. All of these relations are contained in the AccessControlSystem element. Describing that one `role` is allowed to access one `variable`, the RoleToVariableRelation element represents the general RoleToVariable relation from Definition (5.4) in Section 5.1. To distinguish between modifying and influencing access, the boolean attribute `modifies` is utilized. Therefore, a RoleToVariableRelation element where `modifies` is set to `true` represents the RoleModifiesVariable relation from Definition (5.11) in Section 5.2. If `modifies` is set to `false`, the element covers the RoleInfluencesVariable relation from Definition (5.12). Similarly, the RoleToFunctionRelation represents the RoleToFunction relation from Definition (5.3) by linking one `role` to one `function` it may access.

As described in Section 5.3, a caller-specific mapping (CSM) maps an entity to an arbitrary value. Access to this `mapping` should be restricted, so an entity is permitted to only change the value it is connected to. However, this access is only restricted for certain functions. As an example, consider the `currentBids` mapping from the auction use case from Chapter 4. When this mapping is accessed through the `bid` or `withdraw` function, only the caller-specific location is changed. However, when the `emergencyShutdown` is started, all entries in the mapping are modified. To allow for this distinction, the architect differentiates between all other state variables and the CSMs when modelling the FunctionToVariable relation from Definition (5.6) in Section 5.1. Generally, this relation is covered by the abstract *FunctionToVariableRelation* element, which includes the reference to a `function`. This abstract element is modelled concretely using the FunctionToStateVariableRelation element and the FunctionToCsmRelation element. The FunctionToStateVariableRelation connects one `variable` of any type with one `function`, whereas the FunctionToCsmRelation element references a `mapping` variable that is handled like a CSM. To distinguish the access possibilities, the boolean attribute `accessWholeMapping` is used. With this attribute, architects are enabled to enforce the CSM properties on a function to function basis.

To incorporate the temporal authorization constraint from Section 5.1, the abstract *Context* element is added. Elements of this type are added to either RoleToFunctionRelation or FunctionToVariableRelation elements through the `conditions` attribute. One *Context* element describes a single condition that further restricts the modelled access. The kind of restrictions can take multiple forms but due to the scope of our thesis, we create two concrete examples with the BooleanVariableContext and TimeContext. However, future additions are easily created by extending the *Context* element. If a condition has the BooleanVariableContext type, it means that the referenced `variable` must have the modelled `valueToCheck`, otherwise access is prohibited. For example, calling the `bid` function in the auction use case is only possible while the auction is open, which is represented by the boolean variable `auctionClosed`.

By applying the TimeContext, software architects describe that an access must occur either before or after a relative point in time. This point in time is defined in relation to the creation of the smart contract. The `timeValue` describes the amount of time that needs to pass and the `timeUnit` gives the unit, described with the TimeUnits enumeration. In general, the defined units are based on the temporal capabilities of Solidity [32, p. 79]. To differentiate if a TimeContext describes an upper or lower border, the `isUpperBound` attribute is used. For an example, the `close` function from the auction use case is considered. Access to this function is only possible, when the auction was open for at least seven days. Since the auction starts with the creation of the contract, a TimeContext element with `isUpperBound = false`, `timeValue = 7` and `timeUnit = TimeUnits.day` is created.

To model the relations describing function calls and variable influence from Section 5.2, FunctionToFunctionRelation and VariableToVariableRelation elements are applied. Both of them are contained in the AccessControlSystem and underlie a similar structure as the other relation elements by connecting two elements. The FunctionToFunctionRelation is based on Definition (5.21) and defines a function call made by the `caller` to the `callee`. The two referenced Functions do not have to reside in the same instance of the *SmartContractModel*, allowing for the connection of different contracts. In a similar manner, `VariableToVariableRelation` describes the influence relation between two `StateVariables` from Definition (5.18). The `influencer` directly, indirectly or transitively influences the `influenced` variable. However, it is not the only element to model the influence relation since the BooleanVariableContext also describes the indirect dependence between two variables. If it is used as a condition for a FunctionToVariableRelation, the variable referenced by the relation element is indirectly influenced by the boolean variable referenced by the BooleanVariableContext.

6.2. Adding Explicit Constraints to the AccessControlMetamodel

By employing *Ecore* as the meta-metamodel for creating the ACM, our metamodel is susceptible to the same limitations regarding its expressiveness. Since *Ecore* focuses on describing the structure of model elements similar to UML class diagrams, certain rules, conditions or constraints focused on the behaviour cannot be expressed. This includes

```
1  context AccessControlContract
2  inv BalanceModificationsReferenceDifferentFunctions:
3  balanceModifications -> isUnique(function)
4
5  inv NoMoreBalanceModificationsThanFunctions:
6  functions -> size() >= balanceModifications -> size()
7
8  inv BalanceModificationsRegardingThisContractNeedPayableFunction:
9  balanceModifications -> forAll(b | (b.modifiesThisContractsBalance =
   BalanceModificationType::modifiesBothWays or
   b.modifiesThisContractsBalance = BalanceModificationType::onlyIncrease)
   implies b.function.mutability =
   solidityContracts::FunctionMutability::payable)
10
11 inv NoFunctionOverrides:
12 overrideFunctions -> isEmpty()
13
14 inv NoAdditionalLocalFunctions:
15 localFunctions -> isEmpty()
```

Listing 6.1: OCL constraints for the AccessControlContract element

implicit assumptions the developers made during the creation of the metamodel that cannot be added explicitly using the capabilities of the EMF. To circumvent the resulting limitations, a static semantic is described using OCL, which was introduced in Section 2.1. By adding the following OCL constraints, the implicit assumptions about the model elements are made explicit and can be enforced for software architects using the ACM.

For the AccessControlContract element from the *SolidityContractModel* package from Figure 6.2, the five invariants from Listing 6.1 are added. Since every FunctionToBalanceModification element is only allowed to reference a single `function`, the constraint `BalanceModificationsReferenceDifferentFunctions` verifies that all `balanceModifications` reference a different function. Similarly, the `NoMoreBalanceModificationsThanFunctions` invariant limits the amount of FunctionToBalanceModification elements to the amount of Functions. The `BalanceModificationsRegardingThisContractNeedPayableFunction` validates that each Function, for which a FunctionToBalanceModification element specifies that it can modify or increase the contract's balance, has to be declared as `payable`. This enables the Function to handle monetary assets, as explained in Section 2.4.1. The two constraints `NoFunctionOverrides` and `NoAdditionalLocalFunctions`, are added to suppress functionality inherited from the Contract, since these concepts are not part of the domain the ACM aims to support. Adding support for these functionalities would blur the line between the architectural and the source code level.

To correctly enforce the authorization constraints from Section 5.1, the four invariants from Listing 6.2 are added to the Role element. With the `CardinalityIsValid` constraint, it is guaranteed that the cardinality contains a valid value. This can be any positive

```
context Role
inv CardinalityIsValid:
cardinality = -1 or cardinality > 0

inv NoRoleCanBePrerequisiteForItself:
prerequisite -> excludes(self)

inv NoRoleCanBeSuperiorToItself:
superior -> excludes(self)

inv NoRoleInPrerequisiteAndSuperiorSet:
prerequisite->excludesAll(superior)
```

Listing 6.2: OCL constraints for the Role element

```
context MutualRoleExclusion
inv RolesCannotBeInHierarchyOrPrerequisite:
roles -> excludesAll(roles -> first().prerequisite -> union(roles ->
last().prerequisite) -> union(roles -> first().superior) -> union(roles
-> last().superior))
```

Listing 6.3: OCL constraints for the MutualRoleExclusion element

integer or the default value of `-1`, representing an unlimited amount of possible entities. The next constraints relate themselves to the role-to-role connections, validating that the role does not depend on itself. This means that the role cannot be set as a `superior` (`NoRoleCanBeSuperiorToItself`) or `prerequisite` (`NoRoleCanBePrerequisiteForItself`) of itself. Similarly, the Role is prohibited from referencing the same Role as a `prerequisite` and a `superior` with the `NoRoleInPrerequisiteAndSuperiorSet` constraint. Since the `prerequisite` requires a specific role before the assignment and each entity is also assigned to the role's `superior`, defining the same role for both leads to problems.

Similar to the constraints on the Role element, the `RolesCannotBeInHierarchyOrPrerequisite` constraint from Listing 6.3 on the MutualRoleExclusion element is used to validate the authorization constraint from Section 5.1. It verifies that the two `roles` marked as mutually exclusive do not depend on each other through the other connections like `superior` or `prerequisite`.

For the abstract *FunctionToVariableRelation* element the `ForbidTimeContextConditions` constraint from Listing 6.4 is defined. This constraint limits the available conditions to define for this relation to only BooleanVariableContext by excluding TimeContexts. This exclusion is implemented because temporal constraints on the modification of a StateVariable through a Function cannot be enforced using the verification tools from Section 2.4.2.

To add an ascertainment for the abstract *FunctionToVariableRelation*, a FunctionToCsmRelation element is modelled to connect a `function` to a `csm`. However, not all `mapping` variables fulfill the necessary requirement to be handled as a CSM. Since entities are

```
context FunctionToVariableRelation
inv ForbidTimeContextConditions:
conditions -> forAll(c | c.oclIsTypeOf(TimeContext) <> true)
```

Listing 6.4: OCL constraints for the FunctionToVariableRelation element

```
context FunctionToCsmRelation
inv VariableTypeNeedsToBeMapping:
csm.type.oclIsTypeOf(soliditycontracts::Mapping)

inv MappingKeyTypeNeedsToBeAddressOrAddressPayable:
let map = csm.type.oclAsType(soliditycontracts::Mapping)
in map.keyType.oclIsTypeOf(soliditycontracts::PrimitiveType) and
(map.keyType.oclAsType(soliditycontracts::PrimitiveType).type =
soliditycontracts::PrimitiveTypeEnum::address or
map.keyType.oclAsType(soliditycontracts::PrimitiveType).type =
soliditycontracts::PrimitiveTypeEnum::address_payable)
```

Listing 6.5: OCL constraints for the FunctionToCsmRelation element

represented on the source code level by their address, a CSM must accommodate for this in its key data type. In Solidity, two data types fulfill this requirement: `address` and `address payable` [32, p. 53]. The `address payable` data type extends the `address` data type by marking the represented entity with the `payable` keyword as an entity that can receive money, as we explained in Section 2.4.1. To validate that the referenced `csm` fulfills this requirement, the OCL constraints from Listing 6.5 are added. These constraints describe two invariants that both must be fulfilled. The invariant `VariableTypeNeedsToBeMapping` validates that the referenced variable has the `mapping` data type. The additional invariant `MappingKeyTypeNeedsToBeAddressOrAddressPayable` checks if the referenced `csm` has the required key data type. As mentioned before, this can either be a Solidity `address` or an `address payable`. If both of these invariants are fulfilled, it is ensured that the referenced `mapping` variable is enforced as a CSM on the source code level.

Since the FunctionToStateVariableRelation is another sub element to the abstract *FunctionToVariableRelation* element, it inherits the `ForbidTimeContextConditions` constraint. Additionally, it adds the `VariableTypeIsNoMappingWithAddressAsKeytype` invariant from Listing 6.6. This invariant describes the complement to the constraints validating the FunctionToCsmRelation element. With this invariant, variables that fulfill the requirement to be handled as a CSM are excluded. This exclusion leaves only the FunctionToCsmRelation as a valid modelling element to describe CSMs.

The BooleanVariableContext is restricted by the `VariableTypeNeedsToBeBoolean` constraint from Listing 6.7, which is employed to validate that the referenced `variable` has the datatype `bool`. Otherwise the functionality connected to this model element isn't enforced since other data types cannot be compared to the boolean `valueToCheck` on the source code level.

```
context FunctionToStateVariableRelation
inv VariableTypeIsNoMappingWithAddressAsKeytype:
not (variable.type.oclIsTypeOf(soliditycontracts::Mapping) and
variable.type.oclAsType(soliditycontracts::Mapping)
.keyType.oclIsTypeOf(soliditycontracts::PrimitiveType) and
(variable.type.oclAsType(soliditycontracts::Mapping)
.keyType.oclAsType(soliditycontracts::PrimitiveType).type =
soliditycontracts::PrimitiveTypeEnum::address or
variable.type.oclAsType(soliditycontracts::Mapping)
.keyType.oclAsType(soliditycontracts::PrimitiveType).type =
soliditycontracts::PrimitiveTypeEnum::address_payable))
```

Listing 6.6: OCL constraints for the FunctionToStateVariableRelation element

```
context BooleanVariableContext
inv VariableTypeNeedsToBeBoolean:
variable.type.oclIsTypeOf(soliditycontracts::PrimitiveType) and
variable.type.oclAsType(soliditycontracts::PrimitiveType).type =
soliditycontracts::PrimitiveTypeEnum::bool
```

Listing 6.7: OCL constraints for the BooleanVariableContext element

7. Specification of Role-Based Access Control Policies on the Source Code Level

After formally specifying the underlying RBAC model in Chapter 5 and creating the AccessControlMetamodel (ACM) in Chapter 6, the presented relations, predicates and formulae are described on the source code level. This description supports the enforcement of the properties these definitions model. For this enforcement, we employ the capabilities of *solc-verify* and *Slither* from Section 2.4.2 and discuss the resulting reasoning about the implemented access control policies.

The three central sets $\mathbb{R}$, $\mathbb{F}$ and $\mathbb{S}$ from Sections 2.2 and 5.1 include all roles, functions and state variables in the system. The functions and state variables are represented by their equivalent in the source code, so each function in the RBAC policies describes a function in the smart contracts. The same applies to the state variables. However, roles have no representation in Solidity. They are collected using an `enum` where each member describes a single role $r \in \mathbb{R}$. This `enum` is part of an access control contract that is generated in addition to the modelled contracts and handles the role assignment as well as the checking of roles.

As explained in Section 5.1, the FunctionToVariable relation from Definition (5.6) in Section 5.1 connects one function with one variable it is allowed to modify. To describe this relation on the source code level, *solc-verify* provides two elements in its annotation language that can validate whether a function has access to the contract's state variables. One option includes the creation of a postcondition for each $(f, s) \notin \mathsf{FtoS}$. This postcondition compares the value of `s` after the functions execution to its value from before the execution by applying the `__verifier_old_()` keyword. However, in *solc-verify*s current version (v0.5.17 released on the 29th of July, 2020) this keyword does not cover complex Solidity data types like `mappings`, `structs` or `strings`. The other option includes the creation of a modification specifier for each $(f, s) \in \mathsf{FtoS}$. This specifier permits a function to make modifications to the referenced variable. If no modification specifier exists for a tuple (f, s), `f` is not allowed to modify `s` and *solc-verify* returns an error. By applying the modification specifier, no additional keywords except `modifies` are applied. Additionally, the modification specifier covers all of Solidity's data types, so the limitation reported by Reiche et al. [1] regarding the usage of postconditions can be circumvented. However, *SciviK* from Section 2.4.2.3, which is a formal verification tool for Solidity smart contracts with similar capabilities to *solc-verify*, does not provide a comparable element in its syntax, as was analyzed in Table 2.1. Therefore, the usage of modification specifiers limits our

approach to *solc-verify* instead of being general enough to be applied in combination with different formal verification tools.

The presented postconditions are equivalent to the concept of blacklisting and modification specifiers are equivalent to the concept of whitelisting [26]. Both concepts are described in Section 2.2. Applying a postcondition to validate that no changes have been made to a variable is a concrete implementation of the blacklisting concept since unwarranted access is explicitly limited. By employing the modification specifier, permitted access is explicitly described as with the whitelisting approach. Impartial to the usage of blacklisting and whitelisting is the handling of function calls from Definition (5.21), since *solc-verify* considers all function calls when validating annotations. Therefore, the transitivity of the FunctionToVariable relation from Formula (5.22) is covered automatically.

In the remainder of this thesis, we apply whitelisting to express the FunctionToVariable relation on the source code level. This option is also employed by the standard RBAC model by Sandhu, Ferraiolo, and Kuhn [2] and the ACM. Additionally, whitelisting enables the enforcement of the the CSM from Section 5.3, since changes to specific `mapping` locations can be specified when applying the modification specifier. In combination with the `msg.sender` keyword, modifications to a `mapping` by a function are permitted to the callers location. This enforces the CSM on the source code level, ensuring caller-specific access on Solidity `mappings`.

To implement the roles from the formal model, an `enum` in the access control contract is used. However, describing the RoleModifiesVariable relation from Definition (5.11) in Section 5.2 using this `enum` in combination with *solc-verify* is not possible. However, by also considering the RoleToFunction relation from Definition (5.3) in Section 5.1, these accesses are covered indirectly by employing the modification specifier. Each function `f` that is accessible by role r is annotated using the modification specifier for all variables `s` where $(r, s) \in \mathsf{RmS}$. In combination with the modification specifiers describing the function's access to the state variable, the following four cases occur:

1. $(f, s) \in \mathsf{FtoS} \land (r, s) \in \mathsf{RmS}$: If both the function and the role may modify the state variable, a single modification specifier is sufficient. An example from the auction use case from Chapter 4 can be examined in Listing 7.1. Here, the `close` function is employed to modify the `auctionClosed` variable. Additionally, the *seller* role is allowed to access the `close` function and modify the `auctionClosed` variable, resulting in a single modification specifier in line 1 to permit the `close` function to make changes to the `auctionClosed` variable.

2. $(f, s) \notin \mathsf{FtoS} \land (r, s) \notin \mathsf{RmS}$: If neither the function nor the role are permitted to modify the variable, no annotation is created.

3. $(f, s) \in \mathsf{FtoS} \land (r, s) \notin \mathsf{RmS}$: When the function is permitted to access the state variable but the role is not, violations to Formula (5.25) would occur, leading to unwarranted access. Therefore, this violation is communicated to the developer. However, our approach detects this case on the architectural level during a soundness check, so it is communicated back to the architect before the source code generation.

4. $(f, s) \notin \mathsf{FtoS} \land (r, s) \in \mathsf{RmS}$: If the function is not allowed to access the variable but the role may, no annotations are generated. The direct access through the function

```
/// @notice modifies auctionClosed
function close() public onlySeller {
      require(auctionClosed == false, "Auction already closed.");
      auctionClosed = true;
}
```

Listing 7.1: `close` function from the auction implementation in Listing A.1

provides the stronger restriction in comparison to the access for the role, since the role may modify the variable through other functions.

Expressing tuples collected with the RoleToFunction relation from Definition (5.3) on the source code level is achieved by employing Solidity modifiers from Section 2.4.1. Each modifier restricting access to a single function collects the roles permitted to access this function using the set $\mathbb{R}_f \subseteq \mathbb{R}$, which contains all roles fulfilling the callsFunc predicate from Definition (5.16). To check if an entity is assigned to any of these roles, the access control smart contract is employed to access the roles `enum` and check the role assignment. The entity is represented by the `msg.sender` keyword. Applying modifiers to handle role-based function access has also been employed by Reiche et al. [1] and Mavridou and Laszka [45].

To enforce the authorization constraints on roles from Section 5.1, additional restrictions and conditions are implemented in the access control smart contract. These restrictions rely on functionality provided by the `require` keyword from the Solidity programming language.

- **Prerequisite**: To cover the Prerequisite relation from Definition (5.9), one or more role checks are added to the role assignment function in the access control contract. These checks validate that an entity covers all roles that are defined as a prerequisite. It is implemented by employing the `require` keyword in combination with the contract's function to check the role assignment for an entity.

- **Cardinality**: To count the number of entities assigned to a role, an integer variable is defined. Before each new assignment, the variable is compared to the constant limit of roles using the `require` keyword. After a new assignment, the counter is increased.

- **Role Hierarchy**: Assigning a junior role to all its superior roles is implemented by calling the corresponding assignment functions. Since these functions also call the assignment functions for their superior roles, the transitivity of the role hierarchy from Formula (5.10) is also covered.

- **SoD**: Similar to covering the Prerequisite relation, the MutualExclusiveRoles relation from Definition (2.5) in Section 2.2 is enforced by adding role checks to the assignment function. However, these checks validate that the assigned role does not possess the other role.

- **Context**: To cover the temporal constraints, the current timestamp during the creation of the access control contract is stored in a variable using the `block.timestamp` keyword [68]. If temporally restricted access is done later on, the current time is compared to the initially saved value, again by employing the `block.timestamp` keyword. In addition to temporal constraints, the ACM also allows for conditions depending on boolean variables. The corresponding checks are added to the modifier checking the role assignment or to the modification specifier.

The remaining relations to consider are the RoleInfluencesVar relation introduced in Definition (5.12), the VariableToVariable relation from Definition (5.18) and the FunctionToFunction relation from Definition (5.21). However, none of these relations are covered by the invariants, preconditions and postconditions available to *solc-verify*. Since its focus is on the verification of formal specifications, these relations are out of the tool's scope. To analyze these relations on the source code level, the static analysis framework *Slither* from Section 2.4.2.1 is applied to analyze the contract's information flows, function calls and variable influence. However, this functionality is not part of the current capabilities provided by the *Slither* framework, prompting us to implement a custom printer in Section 8.2. Despite this custom printer, no additional source code elements are created to support this analysis.

Considering the transitivity of variable influences from Formula (5.19) and function calls from Formula (5.22), both of them are covered by the static analysis performed by *Slither*. However, enforcing the transitive role hierarchy from Formula (5.10) cannot be achieved using *Slither*. Instead, the hierarchy is directly implemented by setting the junior role to all of its superior roles. On a technical level, this means that the data structure containing the role assignment stores a connection from the entity to the junior role as well as the superior role. This enables a role to access all elements its superior roles can access as well.

After summarizing how the predicates and relations from Sections 5.1 and 5.2 are described on the source code level, we explain how the Formulae (5.20), (5.24) and (5.25) are expressed on the source code level. Despite Formula (5.17) describing relevant properties of the RBAC model, this formula is only validated on the architectural level, as explained in Section 5.1. To reason about the properties represented by the remaining three formulae, the verification results regarding the predicates and relations are combined. All of these formulae are additionally evaluated on the architectural level to find violations already existent in the ACM instances.

To detect unwarranted influence to variables by roles, the fulfillment of Formula (5.20) is validated. Here, a role is allowed to modify one variable `x` but prohibited from influencing a second variable `y`. The presented approach must verify that `x` cannot influence `y`. However, *solc-verify* does not cover variable influence, so the enforcement of this formula relies on the static analysis performed by *Slither*. Similar to Formula (5.20), Formula (5.24) formalizes that a role must be permitted to call all functions in a call chain instead of only a subset. Again, this formula is covered by the static analysis done by *Slither* and not by the formal verification of *solc-verify*.

To uncover unwarranted access to variables, whether it's modifying or influencing access, the fulfillment of Formula (5.25) is validated. This formula expands on the ca-

pabilities of Formula (5.5), thus describing that for each role r that is forbidden from influencing a state variable `s` but permitted to call function `f`, `f` is not allowed to influence `s`. This influence is described with the doesInfluence predicate from Definition (5.26) which consists of three parts that are expressed on the source code level. To reason about unwarranted modification access or modification access through function calls the capabilities of *solc-verify* and its modification specifier are employed. To reason about information flows between variables, the static analysis capabilities of *Slither* are employed.

8. Identifying Insecure Information Flows in Smart Contracts

Modifying data in a software system happens in either a direct, indirect or transitive manner. The most obvious kind are direct changes by a subject through accessing and writing to the data location. Another possibility is indirect or influencing access as explained in Section 5.2, which allows a malicious subject to change data they do not have permitted access to through other means. To identify these influences on the source code level, the flow of information in the system is analyzed using static analysis [35].

One available tool for the analysis of smart contracts is *Slither* by Feist, Grieco, and Groce [7] from Section 2.4.2.1. In Section 8.1, the provided capabilities of *Slither* are considered regarding the coverage of the formal sets, relations, predicates and formulae as explained in Chapter 7. The found limitations in *Slither*'s standard set of printers are avoided by creating a custom printer using the public API in Section 8.2. This API and it's functionality are described in Section 8.2.1. Section 8.2.2 explains the algorithms and methods used for identifying indirect influence between variables. Section 8.2.3 summarizes how the transitive closure for variable influence and function calls is calculated and Section 8.2.4 describes how the results are communicated back to the developer.

8.1. Analyzing Slithers Capabilities for Identifying Information Flows

To enforce the formal RBAC formulae from Sections 5.1 and 5.2, the static analysis must cover information flows between functions and variables. When analyzing the information flows between variables, all three types of influence (direct, indirect and transitive) introduced in Section 5.2 are considered. Additionally, the transitivity of function calls from Formula (5.22) should be involved in the analysis.

Since *Slither* is a framework employed for static analysis, it does not analyse the smart contract elements directly but by calling printers or detectors. These provide functionality to either print information regarding the smart contract or detect specific vulnerabilities like reentrancy from Section 2.4.2.1. These printers or detectors have a specific goal and are applied to achieve different results. To cover the function calls, the `function-summary` printer [69] provides the closest suitable behaviour. By employing this printer, *Slither* summarizes each function by collecting the used modifiers, which variables are read and written and what functions are called. However, this collection of called functions only includes the directly called functions, excluding transitive function calls.

To reason about information flows between variables, *Slither* incorporates the `data-dependency` printer [69]. This printer creates an overview for each function in the contract

based on the underlying AST. This overview summarizes the dependencies between variables. Additionally, an overview over the whole contract is created that visualizes all variable dependencies. However, the printer's definition of dependency lacks relevant considerations regarding the different types of influence relation. Here, a variable depends on another if direct influence occurs. Additionally, the `data-dependency` printer calculates the transitive closure on these direct influences. This implementation excludes indirect influences between variables and the corresponding transitive influences.

8.2. Extending Slither to Cover Indirect Influences and Transitive Function Calls

Due to the limitations described in Section 8.1, *Slither*'s public API is employed to create the `influence-and-calls` printer that covers all types of influence as well as the transitivity of function calls. To implement this printer, the functionality of the `data-dependency` printer is adopted to cover direct information flows between variables. The complete implementation of the `influence-and-calls` printer is presented in Appendix A.2.

If a developer wants to use a specific *Slither* printer, the name of the printer is given as a parameter when employing the framework to analyze a specific contract. Internally, each printer follows the same provided structure by extending the `AbstractPrinter` and implementing the `output` method, so the framework maps a valid parameter to the class implementing its behaviour. The returned results of this method are visualized to the developer and can optionally be exported as a JSON (JavaScript Object Notation) file using another parameter.

The `output` method of the `influence-and-calls` printer is mainly concerned with formatting the output the same way the `data-dependency` printer does. Both printers differ in the information that is collected and formatted. During the execution of the `influence-and-calls` printer, the direct and indirect influences are collected before their transitive closure is calculated. The transitive closure for the function calls is calculated independently.

8.2.1. Slither's Public API

To enable the extension of the framework's functionality, *Slither* provides a public API written in Python [70]. This API provides developers with the tools for creating their own printers and detectors according to a predefined structure. As mentioned in Section 8.2, the `output` method is employed as the starting point for a printers functionality. Each printer references the central `slither` class, which implements the singleton pattern [71, p. 127]. The single instance is created after the framework initially iterates over the contract and collects the contract's general structure, as represented by Figure 8.1. Through this object, the contract's elements like functions and variables are accessed and analyzed by the developer.

The API provided by Slither handles its information in an object-based hierarchical structure that allows access to contracts, functions, state variables and single instructions inside each function. These instructions are represented by the nodes of a control flow

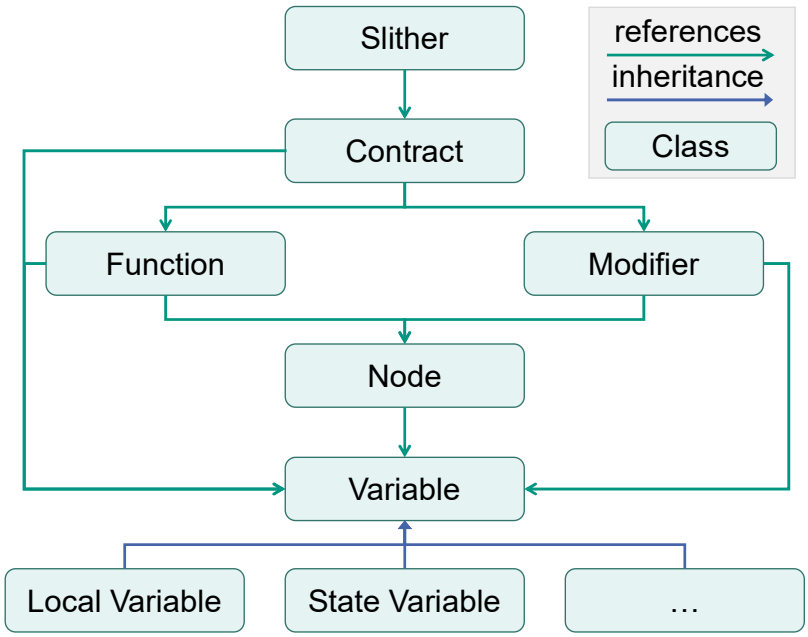

Figure 8.1.: Visualization of the general structure provided by Slither's public API [70]. Except for the initial `slither` object, which is handled as a singleton, all other objects can have multiple instances.

graph (CFG), which describes the underlying data structure employed to handle the instruction sequence of each function. Each node contains the type of expression it encodes as well as a list of children containing the directly following instructions. Additionally, a list of read or written variables is accessible. This list is also available for functions and modifiers. Regarding these variables, the API distinguishes between multiple types by employing an inheritance relation to the abstract `variable` class. The different types cover parameters, local and state variables, function type variables and event variables, enabling developers to handle them differently.

8.2.2. Identifying Influences between State Variables

To cover all types of influence described in Section 5.2, the `influence-and-calls` printer must identify direct and indirect influences by searching for information flows between two variables. These influences are summarized using a dictionary which maps all state variables to their influencers. For this purpose, the `return_all_dependencies` method depicted as pseudocode in Algorithm 8.2 is applied. This method is provided with the current list of contracts for which the printer should analyze the flow of information. These contracts are analyzed by iterating over all state variables in line 4, only skipping the ones that already exist as a key in the created temporary dictionary in line 5 and 6. Afterwards, two identical loops are used to iterate over the potential influencers found by the direct and indirect analysis. The `get_dependencies` method in line 8 is part of the *Slither* framework and is employed by the `data-dependency` printer to collect the direct influences. The `get_indirect_dependencies` in line 12 on the other hand is part of the `influence-and-`

```
function withdrawMoney() external {
      uint amount = currentBids[msg.sender];
      currentBids[msg.sender] = 0;
      if(!msg.sender.send(amount)) {
            currentBids[msg.sender] = amount;
            assert(currentBids[msg.sender] == amount);
      } else {
            assert(currentBids[msg.sender] == 0);
      }
}
```

Listing 8.1: `withdrawMoney` function from the auction implementation in Listing A.1

`calls` printer. This function is provided with a single contract and the current variable. It applies the `get_influencers_for_var_in_func` method from Algorithm 8.3 to return all indirect information flows for the given variable in the given contract. After finding the set of possible influencers, both loops check the found variables for different properties that must be fulfilled. In line 9 and 13 it is verified, that the found variable is a state variable. These checks exclude parameters or local variables in the contract overview, which are included in the function-specific summaries. These summaries also incorporate the `get_dependencies` and `get_indirect_dependencies` functions. Additionally, line 10 and 14 verify that each influencer is only added once to the dictionary. The last property, which is only checked for the direct influencers in line 10, is based on an issue in the current implementation of the Slither framework and its `data-dependency` printer [72]. Due to this issue, a dependency between a variable and itself is found and returned, so this redundant information is excluded in the `influence-and-calls` printer.

To find indirect influence relations in the contract, the `get_influencers_for_var_in_func` method from Algorithm 8.3 searches for possible indirect information flows to the provided variable. In line 3, it checks whether the variable is written by the given function. If that is not the case, the function ends by returning an empty set. In the other case, a loop iterates over all conditional nodes in the function. These nodes represent the beginning of an if clause and are marked by the node type `IF`. For each conditional node, the recursive function `check_children_for_write` is called to traverse the CFG and check whether the current condition node is used to access the variable. If that is the case, all variables read in the conditional node are stored as possible influencers in line 7.

To recursively traverse the CFG and check whether the current variable is written inside the conditional node the call sequence begins with, the `check_children_for_write` method is applied. Its structure can be examined in the pseudocode in Algorithm 8.4. This method is provided with the variable to search for and the current node of the CFG, which is the conditional node in the beginning. The algorithm inspects all children of this node, which are the instructions that can follow directly on the current node. For normal expressions, there is exactly one child, whereas an `if` condition branches into two possible nodes. Line 3 checks if the variable is written in any of the children node, which terminates the recursive function by returning the boolean value `true` in line 4. If the

Algorithm 8.2 Create and return dictionary, which maps all variables to their influencers for the given list of contracts cons

```
1:  function RETURN_ALL_DEPENDENCIES(List<Contract> cons)
2:      res ← empty dictionary
3:      for c ∈ cons do
4:          for v ∈ state variables of c do
5:              if v ∈ res then
6:                  continue
7:              res[v] ← empty set
8:              for inf ∈ get_dependencies(v, c) do
9:                  if inf.type == StateVariable then
10:                     if inf ∉ res[v] & inf != v then
11:                         res[v] ← inf
12:             for inf ∈ get_indirect_dependencies(v, c) do
13:                 if inf.type == StateVariable then
14:                     if inf ∉ res[v] then
15:                         res[v] ← inf
16:     return res
```

Algorithm 8.3 Collect indirect influencers for given variable v in given function f

```
1: function GET_INFLUENCERS_FOR_VAR_IN_FUNC(Variable v, Function f)
2:     res ← empty set
3:     if v ∈ variables written by f then
4:         condNodes ← conditional nodes in f
5:         for cn ∈ condNodes do
6:             if check_children_for_write(v, cn) then
7:                 res ← variables read in cn
8:     return res
```

child is not a node representing the end of the current `if` condition, which is checked with line 5, this function is called recursively for that node, returning a positive value when any of these recursive calls returns a positive value. This enables positive results to be handed upwards through the CFG back to the conditional node for which the function was initially called. If the end of the condition is reached before the variable is written, a negative value is returned. Therefore, the variable is not written in the initially given conditional node.

8.2.3. Calculating the Transitive Closure

To fully cover the formal RBAC sets, predicates and formulae, the transitivity of function calls and variable influences is analyzed as explained in Chapter 7. Since *Slither* does not support this functionality as we laid out in Section 8.1, this functionality is added to the `influence-and-calls` printer.

Algorithm 8.4 Recursively check if given variable v is written in any child node of given node n using the CFG

```
1: function CHECK_CHILDREN_FOR_WRITE(Variable v, Node n)
2:     for c ∈ n.children do
3:         if v ∈ variables written in c then
4:             return true
5:         if c.type ≠ END_IF then
6:             if check_children_for_write(v, c) then
7:                 return true
8:     return false
```

Algorithm 8.5 Recursively calculate the transitive closure of the indirect influence for the variable v

```
1: function CALCULATE_TRANSITIVE_CLOSURE(Dictionary[Variable -> Set[Variable]] d,
   Variable v, Set[Variable] cov)
2:     temp ← empty set
3:     for inf ∈ d[v] do
4:         if inf ∉ cov then
5:             temp ← inf
6:             cov ← inf
7:             temp ← calculate_transitive_closure(d, inf, cov)
8:     return temp
```

When calculating the transitive closure for information flows between variables, the dictionary from Section 8.2.2 is provided as an input. By applying the `calculate_transitive_closure` method provided in Algorithm 8.5, the printer iterates over all state variables in the dictionary. Each variable is added to the results and a set containing already covered variables, so it is skipped in future recursive calls. This method is provided with the dictionary, the current variable to calculate the transitive influencers for and the set of already covered variables. Similar to Algorithm 8.4, this method also calls itself recursively to cover transitive influence through more than one intermediate variable. For all influencers that the method iterates over, line 4 checks whether any influencer has not yet been added to the results. If this check succeeds, the influencer is added to the temporary result set in line 5 as well as the set of already covered variables in line 6. Afterwards, the result set is completed by recursively calling this method with the extended set of already covered variables for the currently covered influencer in line 7. After all influencers have been considered and added to the temporary set, it is returned as the set of variables with transitive influence on the current variable.

In addition to the influence analysis, the described printer also helps to analyze the transitive relationship between the functions to reason about Formula (5.24). Therefore, it also calculates the transitive closure for function calls by applying the same idea implemented in Algorithm 8.5. One modification to this method is that only one function `func` and the set of already covered functions are provided as input, so no dictionary is utilized.

Table 8.1.: Summary of the variable influence results for the *SingleAuction* contract from Listing A.1 using the `influence-and-calls` printer. The complete version is represented in Table A.1.
O marks a found influence relation, and X symbolizes a cell connecting a variable to itself.

	Influencers								
Variables	managingContract	sellerAddress	highestBidder	auctionClosed	moneyCollected	highestBid	bidderCounter	bidders	currentBids
managingContract	X								
sellerAddress		X							
highestBidder	O	O	X	O		O	O	O	O
auctionClosed	O	O		X					
moneyCollected	O	O	O	O	X	O	O	O	O
highestBid	O	O	O	O		X	O	O	O
bidderCounter	O	O	O	O		O	X	O	O
bidders	O	O	O	O		O	O	X	O
currentBids	O	O	O	O		O	O	O	X

This leads to a difference in line 3 as well, since the method iterates over the list of all functions that `func` is calling, which is provided as an attribute of the Function class. The handling of each function element afterwards is the same: If the function has not yet been covered, it is added to the result set and to the set of covered functions before a recursive call to that method is done with the current function from the iterated list. Additionally, the loop from line 3 to 7 is copied and added between line 7 and 8, covering all external function calls. Therefore, the method collects internal as well as external calls in the same set before returning it.

As Anderson [73] explains, static analysis tools and frameworks are limited by the amount of paths needed to consider. The inclusion of loops, exceptions and function calls leads to an unbounded number of paths the tool considers in the worst case. To still reason about these paths in a reasonable amount of time, approximations are applied and certain paths are skipped. The *Slither* framework employed by the `influence-and-calls` printer underlies the same limitations. Since the developers do not describe which paths their framework focuses on, we cannot assume that the information flows found by the `influence-and-calls` printer are complete. Additionally, not considering all available paths in a smart contract leads to an overapproximation regarding the transitive closure since dependencies may be considered that cannot occur during the dynamic execution. So any analysis relying on the transitive closure must consider these additional connections. An example for the analysis of information flows between variables can be viewed in Table 8.1, where variables like `highestBidder` are influenced by almost all of the other state variables despite only depending on three when considering the direct and indirect influence.

8.2.4. Communicating Results Back to the Developer

After the transitive closure has been calculated for the variable influence and the function calls, the information is visually enhanced by employing functionality from the `data-dependency` printer. The results of this output for the auction use case from Chapter 4 can be examined in Table A.1. The shortened version collecting only the information flow between variables can be seen in Table 8.1. The results consist of a table where the first column states the name of the variable or function and the second column summarizes all influencers or called functions. Additionally, similarly structured tables are created for each function to describe the information flow between variables on a function-to-function basis.

As mentioned in Section 8.2.1, the framework can also export the console output into a JSON file. This JSON export depends on the same output as the console, so no additional formatting is done. Therefore, the resulting JSON file contains the complete output as a single string in one element, making further automated handling of these results difficult. However, this representation suffices for the presented approach and enhancements like an automatic reasoning about the results are part of future improvements on this approach. More information on this step is provided in Section 10.1.3.

9. Generation of Formal Specifications to Enforce Role-Based Access Control Policies

After formalizing smart contract RBAC policies in Chapter 5 as a foundation for the AccessControlMetamodel (ACM) from Chapter 6, their coverage on the source code level is described in Chapter 7. To mitigate potential risks for developers and architects that occur when manually implementing the proposed specifications based on the ACM instances, an automatic generation of smart contracts and formal specifications is employed. To automate this translation, M2T transformations from Section 2.1.3 are used. Especially the *Xtend* language [21] is used as a foundation for implementing an automatic generator.

To achieve this goal, we define a mapping between the metamodel elements, the source code elements and the *solc-verify* annotations in Section 9.1. This resulting mapping describes the connections that are implemented in the *Xtend* generator. Section 9.2 presents this generator, beginning with the packages and structure of the generator project in Section 9.2.1. In Section 9.2.2, we describe the preprocessing performed by the generator. This preprocessing step verifies the OCL constraint and validates the model's soundness. Section 9.2.3 explains how the Solidity smart contract stubs are created based on the ACM. Similarly, Section 9.2.4 describes how the formal specifications and other elements responsible for enforcing the RBAC policies are created.

9.1. Mapping Metamodel Elements to Source Code Elements

To employ the developed formal specifications in the Solidity smart contracts from Chapter 7, the metamodel elements representing these specifications are mapped to the Solidity elements. This mapping describes the foundation for the generator. Additionally, the mapping helps with the automatic enforcement of the modelled policies since ambiguities during a manual implementation of the presented approach are minimized.

The ACM described in Section 6.1 consists of two different packages containing elements for modelling the structure of the smart contract and the RBAC elements. The elements from the *SmartContractModel* package from Figure 6.2 are used to describe the structure of the smart contract by defining Functions, StateVariables and data Types. Most of these elements are taken from the *SolidityMetaModel* [66], so their implementation in Solidity is already covered by the *SolidityCodeGenerator* [74]. This includes Function, State Variable and Type. The AccessControlContract element, used as the top-level container for this package, is not directly covered by the generator. Since it inherits most of its attributes and references from the Contract element, its attributes are handled equally. In general, a single

```
1  /// @notice modifies msg.sender.balance
2  /// @notice postcondition msg.sender.balance >=
       __verifier_old_uint(msg.sender.balance)
3  /// @notice postcondition msg.sender.balance <=
       __verifier_old_uint(msg.sender.balance)
4
5  /// @notice modifies address(this).balance
6  /// @notice postcondition address(this).balance >=
       __verifier_old_uint(address(this).balance)
7  /// @notice postcondition address(this).balance <=
       __verifier_old_uint(address(this).balance)
8
9  /// @notice modifies <VariableOrMappingName>
10 /// @notice modifies <MappingName>[msg.sender]
11 /// @notice modifies <VariableOrMappingName> if <Boolean>
```

Listing 9.1: Generated solc-verify annotations based on the ACM elements

Solidity smart contract is generated for each AccessControlContract element. This contract is filled with the functions and variables specified by the contained elements. Lastly, the FunctionBalanceModification elements are used to create additional annotations to the `function` element they reference. The two attributes `modifiesMsgSenderBalance` and `modifiesThisContractsBalance` cover the balance of the function caller (`msg.sender.balance`) as well as the balance of the AccessControlContract the referenced `function` belongs to (`address(this).balance`). For both of them, the possible values are describe by the BalanceModificationType enumeration. The resulting specifications based on these values are shown in Listing 9.1:

1. `doesNotModify`: The referenced `function` is not allowed to modify the specific balance in any way. If this value is selected, no additional annotations are generated.

2. `onlyIncrease`: Here, the referenced `function` is only permitted to increase the balance. To enforce this, the annotations in line 2 or line 6 are generated (depending on the attribute). With these postconditions, it is verified that the balance after the function execution is greater or equal to the value before the function execution.

3. `onlyDecrease`: Similar to the `onlyIncrease` contingency, the referenced `function` is only permitted to decrease the balance. Here, the current value must be less or equal, which can be examined in line 3 for the `modifiesMsgSenderBalance` attribute and line 7 for the `modifiesThisContractsBalance` attribute.

4. `modifiesBothWays`: The `function` is allowed to increase and decrease the specific balance. If this value is selected, one additional annotation is generated. When this value is chosen for the `modifiesMsgSenderBalance` attribute, the annotation in line 1 is created. Line 5 is generated when `modifiesThisContractsBalance` is set to this value.

These postconditions allow for a more precise verification of the generated smart contracts using *solc-verify*. However, the current version of *solc-verify* cannot reason about postconditions related to contract balances in its normal mode [75]. To handle these aspects, modular arithmetic is used as the encoding for arithmetic operations, allowing for range assertions as well as precise wraparound semantics [4]. This encoding can be chosen when starting *solc-verify* using the `mod` parameter instead of the default `int` mode, which encodes the arithmetics using integers. This integer mode "does not capture the exact semantics" of e.g. overflows or unsigned numbers [4, p. 166].

The elements from the *AccessControlSystem* package from Figure 6.3 are not covered by the *SolidityCodeGenerator*, so their generation is originally defined. The central element of this package is the AccessControlSystem, which contains all other elements. On the source code level, this element represents the specific access control contract. This additional smart contract should not be altered after the generation and is employed to handle everything related to roles, their assignment and their inspection. To handle the assignment of entities to roles, the access control contract employs a nested mapping. This mapping stores whether an entity is assigned to a role (represented by an `enum`) by mapping each role to a boolean value in a second, internal mapping. To restrict access to the access control contract, an additional *admin* role is created that is the only one permitted to access the role assignment functions. To check if an entity fulfills a role, Solidity modifiers are implemented like the one from Listing 2.2. This technique for verifying an entities role is also employed by Reiche et al. [1] and Mavridou et al. [50]. During the contracts creation, the *admin* role is assigned to the caller of the contract in the constructor, which is the contract for whom access control should be handled.

Using Solidity modifiers for checking an entity's role is also employed when translating RoleToFunctionRelation elements. First, all roles that are allowed to access a function are summarized by iterating over all RoleToFunctionRelation elements. Then, the roles are combined disjunctively in a single condition to check if an entity has any of these roles. The resulting modifier is added to the function declaration. To further restrict the access of roles to functions, a *Context* can be added to specify certain conditions that must be fulfilled. As explained in Section 6.1, *Context* is an abstract super class which has no concrete elements to translate to the source code level. However, both BooleanValueContext and TimeContext can be added to RoleToFunctionRelation elements by conjoining their condition with the role check. If a role's access to a function is restricted by a BooleanValueContext element, the modifier also inspects if the referenced boolean `variable` has the modelled `valueToCheck`. If a TimeContext element is used, a call to the access control contract is made with the properties specified by the model element. To handle this call, a function to check the current time (represented by `block.timestamp` [68]) against the given limit is created. This limit is given by the `isUpper` attribute and is calculated by converting the `timeValue` to seconds using Solidity keywords [32, p. 79]. To get the reference value for the time comparison, a new variable is added that holds the timestamp during the creation of the access control contract.

Similar to the *Context* element is the *FunctionToVariableRelation* element, which also defines an abstract super class. Since no concrete instances can be created, no translation is defined. However, it has two subclasses FunctionToStateVariableRelation and FunctionToCsmRelation. Both contain a list of *Context* elements, which are limited to the BooleanValue-

Context as explained in Section 6.2. For each FunctionToStateVariableRelation element, the referenced `variable` is ingested into the *solc-verify* modification specifier in line 9 in Listing 9.1. FunctionToCsmRelation elements on the other hand can result in two annotations. When the `accessWholeMapping` attribute is set to true, the FunctionToCsmRelation element is handled like a FunctionToStateVariableRelation element. However, when it is set to false, the restriction to the caller-specific location is applied as explained in Chapter 7. The result can be examined in line 10 in Listing 9.1. If FunctionToStateVariableRelation or FunctionToCsmRelation contain a BooleanValueContext, the referenced boolean `variable` is appended to the generated modification specifier as an additional condition as implemented in line 11 in Listing 9.1.

For each Role element, a new entry is added to the `roles` enumeration in the access control contract. This is reflected in the functions since a new function is added to assign entities to this role. These functions also cover the authorization constraints from Section 5.1. When a role has a `cardinality` that is limited, an integer counting variable is created. Before an entity is assigned to this Role, a check is made if the counter does not exceed the modelled `cardinality`. If another assignment is permitted, the entity is assigned to the role and the counter is increased. If the entity loses this role, the counter is decreased. To handle the connections between roles like hierarchy and prerequisite, additional checks are added to the role assignment function. If a Role has a `prerequisite`, a check controls if the given entity also has that role. In this case, execution may continue, otherwise the function is aborted. Any entity that is assigned to a Role is also assigned to its `superior`'s by adding a call to the `superior`'s assignment function. The last constraint on roles is handled by the MutualRoleExclusion element, which references two distinct Role elements. This MutualRoleExclusion element is handled similarly to the `prerequisite` relation. For each of the two referenced `roles`, a check is added to their assignment function. These checks verify that an entity is not assigned to the other, mutually exclusive role. Only if this check succeeds, the role assignment continues.

The remaining elements from the AccessControlSystem package are not utilized during the generation, but for the soundness check in the preprocessing phase. This includes the FunctionToFunctionRelation, RoleToVariableRelation and VariableToVariableRelation, which are not mapped onto source code elements. However, they are still used to reason about the soundness of the modelled system. Therefore, the formulae from Section 5.1 are considered and verified before the generation is employed. More information on this step will be given in Section 9.2.2.

9.2. Generating Solidity Smart Contracts

To automatically and programmatically translate the ACM elements to elements on the source code level, a code generator is implemented using *Xtend* [21]. This generator is implemented according to the mapping from Section 9.1. As explained in that section, the translation of some ACM elements is already implemented in the *SolidityCodeGenerator* by Dietrich and Reiche [74]. Since this generator employs the *Xtend* language, the presented generator is also implemented with *Xtend*.

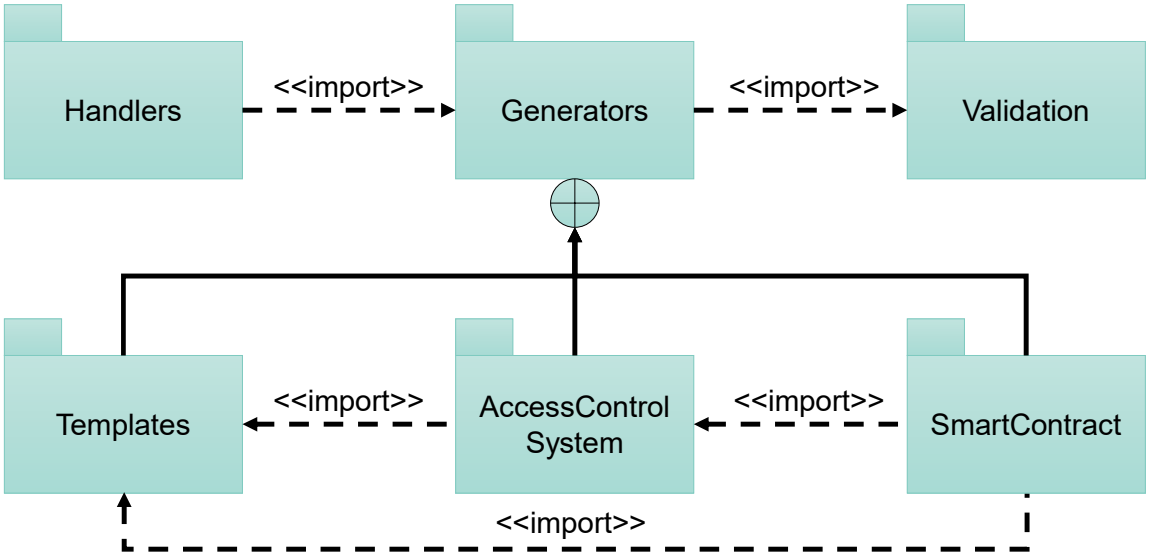

Figure 9.1.: UML package diagram describing the structure of the generator. The ⊕-symbol visualizes a containment relation between packages. Therefore, the *generators* package consists of the three connected sub packages.

The *SolidityCodeGenerator* takes instances of the *SolidityMetaModel* [66] as input and generates the modelled smart contracts. These generated contracts consist of variables, modifier and function stubs. Additionally, *solc-verify* annotations and access control modifiers are created according to the approach by Reiche et al. [1]. This generator is based on the *Ecore2Txt* project by Kramer et al. [76] which implements the basic structure for implementing M2T transformations for *Ecore*-based models. This project is also implemented using *Xtend* and enables the starting of the generator from the Eclipse context menu.

9.2.1. Structure of the Generator

The structure of the generator is visualized in Figure 9.1. It consists of three packages, which all are responsible for a different phase of the generation process. The entry point of the generator when starting the process is the AccessControlGeneratorHandler in the *Handlers* package. This class forwards the call from the context menu to the AccessControlGenerator and AccessControlGeneratorModule from the *Generators* package. This structure is prescribed by the *Ecore2Txt* project [77].

Before the generation is started by the AccessControlGenerator, the input files are preprocessed by employing the classes from the *Validation* package. This includes the validation of OCL constraints as well as checking the fulfillment of the central RBAC formulae from Section 5.1. More information on this process step is provided in Section 9.2.2.

If the validation succeeds, the sub packages of the *Generators* package are employed to create the annotated smart contracts. The abstract templates that function as a blueprint for the creation of either a Solidity smart contract or a Solidity function are collected in the *Templates* package. These templates are implemented in the *SmartContracts* package to enable the generation of Solidity smart contracts based on the model. This generation

```
1 There are violations in the selected AccessControlSystem 'Auction' and
    SmartContracts 'AuctionManagement' 'SingleAuction':
2 1) Role 'Manager' cannot modify variable 'Auction Closed' through any
    function
3 2) Role 'Manager' - Violation for OCL constraint 'CardinalityIsValid':
4 The role cardinality needs to be -1 or bigger than 0
5 3) Violation for OCL constraint
    BalanceModificationsReferenceDifferentFunctions:
6 All balance modifications need to reference different functions
```

Listing 9.2: Example for a violations.log file

is described in more detail in Section 9.2.3. Additionally, the classes in the *AccessControlSystem* package are employed to create the *solc-verify* annotations as well as creating the access control modifiers to enforce the RBAC policies in the smart contracts. The additional access control contract is generated here as well. Section 9.2.4 provides more details on this generation.

9.2.2. Verifying Constraints and Soundness Check

Before the generation of annotated smart contracts begins, the soundness of the ACM instances is validated. This validation is performed by the AccessControlValidator from the *Validation* package (see Figure 9.1). This class validates the OCL constraints in the model as well as the central RBAC formulae from Section 5.1 by using the two available implementations of the ViolationGenerator interface. This interface and its implementations employ the *Strategy* design pattern explained by Gamma et al. [71, p. 315-316]. The ViolationGenerator describes the abstract *Strategy* that is implemented concretely in the OclViolationGenerator and the RbacViolationGenerator. The enumeration IllegalInfluenceResults is employed to differentiate the outcomes by the ViolationGenerators.

To communicate the found violations back to the architect, the AccessControlValidator collects all error messages returned by the implementations of ViolationGenerator and saves them to a text file. An example for such an output is presented in Listing 9.2. These violations are numbered and the models they stem from are annotated in the header in line 1. To identify these violations, the OclViolationGenerator validates the modelled OCL constraints from Section 6.2 using functionality provided by the EMF. For each constraint that cannot be validated, the model element, name of the constraint and a more detailed error message are connected and returned to the AccessControlValidator. An example can be examined in line 3 and 5 of Listing 9.2. Similarly, the RbacViolationGenerator returns detailed error messages like the ones in line 2, which include the concrete names of the model elements responsible for the violation. For this purpose, it checks the Formulae (5.17), (5.20), (5.24) and (5.25) from Sections 5.1 and 5.2 regarding their fulfillment. If no violations are returned by either ViolationGenerator, the generation continues. Otherwise the log is created and further generation is aborted.

```
function checkAccess(address entity, Roles role) public view returns(bool
    result) {
      return roleAssignment[entity][role];
}

function checkTiming(bool upper, uint256 addition) public view returns(bool
    result) {
      if(upper) {
            return block.timestamp >= timeAtStart + addition;
      } else {
            return block.timestamp <= timeAtStart + addition;
      }
}
```

Listing 9.3: Functions to check timing and role assignment in the access control contract from Listing A.4.

9.2.3. Creating Solidity Smart Contracts

As explained in Section 9.2.1, the *Templates* package contains blueprints for the creation of Solidity smart contracts and Solidity functions. These blueprints employ *Xtend* string templates [21] to describe the general structure of the source code elements on an abstract level. The concrete implementations generate the different parts of the templates to create a complete smart contract or Solidity function.

The concrete implementations can be examined in the *SmartContract* package from Figure 9.1. Here, the SolidityContractGenerator and SolidityFunctionGenerator are employed to generate a complete contract or a single function based on the architectural model. When generating the complete contract, the SolidityFunctionGenerator is integrated to generate a single function. Additionally, the ModifierGenerator and AnnotationGenerator from the *AccessControlSystem* package are included to generate the access control modifiers and the *solc-verify* annotations for each function. For generating specific source code elements like the constructor, events, state variables, `modifiers`, `enums` and function declarations, the *SolidityCodeGenerator* [74] is employed.

The *SmartContract* package also includes the two classes SolidityNaming and SolidityConstants. These utility classes are implemented to cover all constant values used at multiple places throughout the generation and to cover all aspects related to naming in the created contract. The latter is responsible for formatting the name a developer gives a variable to a scheme that adheres to Solidity code styles [78] and can be compiled.

9.2.4. Creating Elements Enforcing the Policies

To enforce the modelled RBAC policies, in addition to the smart contract's structure, the generator creates annotations and modifiers for the smart contract. Additionally, another smart contract is generated that handles the role assignment and checking during the execution, as was proposed by Reiche et al. [1]. The generated access control contract

```
/// @notice modifies roleAssignment[entity][Roles.BIDDER]
function changeBidderRoleForEntity(address entity, bool giveRole) external
    onlyAdmin {
      if(giveRole) {
            require(!checkAccess(msg.sender, Roles.SELLER), "The address
    cannot be a Seller as well.");
      }

      roleAssignment[entity][Roles.BIDDER] = giveRole;
}
```

Listing 9.4: Function to assign the *bidder* role in the access control contract from Listing A.4.

for the auction use case from Chapter 4 can be viewed in Appendix A.4. The classes employed to achieve these generations are located in the *AccessControlSystem* package of the generator from Figure 9.1.

Similar to the *SmartContract* package from Section 9.2.3, this package also includes multiple utility classes. Constant values that are related to the access control enforcement (e.g. *solc-verify* keywords) are summarized in AccessControlConstants. Similarly, the AccessControlUtility class contains functionality that is not inherently related to any other class but used throughout this package, like checking if access control references are necessary for a specific contract. The last utility class ModifierRoleAndConditionsHelper describes a data structure used to support the modifier generation by collecting the necessary model elements at a central location.

To create the access control elements, three generators are available. The AnnotationGenerator is employed to create *solc-verify* annotations for a function based on the proposed implementation from Listing 9.1. The ModifierGenerator not only assembles an expressive name for each modifier but also constructs the concrete implementation located at the end of a contract. To generate the access control contract, the AccessControlContractGenerator is utilized. This generator also extends the SolidityContractGenerationTemplate but does not rely on other generators for filling the template methods. In addition, functions for checking access, checking the temporal constraints and assigning roles are created during the generation so no changes to this contract are necessary after its initial creation. To check if an entity is assigned to a role, the `checkAccess` function from lines 1 to 3 in Listing 9.3 is employed. To verify the temporal constraints, the `checkTiming` function from lines 5 to 11 in Listing 9.3 is executed. To assign an entity to the *bidder* role in the auction use case from Chapter 4, the `changeBidderRoleForEntity` from Listing 9.4 is employed. All other role assignment functions are generated by following the same structure.

10. Outlining the Development Process Based on the Presented Approach

To summarize how the presented approach for modelling and enforcing RBAC policies is employed, the resulting development process is outlined in Figure 10.1. Each phase is annotated with either the responsible role or the tool category that is applied. Additionally, the in- and outputs of each phase are stated in Table 10.1. During the process, the two roles software architect and developer are summarized as the *stakeholders* of the system.

To begin, the software architect utilizes the ACM from Section 6.1 to create new instances for the use case to be implemented based on its requirements and the RBAC policies. After the model is created, the soundness check from Section 9.2.2 is applied as a preprocessing step before the generation is started. If the soundness check detects any violations of the OCL constraints from Section 6.2 or the formal RBAC model from Sections 5.1 and 5.2, these violations are collected and communicated back to the architect. The generation process is aborted. If no violations are found, the annotated smart contracts are generated as explained in Section 9.2. Since the generated contracts consist of code stubs due to missing behavioral information in the model instances, the software developer adds the implementation to the empty methods in the next step. If the developer finds any shortcomings in the generated contracts like missing state variables, roles, functions or connections between those, they communicate this back to the software architect. The architect then refines the model, beginning the process anew. If no problems arise during the implementation, the verification of the implementation can be performed.

This verification process begins by employing *solc-verify* from Section 2.4.2.1 and the custom *Slither* printer from Section 8.2. Since both tools are employed independently from each other, the order of execution does not matter. *Solc-verify* is used to compile the contracts and verify the implementation regarding the specifications. The `influence-and-calls` printer, on the other hand, checks the variable influences existing in the contracts as well as calculating the transitive closure for the function calls. After the tools return their results, these results are checked for violations. If no problems are found, the smart contracts are deployed and the correct enforcement of the RBAC policies is enforced. However, if any problems are found, these problems are communicated back to the stakeholders. Since the problems are mostly resulting from a mismatch between the architectural model and the implementation, they can be communicated either to the software architect or the software developer. To inform the software architect, a *correspondence model* [79] is employed to map source code elements back to the architectural elements. Based on the communicated information, the ACM instances are adapted. Informing the software developer is achieved by connecting the found violations to concrete lines in the source code. Therefore, these lines are adapted to resolve the violations in the implementation.

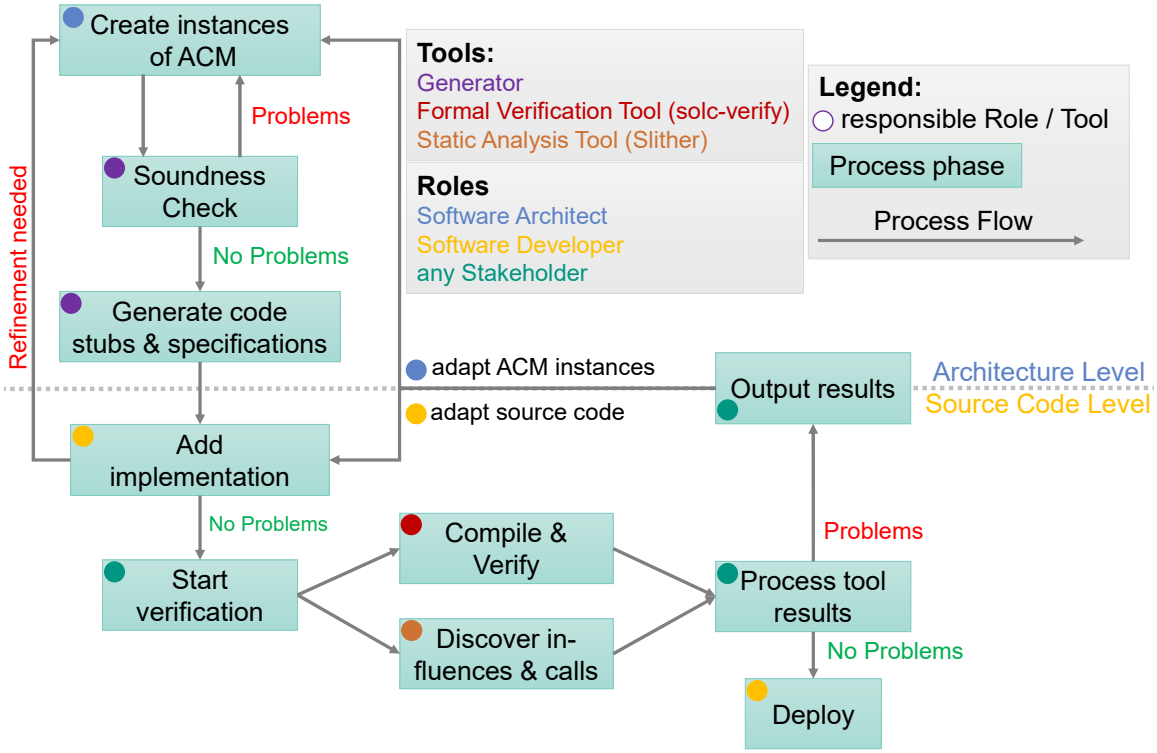

Figure 10.1.: The process envisioned for developers and architects using the presented approach to enforce their smart contract RBAC requirements.

```
$ solc-verify.py <Filename>.sol --arithmetic mod
$ slither <Filename>.sol --print influence-and-calls
```

Listing 10.1: Console instructions to start both tools

The three phases *Start Verification*, *Process Tool Results* and *Output Results* are employed by any stakeholder, as is expressed in Figure 10.1. Here, future work must create an approach for the automatic handling of these steps. However, due to the scope of this thesis, the theoretical concepts behind these steps are explained and the practical implementation is left open. Currently, any stakeholder must employ these steps manually according to the theoretical foundations. For the *Start Verification* step, both tools are started for the smart contract file by executing the console instructions from Listing 10.1. Ideas for collecting and verifying the results in the *Process Tool Results* step are explained in Section 10.1. Section 10.2 explains how the results are communicated back to either the developer or the architect during the *Output Results* step.

10.1. Verifying and Analyzing the Tool Results

After the tools have been employed, their returned results are evaluated to find possible violations to the created model instances. This is performed in the *Process Tool Results* step in Figure 10.1. We introduce the possible changes the developer can introduce during the

Table 10.1.: Summarizing the in- and outputs for the different phases of the envisioned process from Figure 10.1.

Phase	Input	Output
Create ACM instances	Requirements & AC policies	ACM instances
Soundness Check	ACM instances	Validation results
Generation	ACM instances	Contracts with code stubs & Access control contract
Add Implementation	Contracts with code stubs	Fully implemented contracts
Start Verification	Contracts	Commands to start tools
Compile & Verify	Annotated Contracts	Verification errors
Discover Influence & Calls	Contracts	Printer result
Process Results	Tool results	Error information for stakeholders
Output Results	Error information	Feedback for Stakeholders
Deploy	Contracts	Deployment results

implementation in Section 10.1.1. Based on these changes, the analysis of the tool results is explained in Section 10.1.2 for *solc-verify* and in Section 10.1.3 for the *Slither* printer.

10.1.1. Changes and Violations during the Implementation

When the soundness check finishes without any problems, we assume that there are no violations to the modelled access control policies in the ACM instances. An implementation that adheres to these instances does not result in any problems when the tools are employed for verification. However, we distinguish two cases where problems are found. In the first case, the developers implementation differs from the RBAC policies modelled by the architect. This either happens through deliberate changes made by the developer or through errors made during the implementation. In the second case, the developer changed the automatically created parts of the generated contracts. This influences the elements responsible for the RBAC enforcement on the source code level. Both of these cases can also occur in combination when the developer changes the generated code elements as well as implementing behaviour that differs from the one modelled by the architect.

At this point, it is assumed that errors found by our tools happened due to the first case, since the goal of our approach is to generate parts of the code that enforce the modelled RBAC policies. However, the distinction between deliberate changes and implementation errors is irrelevant since both result in the same considerations. Despite this assumption, there are possible changes that fall into the second category without undermining the generated elements. For example, adding new *solc-verify* annotations utilized for functional reasoning can be done without invalidating the generated annotations. These annotations however cannot include modification specifications. Another example would be the implementation and addition of new modifiers to a function declaration. This is only valid, if the new modifiers are evaluated after the generated ones, if they do not rely on the access control contract and do not introduce any new influence relations.

The following enumeration summarizes all changes the developer makes that differ from the modelled RBAC policies. Adding the complete implementation is not considered

a change but the different instructions the implementation comprises of are covered. For each change, we analyze if it introduces any violations to the verification results.

C1 Letting a function directly access a state variable it is not meant to modify. In this case, Formula (5.25) is violated indirectly through a violation in the doesInfluence predicate in Definition (5.26).

C2 Not modifying a state variable through a function, despite the modification being modelled. This case violates Formula (5.17) if no other function modifies the state variable. However, this formula was introduced as a warning on the architectural level. If it is violated, no unwarranted access is introduced that enables a role to illegally access a variable. Therefore, this change does not lead to problems with the correct enforcement of the modelled policies, if it is violated in the source code.

C3 Adding a new function call if the roles that accesses the caller are not allowed to access the callee. In that case, Formula (5.24) is violated. Additionally, Formula (5.25) could be violated through the doesInfluence predicate from Definition (5.26), since the correctness of Formula (5.25) depends on this predicate. The doesInfluence predicate could be violated, since the callee could make changes to variables the caller is not allowed to modify or influence.

C4 Removing a function call that is stated in the model. This case does not introduce any violations to the model.

C5 Letting a variable influence another it is not meant to influence according to the model. Here, Formula (5.20) is violated if a role that is allowed to modify the influencer is not allowed to influence the influenced variable. Also, this introduces violations to Formula (5.20) through the doesInfluence predicate (Definition (5.26)) for the same reason.

C6 Removing the influence of one variable to another, despite this influence being modelled. This also does not result in any additional violations.

C7 Changing the way a function modifies the contract's balance and not updating the model accordingly. This does not result in any violations to the formulae. However, if this case occurs it is communicated back to the stakeholders since unwarranted access to the functions balance can result in monetary losses and exploits.

Due to the transitivity of the FunctionToFunction relation formalized in Formula (5.22) and the VariableToVariable relation in Formula (5.19), all changes except change C7 also lead to each other, depending on the concrete implementation. We do not check change C2 on the source code level, as violations to this formula do not introduce violations to the correct enforcement of RBAC policies. Therefore, only the changes C1, C3, C5 and C7 are checked by employing the verification tools. This leads to *solc-verify* being used to check Formula (5.25) directly and Formula (5.24) indirectly. The `influence-and-calls` *Slither* printer on the other hand is employed to check Formulae (5.20), (5.24) and (5.25). Despite both tools covering the same formulae, they identify different violations by uncovering violations to distinct parts of these formulae. No single tool verifies all elements

```
-- no Violation
<ContractName>::<FunctionName>: OK
-- Violation type 1)
<ContractName>::<FunctionName>: ERROR
- <ContractName>.sol:<Line>:<Column>: Function might modify '<VariableName>'
    illegally
-- Violation type 2)
<ContractName>::<FunctionName>: ERROR
- <ContractName>.sol:<Line>:<Column>: Postcondition 'address(this).balance
    >= __verifier_old_uint(address(this).balance)' might not hold at end of
    function.
-- Violation type 3)
<ContractName>::<FunctionName>: ERROR
- <ContractName>.sol:<Line>:<Column>: Function might modify balances
    illegally
```

Listing 10.2: Violations found by solc-verify regarding the generated annotations

of these formulae. The *Slither* printer uses static analysis to identify information flows between functions and variables whereas *solc-verify* validates the modifying access from functions to variables as well as access to the contracts balance.

10.1.2. Analyzing solc-verify's Results

Solc-verify detects three types of violation to the generated formal specifications. These violations can be examined in Listing 10.2. *Solc-verify* analyzes all functions regarding their specification, resulting in messages that state the contract name followed by the function name. If an error is found, the message begins with the contract name as well as the line and column of the error. However, the line and column describe the beginning of the function declaration and not the execution statement inside the function that is responsible for the violation.

The first violation type shows a violation to the modelled RBAC policies since a function has illegal modification access to a state variable. There are two possible changes by the developer from Section 10.1.1 that lead to this type of violation. Changes C1 and C3 both lead to illegal variable access for a role by violating Formula (5.24) through the introduction of a direct or indirect unwarranted access to a state variable. To communicate this violation back to the stakeholders, the contract, function and variable name are extracted from the error message. To also check if a role gains access to the illegally accessed state variable, all roles that are allowed to call the function are considered. If any of those roles is not allowed to access the state variable, this information is appended to the error message for the next step of the process from Figure 10.1.

For the second type of violation to occur, the developer modifies the contract balances in a manner that differs from the modelled way of accessing the balances, as explained

in change C7 from Section 10.1.1. To communicate this violation back to the stakeholders, the name of the function as well as the address of the endangered balances are extracted.

The last type of violation occurs if balances of addresses that are neither the function caller nor the current contract are changed. This case cannot be covered with the model due to limitations to the FunctionBalanceModification element, which only enables the modelling of changes to the `msg.sender` or the current contract balance. Since the model instances cannot be changed to handle this problem, the stakeholders are informed but the process isn't aborted.

10.1.3. Analyzing Slither's Results

To analyze the implemented smart contracts regarding the variable influences and the function calls, the `influence-and-calls` printer from Section 8.2 is employed. As presented in Table 8.1, the printer summarizes for each state variable the other state variables it is influenced by. Additionally, it shows all functions called by another function. This includes direct as well as transitive calls. Both of these results are returned by the printer in form of a table. To analyze these results, each combination found in the table is considered independently.

If the printer returns that state variable `var` is influenced by the variable `inf`, the stakeholder checks whether this influence is part of the model instances or if it was introduced by change C5 from Section 10.1.1. Therefore, the stakeholder checks if a VariableToVariableRelation element exists in the ACM instance, where the `influenced` variable is `var` and the `influencer` is `inf`. If such an element exists, no changes have been made and no violations are introduced. However, the absence of such an element introduces violations. To find these violations, it is checked if any role exists that is allowed to modify `inf` but not allowed to influence `var`. In that case, a direct violation to Formula (5.20) and to the doesInfluence predicate from Definition (5.26) is detected. Violating this predicate indirectly violates Formula (5.25). These violations are communicated back to the stakeholders. However, if no role gains unwarranted influence on the variable `var`, a VariableToVariableRelation element connecting `inf` and `var` is missing in the model. The stakeholders are informed about this missing element. In both cases, the error information can include the name for both variables as well as the contracts name. For more information, the additional tables can be analyzed which describe the influence relation on a function to function basis. By searching for the functions where `var` is influenced by `inf`, the expressiveness and helpfulness of the error message can be increased.

To analyze the transitive closure for function calls, a similar approach is used. For each function `func` that calls function `call`, it is investigated if a FunctionToFunctionRelation element exists in the ACM instance or if it was introduced by change C3 from Section 10.1.1. If such an element exists, the analysis continues with the next function pair. However, if no such element exists, it is reviewed if a role is allowed to call `func` that is not allowed to call `call`. If this is the case, a direct violation to Formula (5.24) and an indirect violation to Formula (5.25) has been found and is communicated to the stakeholders. However, if no such role exists, the absence of a fitting FunctionToFunctionRelation element is returned. Here, the error information consists solely of the two function names and the name of the contract. No additional information is extracted from the printer results.

Additionally, the `influence-and-calls` printer underlies the limitations to static analysis tools mentioned by Anderson [73] and explained in Section 8.2.3. Therefore, calculating the transitive closure cannot consider all possible execution paths, leading to an overapproximation. However, all information flows are communicated to the stakeholders who investigate the risk associated with each information flow.

10.2. Communicating Results Back to the Stakeholders

After the tools have been applied and their results have been analyzed as described in Section 10.1, the found violations are communicated back to the stakeholders. Since the outlined process supports two stakeholders working on different levels of abstraction and at different steps during the process, the results are not communicated to both of them simultaneously.

Both tools verify the system's implementation on the source code level and despite *solc-verify* relying completely on the generated formal specifications from Section 9.2.4, the `influence-and-calls` printer cannot detect violations without comparing the results of its static analysis to the policies in the ACM instances, as explained in Section 10.1.3. Therefore, the source code elements like functions or state variables are linked to their counterpart in the architecture model. One approach employed for the connection of elements from different models are correspondence models [79]. A correspondence model has a connection to the elements of both models it links together, describing a mapping between two elements. For example, a StateVariable element from the ACM is connected to the Solidity state variable in the smart contract.

Such a correspondence model is automatically created during the generation based on the mapping from Section 9.1 and it stores a reference to each model and source code element in its own elements [80]. To save these references, the model elements as well as the source code elements must be unique. For the ACM elements, this is guaranteed through the usage of the *Metamodel-Modeling-Foundations* [67] explained in Section 6.1, which equips every element with a unique identifier. Similarly, many source code elements are uniquely identified due to the name they are generated with. For example, two state variables with the same name in the same contract cannot exist. Additionally, the EMF enables referencing of the created model instances. However, model elements with no concrete equivalent on the source code level like FunctionToFunctionRelation elements cannot be connected to the source code level. Later in the process, the correspondence model is applied to recover these connections, mapping source code elements back onto their architectural counterparts. Therefore, analyzing the printer results as well as communicating violations back to the architect is achieved.

Disclosing information about the violations to the developer is achieved directly on the source code level once the tool results have been collected. Since all of the results from both tools include at least the name of the contract and the responsible function and/or variable, identifying the location of the violation in the source code is achievable even without syntax highlighting or the addition of concrete line numbers. However, addressing both stakeholders independently leads to additional communication efforts since no clear responsibilities have been specified. Therefore, we propose to communicate the results

solely back to the software developer, assuming that the provided ACM instances provide the ground truth of the RBAC policies. As explained in Section 10.1.1, we consider the only changes leading to violations to come from a mismatch between the software architect's policy definition and the concrete implementation created by the software developer.

11. Evaluation

After the development process employing our approach is outlined in Chapter 10, the fulfillment of the main goal described in Chapter 1 is evaluated. This goal focuses on the correct enforcement of RBAC policies for smart contracts on the source code level based on an architectural model. We introduce the following two research questions to reason about the goal's achievement:

RQ1 How can access control policies be modelled on an architectural level and translated into formal specifications to ensure a correct implementation?

RQ2 Which information is necessary and appropriate on the architectural level to describe smart contract RBAC policies?

To evaluate the goal and the research questions in a structured manner, the Goal-Question-Metric (GQM) approach by Basili, Caldiera, and Rombach [81] is applied. This approach defines an organized procedure for achieving certain evaluation goals and is described in Section 11.1. In Section 11.2, an argument for the correct enforcement of the modelled RBAC policies on the source code level is provided. To uncover limitations in the outlined process, a case study is conducted throughout Section 11.3. This begins by describing the underlying use cases, beginning with Augur in Section 11.3.1. We summarize the other two use cases Fizzy and Palinodia in Sections 11.3.2 and 11.3.3. Section 11.3.4 describes the designed case study based on the guidelines by Runeson and Höst [82]. We introduce violations to the Augur and Fizzy use cases in Section 11.3.5 and we examine the results of the case study in Section 11.3.6. In Section 11.4, we apply a Metamodel Coverage Analysis proposed by van Amstel and van den Brand [83]. Section 11.5 discusses the results for the complete evaluation and answers the research questions accordingly. Additionally, we state the threats to validity in Section 11.6.

11.1. Goal-Question-Metric Plan

To evaluate the presented approach, a structured top-down approach is desirable. Therefore, the Goal-Question-Metric (GQM) approach by Basili, Caldiera, and Rombach [81] is applied. Here, the evaluation is described on three different layers. On the conceptual level, the evaluation's *goal*s are stated and defined. This description includes the aspects to consider, the target and the point of view. The qualities that are evaluated are related to the goals using *question*s on the operational layer. Each question influences how the evaluation is performed by stating a concrete point of view for the evaluation. To answer the *question*s, multiple *metric*s are defined on the quantitative level. These *metric*s define the data that is collected to reason about the *question*s and thus measure the *goal*s' achievement.

We introduced two research questions to verify the fulfillment of the main goal of our approach in the beginning of this chapter. This goal is the correct enforcement of the modelled RBAC policies in the generated smart contract implementation. A correct enforcement is achieved, when each role can only access the state variables it is permitted to and no unwarranted accesses occur. This goal is examined in **RQ1**, from which we derive the first goal **G1** for our evaluation.

G1 **Correct Enforcement**: The generated smart contracts enforce the modelled RBAC policies correctly.

- **Q1** Do the generated formal specifications enforce the modelled RBAC policies correctly?
  - **M1.1** Argument supporting the correct enforcement
  - **M1.2** Percentage of introduced violations found during the case study

From this first goal **G1**, the question **Q1** is directly derived. We collect the data necessary to reason about **Q1** with the metrics **M1.1** and **M1.2**. **M1.1** discusses the correctness of the enforcement in Section 11.2 based on the mapping between the formal model, the ACM and its enforcement on the source code level. Additionally, we manually add violations during the case study in Section 11.3. These violations describe unwarranted access on the architectural and source code level. To examine the correct enforcement, we examine the percentage of detected violations with **M1.2**.

In addition to **RQ1**, **RQ2** evaluates the expressiveness of the description on the architectural level. By asking, what information is necessary and appropriate to model on the architectural level, it supports the fulfillment of our goal by reasoning about the metamodel's capabilities to describe the policies to enforce. To evaluate **RQ2**, the metamodel's completeness is evaluated with the goal **G2**.

G2 **Completeness**: The AccessControlMetamodel contains all elements that are necessary to model RBAC policies for smart contracts. Also, no unnecessary elements are included on the architectural level.

- **Q2** Are the created metamodel elements sufficient to model smart contract RBAC policies?
  - **M2** Amount of limitations to the metamodel found during the case study
- **Q3** Does the ACM contain model elements that are not necessary to enforce RBAC policies?
  - **M3** Percentage of model elements covered by the generator

To examine the metamodel's completeness in **G2**, we ask the question whether the ACM has sufficient elements to model smart contract RBAC policies (**Q2**). The data to answer this question is collected by relying on the same case study used for **M1.2**. However, only problems or limitations with the model are considered for **M2**. Additionally, to evaluate whether the ACM contains any redundant model elements, **Q3** is examined. This question considers the capabilities of the generator to reason about unnecessary model elements

by looking at gaps in the coverage of the implemented generator. The data to answer this question is collected by executing a Metamodel Coverage Analysis proposed by van Amstel and van den Brand [83] in Section 11.4. This analysis identifies and visualizes the model elements handled during a model transformation, allowing for the identification of gaps in the model's coverage (**M3**). For these gaps, we evaluate whether they were introduced on purpose or if the corresponding model element is not necessary to enforce RBAC policies on the source code level.

11.2. Reasoning About the Enforcement's Correctness

The main goal of our approach is to provide architects and developers with the tools to describe smart contract RBAC policies on the architectural level and automatically translate the modelled policies into Solidity source code. This source code employs the verification tools presented in Section 2.4.2 to correctly enforce the policies on the source code level. In the following section, we discuss the enforcement's correctness. Therefore, the presented argument shows how entities do not access smart contract state variables that they are not allowed to access in the underlying model.

For this argument, we make multiple assumptions about the stakeholders and their influence during the development process. As we explained in Section 10.1.1, we assume that the developer does not change the generated elements in the smart contract, since these elements are responsible for the correct enforcement of the modelled policies. Additionally, we assume that the developer implementing the smart contract adds no unwarranted role assignments using the access control contract.

To stop an entity from accessing prohibited state variables, we first consider the means employed by an entity to modify a state variable. As we explained in Section 2.3, smart contract systems commit their state variables to the underlying blockchain technology. Due to the immutability of this technology, changes to these state variables can only be induced through transactions. In Solidity, these transactions are represented as functions. So for an entity to modify a state variable, it must access a function permitted to access that variable.

To restrict function access to entities that possess the permitted roles in the ACM instances, we employ Solidity `modifiers` in combination with the access control contract, as we explained in Chapter 7. Inside each `modifier`, a `require` instruction checks the entities assignment to any permitted role. As we explained in Section 2.4.1, the Solidity programming language guarantees that both the `modifier` and the `require` keyword provide correct functionality and can be utilized to restrict access to functions. However, the `checkAccess` function of the access control contract is investigated further. This function can be examined in Listing 11.1 and is responsible for checking whether a given entity possesses a certain role. The returned value corresponds to the correct location in the nested mapping describing the role assignment. Due to the simplicity of this function, we determine its correctness through observation instead of providing a complex formal reasoning. As this function is generated in the same manner for all ACM instances, the correct execution is universal for all application scenarios. However, if entities are wrongly assigned to roles through the access control contract, the `checkAccess` does not return

```
function checkAccess(address entity, Roles role) public view returns(bool
    result) {
      return roleAssignment[entity][role];
}
```

Listing 11.1: `checkAccess` function from the generated access control contract

a correct result. To prevent these incorrect assignments, access to each function that assigns an entity to a role is limited to the *admin* role. This role is solely granted to the contract which created the access control contract. This assignment to the *admin* role is also checked with a `modifier` in combination with the `checkAccess` function. Since we assume that no changes have been made to the access control contract and that the developer only implements correct entity-to-role assignments, no entity is assigned to a role that they are prohibited from possessing.

Now that access to functions has been correctly restricted to only cover permitted roles, access of these functions to the state variables must be restricted. As we explained in Chapter 7, we rely on the verification capabilities provided by *solc-verify*. In particular, we create a modification specifier for each variable that a function may modify. Therefore, any change to a variable that is not represented by a modification specifier is found during the verification with *solc-verify*, as explained in Section 10.1.2. Due to the provided functionality by *solc-verify*, all outgoing function calls are considered automatically when verifying one function. The creation of these modification specifiers depends on the FunctionToVariableRelation elements from the ACM, as explained in Section 9.1. Only when such an element exists in the concrete model, a modification specifier is added. Therefore, only access relations that are part of the modelled RBAC policies are translated into modification specifications.

To conclude, the unwarranted access to state variables by entities is prevented by employing our described approach. However, this correctness is only guaranteed as long as the previously described assumptions hold. Additionally, the reasoning only concerns modification access to state variables. However, as we explained in Section 5.2, our approach also considers influence access based on information flows between variables. We employ the `influence-and-calls` *Slither* printer from Section 8.2 to uncover all types of information flows in Solidity: direct, indirect and transitive influence. By analyzing each flow independently according to the instructions from Section 10.1.3, insecure information flows in the implementation are discovered. If no insecure information flows are discovered, the correct enforcement of RBAC policies is achieved. However, detected insecure information flows are communicated back to the developer, who must change the source code accordingly. Due to the scope of this thesis, we do not reason about the printer's completeness, so our approach does not guarantee that all influence relations are found.

Additionally, we check whether the authorization constraints introduced in Section 5.1 influence the correct enforcement of the RBAC policies. Connecting two roles using the Prerequisite relation from Definition (5.9) or the mutual exclusion from Definition (2.5) in Section 2.2 is covered on the source code level by adding more calls to the `checkAccess` function in the corresponding role assignment functions. This coverage is explained in

```
1  /// @notice modifies roleAssignment[entity][Roles.EXAMPLE]
2  /// @notice modifies exampleCounter
3  /// @notice postcondition (exampleCounter ==
       __verifier_old_uint(exampleCounter) + 1) == giveRole
4  /// @notice postcondition (exampleCounter ==
       __verifier_old_uint(exampleCounter) - 1) == !giveRole
5  function changeExampleRoleForEntity(address entity, bool giveRole) external
       onlyAdmin {
6        if(giveRole) {
7              require(exampleCounter < 5, "...");
8              exampleCounter++;
9        } else {
10             require(exampleCounter > 0, "...");
11             exampleCounter--;
12       }
13
14       roleAssignment[entity][Roles.EXAMPLE] = giveRole;
15 }
```

Listing 11.2: Enforcing the correct increase and decrease to role cardinality counters

```
1  AccessControlContract::[constructor]: OK
2  AccessControlContract::checkAccess: OK
3  AccessControlContract::changeExampleRoleForEntity: OK
4  AccessControlContract::changeAdminRoleForEntity: OK
5  No errors found.
```

Listing 11.3: solc-verify results for an annotated access control contract

more detail in Chapter 7. Since the correctness of the `checkAccess` function under the defined assumptions has been shown before, both constraints do not influence the correct enforcement. Similarly, the RoleHierarchy relation from Definition (2.4) is enforced by adding additional calls to the assignment function for the superior role. This follows the same principles applied to other role assignments, so it also does not influence the enforcement's correctness. To correctly enforce a role's cardinality, we add a counter variable that is compared to the allowed upper bound each time an entity is assigned to the role. An example implementation can be examined in Listing 11.2. This private variable is only modified by the corresponding role assignment function by employing *solc-verify* modification specifiers like the one in line 2. To reason about the correct increase and decrease of this counter, the postconditions in line 3 and 4 are added and verified using *solc-verify*. The results can be examined in line 3 from Listing 11.3.

For the abstract concept of access context, two concrete implementations exist. Enforcing the BooleanVariableContext is achieved by employing the functionality of Solidity or *solc-verify* by adding the boolean conditions as explained in Chapter 7. To enforce a TimeContext,

Solidity's `block.timestamp` keyword is employed to access the current time for a function call. However, this time can be manipulated by the block's miner in a range of 30 seconds, according to Goldberg [68]. Due to this deviation, a correct enforcement of TimeContext isn't guaranteed.

To summarize, our approach correctly enforces modelled RBAC policies under the explained assumptions. Therefore, it is guaranteed that entities cannot modify variables they are prohibited from accessing. Additionally, unwarranted influencing accesses are detected and removed.

11.3. Case Study

According to Gustafsson [84, p. 2], "a case study can be defined as an intensive study about a person, a group of people or a unit, which is aimed to generalize over several units". Adding to this definition, Runeson and Höst [82] state that the target of a case study is a contemporary phenomena, which cannot be studied in isolation. Thus, real-world applications are taken as the foundation for collecting data and reasoning about a system's properties.

We follow these definitions by studying three different use cases describing real-world smart contract applications. The resulting smart contracts describe the units of our case study. By choosing these use cases independently from each other, we aim at reasoning about general properties of our approach. The phenomena we examine are correct enforcement and the completeness of the ACM, as we defined in the GQM plan from Section 11.1.

For this purpose, we consider three real-world software systems employing smart contracts and the underlying blockchain technology to provide their functionality. The Augur [85] use case, described in Section 11.3.1, implements a decentralized betting system for sporting events. In Section 11.3.2, Fizzy [86], a blockchain-based flight delay insurance, is introduced. The last use case is Palinodia by Stengele et al. [63], which was already mentioned in Section 3.4.2. It is a smart contract system for binary integrity protection. To achieve a correct and repeatable case study, we follow the guidelines by Runeson and Höst [82] to create the initial research design in Section 11.3.4. To underline the correct enforcement provided by the approach, we introduce violations to the implementation of the Augur and Fizzy use case in Section 11.3.5. We present the results of the case study regarding the detection of these violations as well as violations in the Palinodia use case in Section 11.3.6.

11.3.1. Augur

A more complex use case based on an already existing smart contract system is Augur [85], which is a secure and decentralized betting and prediction platform. On the platform, individuals speculate and bet on the outcome of real world events. Currently, only sporting events are supported although events from other domains like political votes are planned in the future. To allow users to bet on events without the need for a central organisation to confirm the outcome, Augur defines a five step process:

1. **Market Creation**: A market for an upcoming event is created by an arbitrary user. This user then chooses a designated reporter and a resolution source that should be used by the reporter to get the event outcome. Lastly, they post two bonds, the validity and creation bond. The validity bond is returned when the market is settled with a valid outcome. The creation bond is given back once the assigned reporter reports an outcome within the first 24 hours after the event has ended.

2. **Trading**: Market participants can trade shares of the different event outcomes until the event takes place. By doing so, they bid on the events development.

3. **Reporting**: After the event occurred, the outcome must be reported to the system in a way that stops a single entity from cheating by changing the results. This process begins with a 24 hour time window in which the designated reporter can employ the resolution source to report how the event ended and what outcome was achieved. If this report is not provided within the time period, the reporting is opened for all participants. Here, the first participant that provides an outcome is rewarded with the creation bond, when their reported results turns out to be the accepted outcome in the end of this step.

4. **Disputing**: After the initial outcome is reported, a dispute round is started. During this phase, any participant can dispute the tentative result by staking their system-wide reputation. This round has different outcomes, which either end the reporting phase directly by accepting the current outcome or prolong the decision process for up to 60 days. A more extensive summary can be found in [85].

5. **Market Settlement**: Finally, the results are evaluated, reporters and dispute participants are penalized or rewarded for their help and the market is settled. To do so, a participant either sells their shares to other participants or trade with the market platform directly.

Due to the complex nature of this use case, the system[1] consists of up to 30 different smart contracts. These are employed in combination with each other to achieve the aforementioned process. Since modelling all of these contracts, their functions and state variables as well as the different roles would exceed the scope of our thesis, we use a simplified version with only two smart contracts that focuses more on the general procedure than the technical details.

In the simplified version summarized in Figure 11.1, the first smart contract *MarketManagement* is used as a public starting point where any entity can call the `createNewMarket` function to create a new market. This market is represented by an instance of the second smart contract *Market*. For the *Market* contract, we establish the different roles based on the process description from [85]: The *market creator* is the address responsible for starting the market and thus sets the validity and creation bond as well as defining the designated reporter. This *designated reporter* can set the tentative outcome by reporting it once the event occurred. The *shareholder* role contains every entity that bought shares of the event during the Trading phase. They are permitted to dispute the outcome in the

[1] `https://github.com/AugurProject/augur` - Last accessed: 01.12.2021

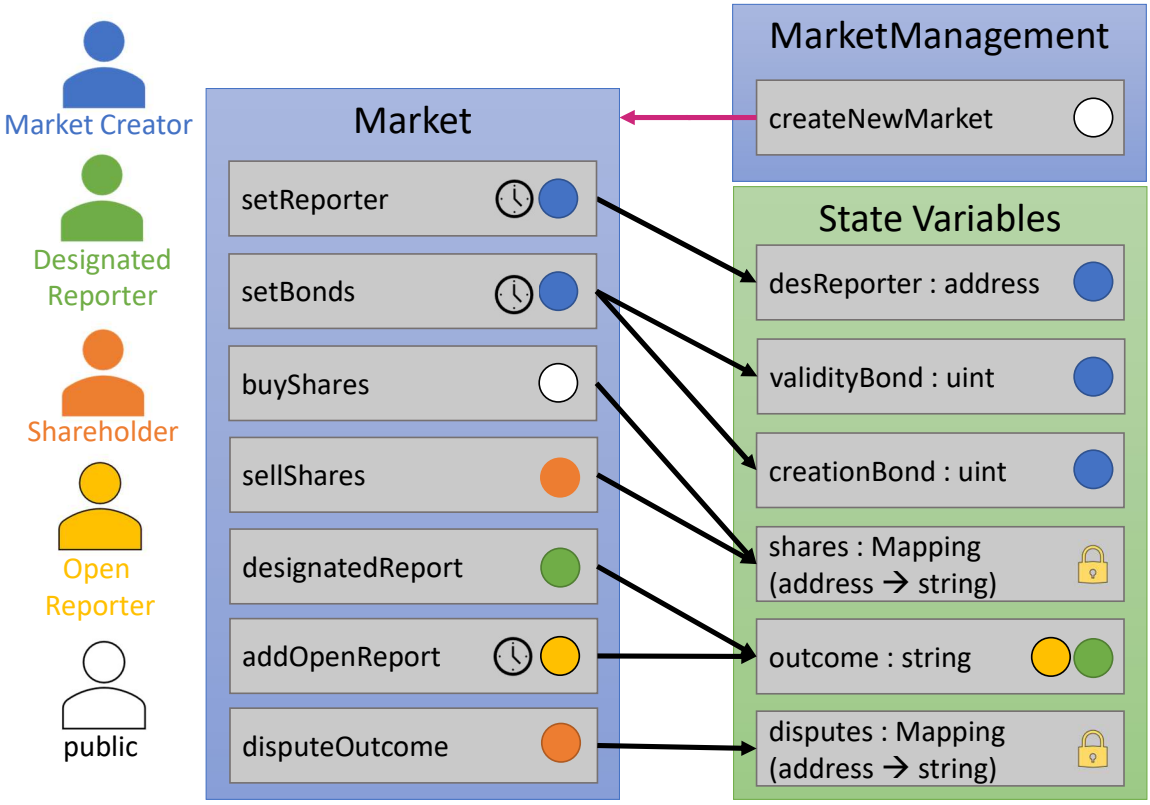

Figure 11.1.: Visualizing the simplified version of the Augur use case. This use case consists of the two smart contracts *MarketManagement* and *Market*. This diagram uses the visual elements from Figure 4.1.

dispute round as well as closing their market position during the Settlement phase. If the *designated reporter* fails to report within the first 24 hours after the event occurs, the first *shareholder* to use the open report feature becomes the *open reporter*. In addition to these restricted functions, the buying of shares is open to all users of the blockchain during the Trading phase. However, only a *shareholder* can sell their acquired shares. The current state of the system is represented through multiple boolean variables and functions that enable a transition between the process states.

Our presented approach contains multiple concepts that are applied to the simplified version of the Augur use case. The different roles with their differing access rights are modelled and enforced. Additionally, to keep track of the shares, the CSM is employed. Access is further restricted either through time (e.g. the *designated reporter* is only able to report in the first 24 hours after the event occurred) or through the current state of the smart contract (e.g. selling or buying shares is only allowed during the Trading phase). Therefore, information flows between variables are modelled and considered.

11.3.2. Fizzy

Created in 2017 by the insurance company AXA, Fizzy [86] was an experiment for combining insurances and their handling with the blockchain technology. Through this system, any user could purchase a flight delay insurance by providing the corresponding flight, their data and account information. This information is saved on the blockchain by a smart contract that connects to a global air traffic database. When this database notices a delay for the specified flight of more than two hours, the initially stipulated amount is automatically transferred to the user's account. Despite being one of the first applications by a big insurance company that combined insurances with smart contracts, the system was taken offline in 2019 due to a lack of interest from the public as well as the travel and airline industry [87]. Additionally, the general approach of combining the outcome of real world events with transactions on the blockchain is similar to the Augur use case from Section 11.3.1. However, the technical realization is different which leads us to consider both as use cases for the evaluation.

This use case consists of two smart contracts with two roles, *insurance company* and *insurant*, which are visualized in Figure 11.2. The first contract *InsuranceManagement* represents a manager for all covered insurances. It handles the connection to the global air traffic database and is employed to take out a new insurance. Changing the database can only be realized by the *insurance company*. Taking out a new insurance creates an instance of the *Insurance* contract and is publicly available. The entity that called this function is assigned the *insurant* role for the created *Insurance* contract. In this contract, the *insurant* role can change the account where the insured amount is paid to and it can cancel the insurance. The *insurance company* utilizes their connection to the air traffic database to check for a delay of the insured flight. If that is the case, the payout is triggered and the contract is closed.

With this setup, the Fizzy use case is used to explore the capabilities of the approach regarding role management and access restrictions. For example, the `payout` depends on the temporal constraint referencing the real-world time of the insured flight. The CSM is not necessary but balance modifications through functions are considered as well.

11.3.3. Palinodia

The Palinodia system developed by Stengele et al. [63] employs smart contracts to ensure access control for binary integrity protection. The use case consists of three smart contracts, which makes it suitable to handle the complete version of this use case instead of a simplification like with Augur from Section 11.3.1. The architecture of these three contracts as well as the connection to two roles is visualized in Figure 11.3. The following understanding of the smart contracts is based on the prototypical implementation by the authors and the description from the publication instead of or our interpretation. Therefore, the concrete technical details are integrated into our implementation and model.

One similarity of all three contracts is the role of the *root owner*, which is comparable to an admin role that is employed to manage all aspects of the contracts for the case that the other addresses are compromised. The first contract called *Software* describes a single software identity independent of its version or platform. A visual summary can be

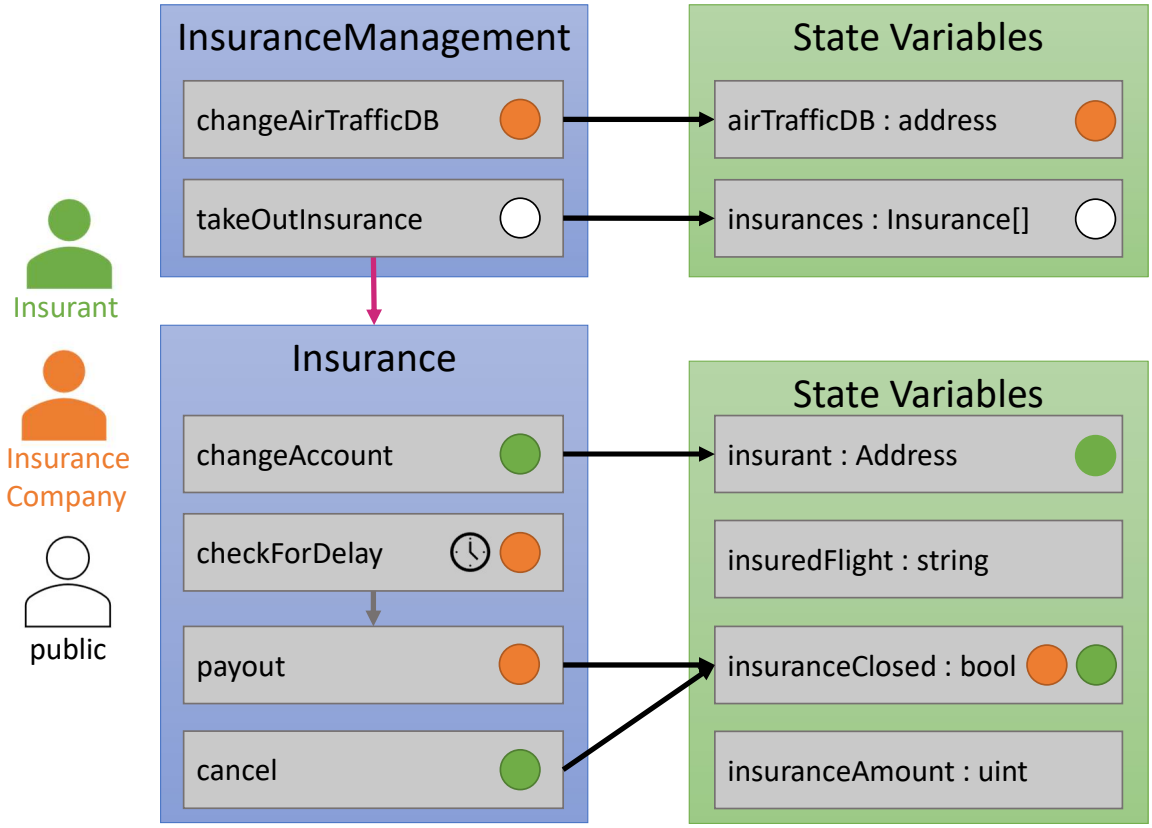

Figure 11.2.: Visualizing the two contracts *InsuranceManagement* and *Insurance* for the auction use case. This diagram uses the visual elements introduced in Figure 4.1.

examined in Figure 11.4. In this contract, the *root owner* is permitted to add new *developers* as well as changing the current *root owner*. The *developer* role is used to change the name of the software as well as (de-)registering a certain binary hash storage through their respective software distribution platform (SDP) ID. This ID can be changed later by the *platform* itself.

The smart contract *Binary Hash Storage* (BHS) from Figure 11.5 abstracts the SDP for a single software type, collecting all available versions as a hashed value. Similarly to the previous smart contract, the *root owner* can either change the *root owner* or add a *maintainer* for this platform. The roles *developer* and *maintainer* are mutually exclusive. The platform's *maintainer* is allowed to change the SDPs ID as well as publishing or revoking the hash of a software version. Lastly, anyone can initially register the software to maintain.

The developers also provide an *Identity Management* to store the addresses of authenticated users. This contract is presented in Figure 11.6. It consists of the *root owner* role, which is capable of changing the *root owner* as well a resetting the currently saved identities. Additionally, it can register new *identities* in the underlying data structure consisting of an array in combination with a mapping. All already registered *identities* can add and

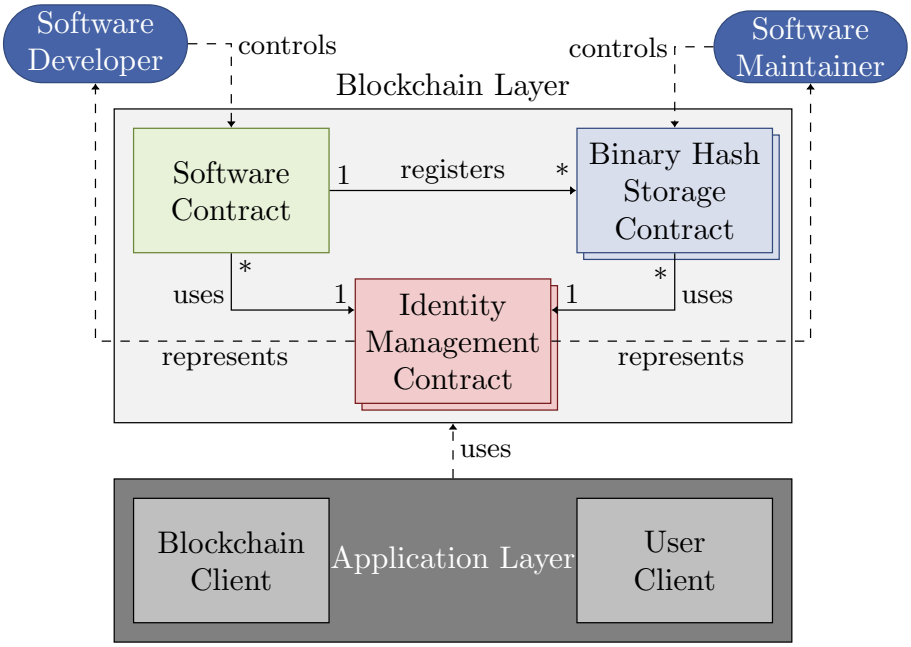

Figure 11.3.: Visualization of Palinodia's layered architecture containing three smart contracts and the two roles *Software Developer* & *Software Maintainer*. This figure is extracted from [63, Fig. 3].

remove other identities as well as changing their ID. Lastly, anyone can use this contract to verify the role and identify of a given address.

The Palinodia system enables the application of the role management in combination with handling access restrictions. Here, no temporal constraints are necessary but further role authorizations like role hierarchy and cardinality are employed. For example, the two roles *developer* and *maintainer* both inherit from the same superior role called *registered identity* in our implementation. However, balance modifications and the CSM are not applicable here.

11.3.4. Preliminaries

To conduct the planned case study in a structured manner that also focuses on the repeatability, we follow the case study guidelines presented by Runeson and Höst [82]. These guidelines include a checklist with questions that need to be answered before the case study is conducted. Due to the nature of our cases, not all of these ten questions provide relevant information. Therefore, question six, eight and ten are not considered in this section. In the following, the guidelines of Runeson and Höst [82] are mapped to our case study evaluation.

1. **What is the case and its units of analysis?**
   For our case study, we consider three different, concrete software systems that have been developed for the Ethereum blockchain. Augur [85] describes a decentralized betting platform, Fizzy [86] introduces an automatic flight insurance system and

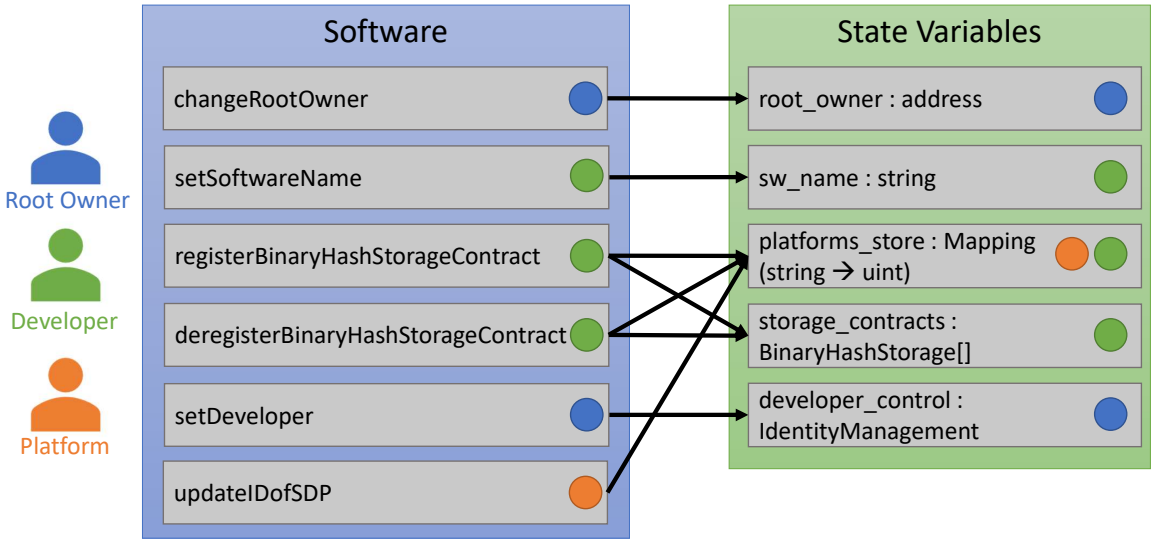

Figure 11.4.: Visualizing the roles, functions and variables as well as their connection to each other for the *Software* contract from the Palinodia use case. The visual elements are described in Figure 4.1.

Palinodia [63] concerns itself with binary integrity protection using smart contracts. All three of them are susceptible to violations stemming from a lack of access control or errors in the implementation of access control, so they all are suitable for the application of our approach.
We use multiple use cases instead of a single one, since the combination and comparison of their results allows for a stronger conclusion and different findings than a single use case [84].
For all three use cases, the created smart contracts describe the units of analysis. For the Augur use case, this includes the *Market* and *MarketManagement* contracts. Similarly, Fizzy implements the *Insurance* and *InsuranceManagement* contracts. The prototype implementation of the Palinodia use case consists of the three contracts *Software*, *BinaryHashStorage* and *IdentityManagement*.

2. **Are clear objectives, preliminary research questions, hypotheses (if any) defined in advance?**
   We defined the research questions regarding our approach in Chapter 11. Additionally, we created the GQM in Section 11.1, where we explain the questions we want to answer with this case study.
   The employed process is the same for all three use cases. We create instances of the ACM according to the use case description and generate the formally specified smart contract stubs. Afterwards, the generated contract skeletons are manually filled with their implementation. This implementation is checked for its correctness using the verification tools from Section 2.4.2. All in all, we follow the process from Chapter 10 for each use case.
   During this process, Augur and Fizzy are handled differently then Palinodia. Since the implementation for Augur and Fizzy is based on our interpretation instead of a

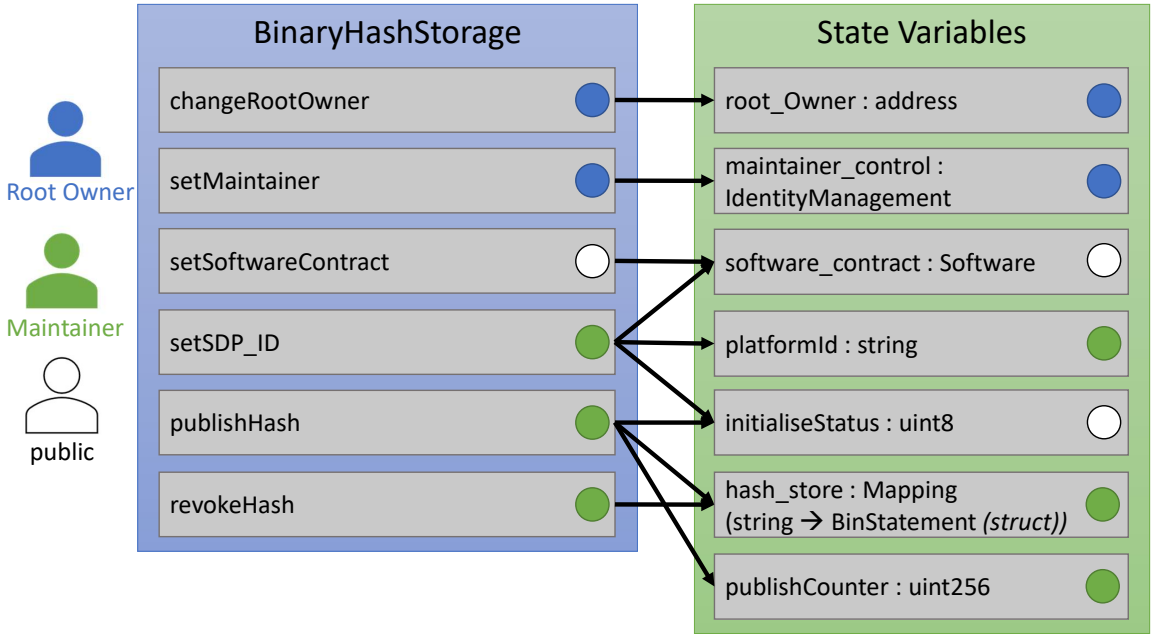

Figure 11.5.: Visualizing the roles, functions and variables as well as their connection to each other for the *BinaryHashStorage* contract from the Palinodia use case. The visual elements are described in Figure 4.1.

concrete implementation, we introduce violations to the underlying RBAC policies. Therefore, we verify the capability of our approach to detect these violations on the architectural and the source code level. This procedure evaluates whether the generated formal specifications enforce the access control policies correctly and is described in more detail in Section 11.3.5.

3. **Is the theoretical basis—relation to existing literature or other cases defined?**
   The basic definitions of the three use cases have been taken from [63, 85, 86]. From these three use cases, Palinodia was already covered by two other case studies. However, Friebe et al. [88] employ it in a case study for a different domain and the study by Schiffl et al. [55] applies a different approach to reason about its security, as explained in Section 3.3. Covered by one case study is Augur. However, the case study by Hobeck et al. [89] focuses on process mining instead of policy enforcement. Only the Fizzy use case has not yet been the target of any case studies.

4. **Are the authors' intentions with the research made clear?**
   Our intentions with the case study are explained in the GQM plan in Section 11.1.

5. **Is the case adequately defined (size, domain, process, subjects…)?**
   The use cases are described in Sections 11.3.1 to 11.3.3 and the implementation of their smart contracts is summarized in Table 11.1. However, the foundation for the concrete implementation differs for each use case. For the Augur use case, we describe a simplification in Section 11.3.1 that is based on the original system but is adapted to fit the scope of this thesis. Since Fizzy has been taken offline since its

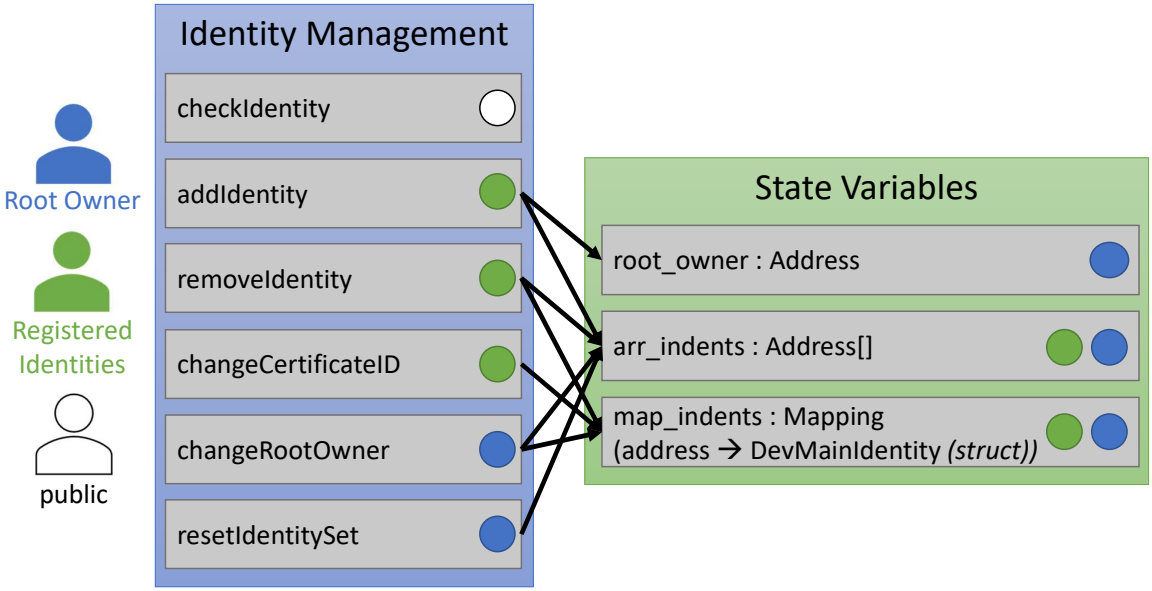

Figure 11.6.: Visualizing the roles, functions and variables as well as their connection to each other for the *IdentityManagement* contract from the Palinodia use case. The visual elements are described in Figure 4.1.

original publication [87], we base the concrete implementation on our understanding of the system's description by Benichou [86]. For the Palinodia system, we rely on the prototypical implementation provided by the authors. The three use cases have no connection to each other.

7. **Does the design involve data from multiple sources (data triangulation), using multiple methods (method triangulation)?**
   As explained in the answer to the first question, we rely on three use cases as the input for our case study. However, since we collect qualitative instead of quantitative data, there is no need for data triangulation. Since we apply the same method on all three use cases, there is no need for method triangulation.

9. **Is the specified case relevant to validly address the research questions (construct validity)?**
   All three use cases describe a scenario that is either already employed for access control purposes (Palinodia) or covers a domain susceptible to access control violations (Augur and Fizzy). Therefore, all use cases benefit from the correct enforcement of RBAC policies. Additionally, the detection of the manually introduced violations to the Augur and Fizzy use case underlines the correct enforcement of RBAC policies by demonstrating the capabilities of the presented approach.

Table 11.1.: Summarizing the implemented contracts analyzed by the case study. The number of roles per contract describes the amount of roles that are represented in the contract through Solidity `modifers`.

Case	Contract	#Functions	#Variables	#Roles	LoC
Augur	Market	13	19	5	269
Augur	MarketManagement	1	2	0	21
Fizzy	Insurance	4	6	2	78
Fizzy	InsuranceManagement	2	4	1	44
Palinodia	BinaryHashStorage	8	8	2	143
Palinodia	IdentityManagement	6	5	2	119
Palinodia	Software	9	6	4	135
Augur	2 Contracts	14	21	5	290
Fizzy	2 Contracts	6	10	2	122
Palinodia	3 Contracts	23	19	5	306
3 Use Cases	**7 Contracts**	43	50	12	718

11.3.5. Introducing Violations to Augur and Fizzy

To evaluate whether the presented approach detects violations to the RBAC policies, we introduce violations into the ACM instances and the implementations for the Augur and Fizzy use case. For both use cases, we added violations to the OCL constraints from Section 6.2 and the RBAC policies discussed in Section 9.2.2 on the architectural level. Therefore, we evaluate whether the generator's preprocessing step and soundness check succeed in identifying these cases. During the implementation, additional violations are introduced to examine the capabilities of our approach on the source code level. An overview for all violations introduced to the Augur use case can be examined in Table 11.2 and a similar overview for the Fizzy use case is presented in Table 11.3. In both tables, the introduced changes are described in the *Introduced Change* column. For each change, the *Resulting Violations* column summarizes the violations and problems that are introduced through that change. For this purpose, we state the category of the violation and than a specific description. For example, *RBAC* states that a violation to the RBAC formulas from Chapter 5 was introduced. Additionally, the violated formula is referenced.

Section 6.2 enhances the ACM by adding 15 OCL constraints to the model elements to explicitly formulate implicit assumptions about the model elements, their attributes and relationships. As described in Tables 11.2 and 11.3, the introduced violations to these 15 constraints are distributed between the Augur and Fizzy model, with three being part of the Augur use case and twelve being introduced to the Fizzy model. Additionally, the soundness check of the generator covers four RBAC formulae, as described in Section 9.2.2. We introduce one violation to each formula to the Augur use case. Since Formula (5.25) distinguishes between illegal modification and influencing access depending on the elements that are violated, we introduce one violation for each contingency. This results in five violations introduced to the ACM instances for the Augur use case.

Table 11.2.: This table provides an overview over the changes made to the model and implementation of the Augur use case from Section 11.3.1 during the case study. For each change, we state the resulting violations.
The changes 1 to 5 introduce violations on the architectural level whereas the changes 6 to 10 are implemented on the source code level.

#	Introduced Change	Resulting Violations
1	FunctionToStateVariableRelation element 'Agree with the reported outcome' references the CSM variable `disputes`.	OCL - `VariableTypeIsNoMappingWithAddressAsKeytype`
2	FunctionToCsmRelation element 'Outcome disputes are saved in CSM' references the variable `marketCounter`, which is no mapping. This also grants the *shareholder* role unwarranted access to the `marketCounter`.	OCL - `VariableTypeNeedsToBeMapping` OCL - `MappingKeyTypeNeedsToBeAddressOrAddressPayable` RBAC - Formula (5.25) *[modification]*
3	Role *shareholder* is allowed to modify the `agreeCounter` variable. However, it cannot access any function that modifies this variable.	RBAC - Formula (5.17)
4	Role *market creator* is allowed to modify the `reporterSet` variable, which influences the `designatedReporter` variable in the `setReporter` function. The *market creator* is not allowed to influence the `designatedReporter`.	RBAC - Formula (5.20) RBAC - Formula (5.25) *[influence]*
5	Role *designated reporter* is allowed to call the function `designatedReport`, which calls the function `report`. The *designated reporter* is not allowed to call `report`.	RBAC - Formula (5.24)
6	The `sellShares` function is modelled to increase the contract's balance.	*solc-verify* - illegal balance modification
7	The `closeDisputingWindow` variable deletes the `reportedOutcome` if a disagreement was found in the `disputeOutcome` function.	*solc-verify* - illegal access
8	The `allowOpenReport` function calls `claimReporterRole` function. This allows the `allowOpenReport` function to modify the `openReporter` variable.	*Slither* - illegal function call *solc-verify* - illegal access
9	Changes to the `designatedReporterReported` variable in the `allowOpenReport` function depend on the state of the `designatedReportAllowed` variable. Both variables can be modified by the *designated reporter*.	*Slither* - missing VariableToVariableRelation element
10	Changes to the `agreeCounter` in the `disputeOutcome` function depend on the state of the `disputesAllowed` variable. This grants the *designated reporter, open reporter* and the *manager* unwarranted influencing access.	*Slither* - insecure information flow
11	Changes to the `disagreeCounter` in the `disputeOutcome` function depend on the state of the `disputesAllowed` variable. This grants the *designated reporter, open reporter* and the *manager* unwarranted influencing access.	*Slither* - insecure information flow

Table 11.3.: This table provides an overview over the changes made to the model and implementation of the Fizzy use case from Section 11.3.2 during the case study. For each change, we state the resulting violations.
The changes 1 to 8 introduce violations to the OCL constraints on the architectural level. The changes 9 to 12 on the other hand are implemented on the source code level.

#	Introduced Change	Resulting Violations
1	Cardinality for the *insurance company* role is set to -2.	OCL - `CardinalityIsValid`
2	The *insurance company* role is set as a `superior` and a `prerequisite` of itself.	OCL - `NoRoleCanBePrerequisiteForItself` OCL - `NoRoleCanBeSuperiorToItself` OCL - `NoRoleInPrerequisiteAndSuperiorSet`
3	The two roles *insurant* and *insurance company* are mutually exclusive. Additionally, the *insurance company* is set as the `superior` to the *insurant*.	OCL - `RolesCannotBeInHierarchy-OrPrerequisite`
4	The set BooleanValueContext for the FunctionToStateVariableRelation 'Change the insured account' does not reference a boolean variable.	OCL - `VariableTypeNeedsToBeBoolean`
5	The FunctionToStateVariableRelation 'Checking for a delay indirectly closes the insurance' references a TimeContext element.	OCL - `ForbidTimeContextConditions`
6	The model for the *Insurance* contract includes one OverrideFunction and one LocalFunction.	OCL - `NoFunctionOverrides` OCL - `NoAdditionalLocalFunctions`
7	The model for the *Insurance* contract includes one FunctionBalanceModification element for each function. Additionally, a second element referencing the `payout` function is added.	OCL - `BalanceModificationsReference-DifferentFunctions` OCL - `NoMoreBalanceModifications-ThanFunctions`
8	One FunctionBalanceModification element referencing the `payout` function allows for an increase to the contract's balance. However, the `payout` function is not marked as `payable`.	OCL - `BalanceModificationsRegarding-ThisContractNeedPayableFunction`
9	The `payout` function is modelled to increase the contract's balance.	*solc-verify* - illegal balance modification
10	The `cancelInsurance` function calls the `payout` function. This allows the `cancelInsurance` function to increase the contract's balance.	*Slither* - illegal function call *solc-verify* - illegal balance modification
11	The `cancelInsurance` function sets the `insurant` variable to zero.	*solc-verify* - illegal access
12	Changes to the `insurant` in the `changeAccount` function depend on the state of the `insurance-Closed` variable. This grants the *insurance company* unwarranted influencing access.	*Slither* - insecure information flow

After removing the violations on the architectural level and generating the formally specified smart contracts, we introduce additional violations in the implementation to show how our approach detects violations to the modelled RBAC policies on the source code level. Beginning with the verification done by employing *solc-verify*, we distinguish between the three possible outcomes explained in Section 10.1.2. We introduce three violations where a function modifies a variable it is not meant to modify, possibly allowing a role unwarranted access to that variable. Two of these violations are introduced directly and one is a byproduct of the illegal function call by the `allowOpenReport` function to the `claimReporterRole` function. This allows the `allowOpenReport` function to modify the `openReporter` variable without permission. Additionally, we introduce three violations to the generated postconditions validating the balance modifications as described in Section 9.1. One of the two violations added to the Fizzy use case is a byproduct of the illegal function call by the `cancelInsurance` function to the `payout` function, which allows the `cancelInsurance` function to increase the contract's balance. In all three cases, a function that is only allowed to increase the contract's balance decreases it instead.

In addition to *solc-verify*, the `influence-and-calls` printer from Section 8.2 is employed to analyze the implemented contracts. To investigate its capabilities, we introduce three types of violations to the implementation. These violations have been described in Section 10.1.3. First, we add two illegal function calls. Additionally, we add four information flows that do not occur in the ACM instances. Of the three information flows introduced to the Augur use case in Table 11.2, the one introduced from the `designatedReportAllowed` variable to the `designatedReporterReported` variable does not lead to a violation of the modelled RBAC policies. The other two information flows on the other hand, connecting the `designatedReportAllowed` variable to the `agreeCounter` and `disagreeCounter`, lead to insecure information flows. The information flow in the Fizzy use case, where the `insurant` is influenced by the `insuranceClosed` variable, is also insecure. It allows the *insurance company* to influence the `insurant` variable, despite being prohibited from doing so.

11.3.6. Results

During the case study, we introduced violations to the Augur and Fizzy use case as explained in Section 11.3.5. To examine the capabilities of the soundness check on the architectural level, we introduced 15 violations to the OCL constraints and five violations to the underlying RBAC model. All of these twenty violations are detected by employing the presented approach, as is visualized in Table 11.4. The resulting logging files for both use cases can be examined in Appendix A.5.

After removing these violations and generating the formally annotated smart contracts, we added twelve violations on the source code level. Six of these violations should be detected by *solc-verify* and six should be uncovered by the `influence-and-calls` printer. Both tools detect 100% of the violations that were manually introduced, as can be examined in Table 11.5.

In addition to the detection of the manually introduced violations, the case study also employs both verification tools to analyze the implementation for additional violations not introduced on purpose. Beginning with the verification done by *solc-verify*, no additional

Table 11.4.: Summarizing the amount of manually introduced violations to the RBAC policies as well as the OCL constraints on the architectural level to the Augur and Fizzy use case. Additionally, this table visualizes the amount of violations that the generator's preprocessing step detected.
All in all, 100% of the introduced violations are detected on the architectural level.

	Case	OCL Violations	RBAC Violations
Introduced	Augur	3	5
	Fizzy	12	0
Detected	Augur	3	5
		100%	100%
	Fizzy	12	0
		100%	-

Table 11.5.: This table visualizes the amount of violations manually introduced in the implementation of the Augur and Fizzy use case. The columns *Illegal access* and *Violated postconditions* describe violations that can be identified using *solc-verify*, the other three columns represent violations detected by the `influence-and-calls` *Slither* printer.
All in all, 100% of the introduced violations are detected.

	Case	Illegal access	Violated postconditions	Illegal Function call	Missing element	Insecure information flow
Introduced	Augur	2	1	1	1	2
	Fizzy	1	2	1	0	1
Detected	Augur	2	1	1	1	2
		100%	100%	100%	100%	100%
	Fizzy	1	2	1	0	1
		100%	100%	100%	-	100%

violations were detected in the three use cases. However, in the *BinaryHashStorage* contract from the Palinodia system, two functions could not be verified by *solc-verify*. These two functions rely on an unsupported binary operator, which cannot be verified by the current version of *solc-verify* [90].

To further analyze the implemented contracts, the `influence-and-calls` printer from Section 8.2 is employed. This printer investigates each contract regarding the transitive closure of function calls as well as information flows between variables. The results are categorized according to the explanations from Section 10.1.3. Regarding the function calls, five calls have been detected by the printer for all three use cases. This does not include the two erroneous function calls that we introduced in Section 11.3.5, since they are summarized in Table 11.5 instead. From these five function calls, only one is not part of the modelled ACM instances, the remaining four are described in the corresponding model instances. This one illegal function call exists in the *Software* contract in the Palinodia use case. However, after manually analyzing the two corresponding functions, we conclude

Table 11.6.: Summarizing the results found by the `influence-and-calls` printer for the three use cases. This table summarizes the found information flows between variables and sorts the ones not modelled in theACM instances into one of three categories. These categories have been defined in Section 10.1.3. The manually introduced violations are already covered in Table 11.5 and are thus not included in this table.

Case	Contract	Information flows	Flows not in ACM	Nothing	Missing elements	Violations	Unique violations
Augur	Market	33	28	1	5	22	13
Augur	MarketManagement	0	0	0	0	0	0
Fizzy	Insurance	3	3	0	3	0	0
Fizzy	InsuranceManagement	0	0	0	0	0	0
Palinodia	BinaryHashStorage	8	8	1	3	4	2
Palinodia	IdentityManagement	5	5	3	2	0	0
Palinodia	Software	3	3	1	1	1	1
Augur	2 Contracts	33	28	1	5	22	13
			100%	~ 4%	~ 17%	~ 79%	
Fizzy	2 Contracts	3	3	0	3	0	0
			100%	-	100%	-	
Palinodia	3 Contracts	16	16	5	6	5	3
			100%	~ 31%	~ 38%	~ 31%	
3 Use Cases	7 Contracts	52	47	6	14	27	16
			100%	~ 13%	~ 30%	~ 57%	

that this illegal call does not introduce unwarranted access to variables for any role. If unwarranted access would be granted through this call, *solc-verify* would detect a violation to the modification specifiers.

To reason about insecure information flows in the implemented contracts, the `influence-and-calls` printer is employed as well. Its results can be examined in Table 11.6. Beginning with the *Information flows* column, all detected information flows that were not deliberately introduced in Section 11.3.5 are collected. The information flows that we added in Section 11.3.5 are summarized in Table 11.5 instead. With the *Flows not in ACM* column, the amount of information flows that are not part of the ACM instances is summarized. Each of these information flows is categorized regarding the possible violations they introduce, as we explained in Section 10.1.3. Since the transitive closure may discover information flows that do not occur in the implementation due to the overapproximation explained in Section 8.2.3, these discoveries are counted in the *Nothing* column. If the variable connection does not introduce an insecure information flow, a missing VariableToVariableRelation element in the ACM instances is reported in the *Missing elements* column. However, all insecure information flows are counted in the *Violations* column. To further examine these violations, they are compared and only distinct access gained by a set of

roles to a variable is counted in the *Unique violations* column. The concrete explanation for the classification of each information flow can be examined in Appendix A.6, where we also provide summaries with the Tables A.2 to A.4.

To describe some examples, three classifications from the *Market* contract from the Augur use case are discussed. The first information flow is an indirect influence detected between the `designatedReporterReported` and the `creationBondPaidOut` variable and it occurs inside of the `sentCreationBond` function. When analyzing the concrete implementation, this influence does not occur since the *creationBondPaidOut* is changed the same way in either branch of the condition. Therefore, this information flow is classified as *Nothing*. Another variable influencing the `creationBondPaidOut` variable in the `sentCreationBond` function is the `settlementPhaseActive` variable. This indirect influence occurs but both variables can be modified by the *manager* role. Therefore, a missing VariableToVariableRelation element is reported. Additionally, the `openReporter` variable also indirectly influences the `creationBondPaidOut` variable in the `sentCreationBond` function. However, the `openReporter` can be modified by the *shareholder* role whereas the `creationBondPaidOut` variable can only be modified by the *manager*. This leads to a violation where the *shareholder* can illegally influence the `creationBondPaidOut` variable.

Regarding the information flows between variables, 52 are detected throughout all three use cases. 47 of these are not part of the underlying ACM instances and thus are categorized regarding the risk they introduce. Approximately 57% of these influences describe a possible violation to the underlying RBAC policies by allowing a role to influence a variable they are prohibited from accessing. Only about 13% of these information flows are introduced by the overapproximation described in Section 8.2.3. Another aspect of these results is identified when comparing the results for the Augur and Palinodia use case. As summarized in Table 11.1, both use case define a similar scope regarding the amount of Solidity code, variables and roles. However, Augur contains 13 unique insecure information flows whereas Palinodia only contains three, even though Palinodia contains 23 state variables instead of 14 state variables in the Augur use case.

During the case study, limitations to the ACM were discovered. For one, it provides no capabilities for marking a variable as publicly accessible. For example, the `software_contract` variable in the Palinodia use case can be modified by any entity, which cannot be expressed in the ACM. During the creation of the model, this was circumvented by allowing all available roles to access this variable.

Additionally, as we mentioned in Section 6.1, BooleanVariableContext elements also describe an indirect information flow between two variables, if these elements are used to restrict FunctionToVariableRelation elements. However, they are currently not analyzed during the soundness check. The last limitation was found for the Fizzy and Augur use case. Both rely on the input of external sources which provide the system with information about a real-world event. According to Ladleif and Weske [46] and Zupan et al. [47], these external sources are called *oracle*s. Currently, the ACM does not model these oracles so their behaviour is approximated by employing TimeContext elements.

11.4. Metamodel Coverage Analysis

To analyze the completeness of the ACM by detecting redundant model elements, we employ a *Metamodel Coverage Analysis* proposed by van Amstel and van den Brand [83]. This analysis examines the input metamodel for a model transformation and visualizes the elements, attributes and relationships that are utilized by the transformation. Based on this visualization, the completeness of the transformation is analyzed by showing whether any elements are not covered. For our approach, the input metamodel is the ACM from Section 6.1 and the output is the annotated source code that covers the RBAC policies as described in Chapter 7. The transformation we consider is the source code generation described throughout Chapter 9.

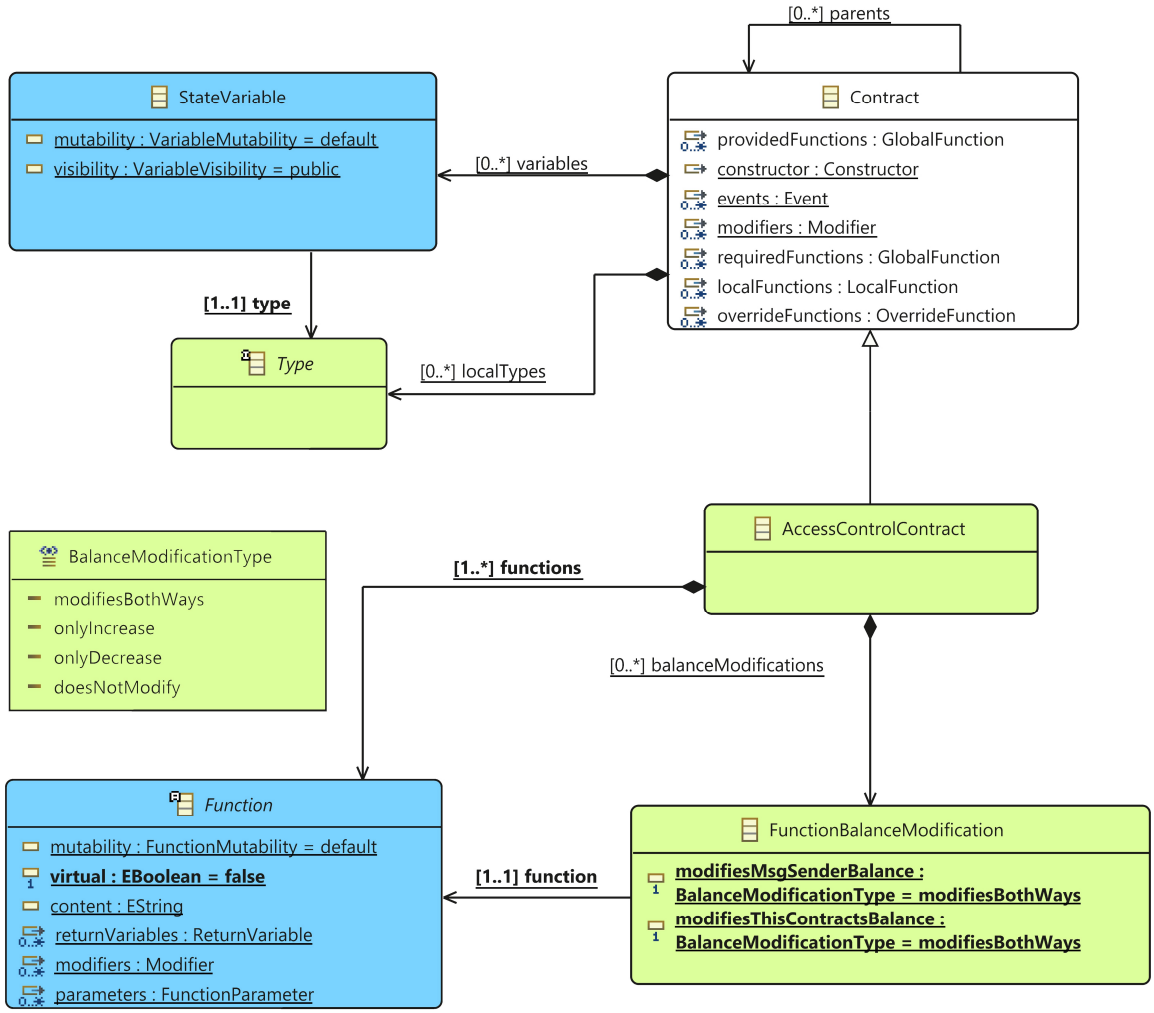

Figure 11.7.: Metamodel coverage defined by van Amstel and van den Brand [83] for the *SmartContractModel* from the ACM in Section 6.1. Here, any attribute or relation that is covered by the generator is underlined. If a model element is employed during the generation, it is colored in green. Any element that is additionally employed during the soundness check is marked in blue. White elements and not covered by the generator.

Table 11.7.: Summarizing the results found by the Metamodel Coverage Analysis [83] for the two metamodel packages of the ACM. This table summarizes the amount of elements, attributes and relationships and whether they are covered by the implemented generator or not. For the model elements the coverage is categorized based on the part of the implementation that covers them (the soundness check, the generation or both). Regarding the complete coverage of model elements, attributes and relations, the generator reaches a coverage of approximately 90%.

Package	# Elements	Soundness Check	Generation	Both	Not covered	# Attributes & Relations	Covered	Not covered
SmartContractModel	7	0	2	4	1	24	20	4
AccessControlSystem	14	2	5	5	2	32	31	1
ACM	21	2	7	9	3	56	51	5
		~ 10%	~ 33%	~ 43%	~ 14%		~ 91%	~ 9%

To visualize the metamodel coverage, the authors color every used model element in grey. Attributes and relationships that are employed are marked by underlining them in the visualization. To increase the expressiveness of this analysis, we distinguish for each model element if it is used during the generation, during the soundness check or during both phases. For this, we color model elements only employed during the generation green, elements used to verify the systems soundness red and all elements that fulfill both criteria are marked in blue. If a model element is not covered, it stays white. Attributes and relationships are handled the same way the authors propose and thus are underlined when they are employed anywhere in the generator.

The visualization for both ACM packages can be examined in Figures 11.7 and 11.8. The quantitative results are summarized in Table 11.7. Throughout the whole metamodel, only three elements are not covered by the generator. Two of them are the *FunctionToVariableRelation* and the *Context* from the *AccessControlSystem* package. Since both describe abstract classes with concrete implementations through other elements, they do not express a gap in the ACM coverage. Similarly, the Contract element from the *SmartContractModel* is indirectly covered through the AccessControlContract class inheriting its properties. Regarding the coverage of relationships between metamodel elements, all described connections are covered by the generator.

Beginning with the *SmartContractModel*, the coverage of metamodel attributes are discussed. Here, the attributes that are not covered all belong to the Contract element and are thus inherited to the AccessControlContract element. The two attributes `providedFunc-`

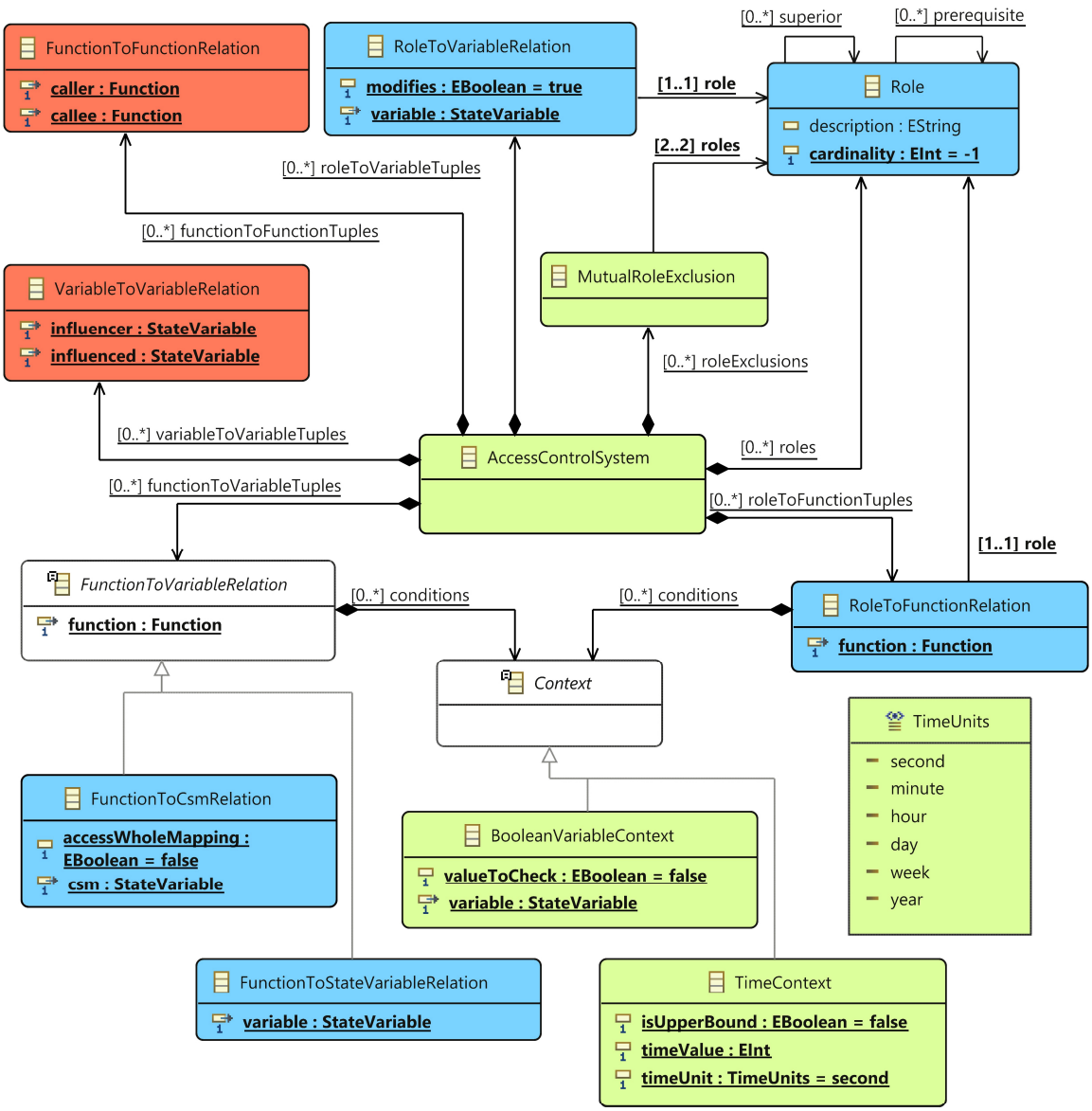

Figure 11.8.: Metamodel coverage defined by van Amstel and van den Brand [83] for the *AccessControlSystem* from the ACM in Section 6.1. Here, any attribute or relation that is covered by the generator is underlined. If a model element is employed during the generation, it is colored in green, whereas elements employed only during the soundness check are marked in red. If an element is used for both phases, it is colored blue and white elements and not covered by the generator.

tions and requiredFunctions are necessary for the connection of multiple smart contracts in the *SolidityMetaModel* [66]. However, in the ACM this is done by connecting functions directly using the FunctionToFunctionRelation element. The other two attributes that are not covered (localFunctions and overrideFunctions) have been excluded deliberately, as we explained in Section 6.2. In the *AccessControlSystem*, only one attribute is not covered during the generation. As we explained in Section 6.1, the description of each role is an optional parameter utilized to add more information to the model elements on the architectural level.

11.5. Discussing the Results

This section summarizes the results of the evaluation by answering the questions from Section 11.1. According to these answers, the fulfillment of the defined goals is evaluated. Based on the achievement of these goals, the research questions from Chapter 11 are resolved.

The first goal **G1** focuses on the correct enforcement of modelled RBAC policies in the generated source code. To support this goal, **Q1** directly poses the question whether our approach provides a correct enforcement. To evaluate this question, we present the argument in Section 11.2 (**M1.1**), which reasons about unwarranted modification access and if it still occurs in our approach. The presented argument does not correspond to a sound and formal proof. However, a correct enforcement is still achieved under the given assumptions. As explained in Section 11.2, applying the TimeContext model element introduces an attack surface for the miner by underlying uncertainties of up to 30 seconds [68]. Also, the analysis of insecure information flows between variables is handled correctly but the completeness of the found flows is not guaranteed. To reason about an additional metric **M1.2**, we introduced twenty violations to the RBAC policies on the architectural level and twelve on the source code level, visualized in Tables 11.4 and 11.5. All of these violations are detected by employing our approach, resulting in a coverage of 100%.

However, additional limitations have been uncovered by the employed case study. As described in Section 11.3.6, the verification by *solc-verify* does not handle the usage of binary operators, aborting the verification of two out of 43 functions that apply them. This describes a limitation in the current implementation of *solc-verify* [90] that restricts the correct enforcement to contracts that do not employ binary operations in Solidity.

Apart from this limitation to *solc-verify*, detecting insecure information flows is problematic in sequential use cases like Augur, where 28 information flows between 19 variables in the *Market* contract have been detected. From these information flows, 79% lead to 22 violations with 13 of them being unique. In the Palinodia use case with a comparable amount of variables (19) and similar size (both have about 300 lines of code (LoC)), only 16 information flows have been discovered. From these, only approximately 31% lead to violations, with three of them being unique. One difference between both use cases that could have lead to this discrepancy is the sequence of states inside the contracts. In the Augur use, the different process states are modelled in a linear sequence, meaning that each process phase relies on the completion of the previous phase. If a role like

the *market creator* is responsible for the market creation and set up, this role influences almost all aspects of the remaining process. The Palinodia system on the other hand arranges its states in a non-linear sequence where different process phases do not depend on each other but happen in parallel. Therefore, the Augur use case entails more insecure information flows then the Palinodia use case. However, further research needs to be done to empirically confirm or deny this interpretation.

From these metrics, it is concluded that **G1** is achieved under the explained restrictions. Our approach enforces the modelled RBAC policies regarding modification access correctly, as long as the assumptions from Section 11.2 are not violated. Additionally, the smart contracts are prohibited from employing binary operations or the TimeContext element from the ACM, if a correct enforcement is guaranteed. When influence relations through information flows are considered, the completeness is not guaranteed so the enforcement's correctness is limited.

To evaluate the completeness of the created metamodel, **G2** is evaluated. Regarding the metric **M2**, the case study from Section 11.3 is analyzed regarding limitations to the ACM. Therefore, limitations regarding the expression of aspects relevant to model RBAC policies are investigated, beginning with the modelling of publicly available variables. As explained in Section 11.3.6, a state variable cannot be marked as accessible to any entity. Additionally, including information from external sources through the employment of oracles [46, 47] cannot be achieved using the ACM.

In addition to these identified limitations, the metamodel coverage analysis from Section 11.4 (**M3**) is employed to answer **Q3**. By visualizing the elements that are utilized throughout the generation, possible overhead in the ACM is identified. However, no unnecessary elements are discovered since approximately 86% of model elements are employed during the generation. Regarding the attributes and element relationships, approximately 91% are covered by the generator, resulting in a coverage of approximately 90% for the whole ACM. The remaining elements and attributes are either abstract or inherited from the extensive *SolidityMetaModel* [66], as explained in Section 11.4. Therefore, they are excluded on purpose.

Based on the executed evaluation, the ACM does not consist of any unnecessary model elements since all relevant elements are employed during the generation and thus enforced through the source code elements. However, some limitations to the expressiveness are detected regarding public variables and oracles. These limitations must be addressed in the future.

11.6. Threats to Validity

According to Runeson and Höst [82, p. 153], "the validity of a study denotes the trustworthiness of the results". This is influenced by different factors like the researcher's bias or the selected use cases we investigated during the case study. To address the possible threats to our evaluation's validity, we examine internal, external and construct validity as well as repeatability separately.

Construct Validity: This validity explores the connection between the measurements and the questions these measurements should answer. To support a structured evaluation

where the connection between metrics, questions and goals is clear, we created the GQM plan from Section 11.1, which we followed with our evaluation. As we already explored in question nine of the case study checklist from Section 11.3.4, the chosen use cases for the case study all describe real-world smart contract system whose process relies on RBAC policies. By manually introducing violations to two of the three use cases, the detection of these violations further supports the reasoning about the thesis' goals.

Internal Validity: If the evaluation is influenced by another factor than the one it focuses on, threats to the internal validity are introduced. As we explained in Section 11.3, for two of the three use cases we studied (Augur and Fizzy), the implementation is based on our interpretation instead of a prototype. Additionally, the definition of roles is not directly given by the use case authors but based on the perceived roles mentioned in the use case description. Also, example violations were introduced to the architectural model and the implementation of these two use cases. For the Palinodia use case, the implementation as well as the concrete definition of roles are provided by Stengele et al. [63]. These interpretations, as well as the introduction of violations, are done solely by the author of this thesis, who is also the only one executing the evaluation. Both of these aspects introduce possible threats due to the author's bias towards their approach. Additionally, the concrete implementation was created by the author using the Solidity programming language, with whom the author had no prior experience. This lack of experience introduces additional threats to the internal validity as more experienced developers would have implemented more optimized code, possibly negating violations found during the case study.

External Validity: The goal of our case study from Section 11.3 is to reason about the three investigated use cases in a way that allows us to generalize about the approaches properties for all possible use cases. How well our study supports this goal is examined with the external validity. As we explained in Section 11.3.4, we chose three use cases as the input for our case study instead of one to increase its generalisability [84]. However, the selection of these three use cases was not done at random but by choosing systems that are suitable for describing RBAC constraints. Additionally, the inclusion of aspects to consider for the two use cases based on the authors interpretation (Augur and Fizzy) was done to support the applicability of RBAC constraints. Also, the amount of use cases is not sufficient to conclude general statements about our approach but it is sufficient enough regarding the scope of this thesis.

Repeatability: As the name suggests, threats to this validity describe problems when other researchers try to emulate and repeat this evaluation. To allow for the repetition of the case study, we introduce the goals and questions to answer before creating the use case artifacts with the GQM plan. Additionally, all artifacts we created for the use cases as well as the used metamodel and generator are available in our GitLab repository [91].

12. Conclusion and Future Work

12.1. Conclusion

In this thesis, we created an approach to correctly enforce RBAC policies for smart contracts in the source code after initially describing them on the architectural level. This thesis addresses and mitigates the limitations to the original concept by Reiche et al. [1]. To formally define the underlying RBAC policies, we formalized and extended the standard RBAC model by Sandhu, Ferraiolo, and Kuhn [2]. This extension includes the role authorization constraints by Ben Fadhel, Bianculli, and Briand [5] and information flows between variables that allow roles to influence variables they are prohibited from accessing.

To describe these extended policies on the architectural level, we designed the AccessControlMetamodel (ACM). It extends the *SolidityMetaModel* by Dietrich and Reiche [66] to describe the smart contracts structure as well as incorporating the extended formal RBAC model. To correctly enforce these policies on the source code level, we employ *solc-verify* and *Slither*. For the purpose of reasoning about access control policies with *solc-verify*, formal specifications are generated for each function. To support the analysis of insecure information flows, we extended the *Slither* framework by creating the `influence-and-calls` printer.

For the purpose of automatically enforcing the modelled policies, we implemented a source code generator with *Xtend*. This generator creates the modelled smart contract, an additional access control contract and the formal specifications and Solidity modifiers to enforce the RBAC policies. For this generation, we introduced a mapping between the metamodel and source code elements. Additionally, we define the resulting process when applying the presented approach. This definition includes the process phases, the artifacts connected to each phase and which roles and tools are responsible for the creation of these artifacts.

To evaluate different properties of the presented approach, we execute an evaluation that follows a predefined GQM plan [81]. It consists of an argument for the approach's correctness, a case study employing three different, real-world use cases and a metamodel coverage analysis [83]. To verify that the presented approach detects violations to the underlying RBAC model, we introduced twenty violations on the architectural level and twelve violations on the source code level during the case study. All of these violations are uncovered by applying the presented approach. Additionally, the argument for correctness shows that the presented approach correctly enforces the modelled RBAC policies as long as the assumptions from Section 11.2 hold and no influence relations are considered. These influence relations are handled correctly but the completeness of the `influence-and-calls` printer is not guaranteed. Regarding the completeness of the ACM, the metamodel

coverage analysis demonstrates that approximately 90% of the model elements are covered, with the remaining 10% being excluded on purpose.

12.2. Future Work

To provide directions for future research, we summarize possible extensions to the presented approach. Beginning with the ACM described in Chapter 6, the role assignment could be incorporated into the metamodel to restrict functions from setting any role and thus mitigating the possible errors during the implementation. Additionally, incorporating the role assignment would allow the reasoning about the enforcement's correctness from Section 11.2 to rely less on assumptions about the developer's behaviour. Additionally, the ACM could extend its coverage of possible balance modifications by enabling the architect to define a balance modification restriction for every variable with the `address` data type. Similarly, the coverage of the CSM can be extended to allow for more complex descriptions of the access location or cover other complex data types like `struct` or arrays. For the purpose of increasing the metamodel's expressiveness, the limitations described in Section 11.5 can be circumvented by adding support for oracles and public variables.

Regarding the enforcement of RBAC policies on the source code level, the current approach relies heavily on the functionality provided by *solc-verify*. However, as we explained in Section 2.4.2.4, similar formal verification tools like *SciviK* exist. By adapting the enforcement on the source code level to use e.g. postconditions, multiple formal tools could be employed and their results compared or used in tandem with each other. However, this adaption to the RBAC enforcement can lead to other disadvantages we examined in Chapter 7. In combination with the implementation of *SciviK*, which is not yet publicly available, this improvement can be considered once additional verification tools are available.

To decrease the overhead when analyzing the results provided by the `influence-and-calls` printer from Section 8.2, it could be transformed into a *Slither* detector searching for specific violations in the implementation instead of printing a summary that is analyzed manually by the stakeholders. This improvement could be combined with adding an automatic implementation supporting the last steps of the development process from Chapter 10 that are currently handled manually. This includes starting the verification by employing the verification tools, analyzing their results and communicating these results back to the stakeholders.

Mitigating the threats to validity from Section 11.6, the evaluation could be improved by employing a more extensive case study relying on even more different smart contract systems. This second case study could also be employed to investigate the relationship between sequential smart contracts and insecure information flows described in Section 11.3.6. Additionally, a formal and sound proof regarding the correct enforcement of RBAC policies can be conducted to enhance the reasoning provided in Section 11.2. Since our approach involves interaction with stakeholders, a usability study can also be employed in the future to identify potential optimizations regarding the support for architects and developers.

Bibliography

[1] Frederik Reiche et al. *Modeling and Verifying Access Control for Ethereum Smart Contracts*. Preprint. Ed. by Karlsruher Institut für Technologie. Karlsruhe: Kompetenzzentrum für angewandte Sicherheitstechnologie, Institut für Programmstrukturen und Datenorganisation, and Institut für Theoretische Informatik, 2021. DOI: 10.5445/IR/1000129607. URL: https://publikationen.bibliothek.kit.edu/1000129607 (visited on 05/05/2021).

[2] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. "The NIST Model for Role-Based Access Control. Towards A Unified Standard". In: *5th ACM Workshop on Role-based Access Control (RBAC'00). Proceedings: 26-28 July 2000* (Berlin, Germany,). Ed. by Klaus Rebensburg, Charles Youman, and Vijay Atluri. Association for Computing Machinery. New York: ACM, 2000, pp. 47–63. ISBN: 158113259X. DOI: 10.1145/344287.344301.

[3] Zibin Zheng et al. "An Overview on Smart Contracts: Challenges, Advances and Platforms". In: *Future Generation Computer Systems* 105 (2020), pp. 475–491. ISSN: 0167739X. DOI: 10.1016/j.future.2019.12.019.

[4] Ákos Hajdu and Dejan Jovanović. "solc-verify: A Modular Verifier for Solidity Smart Contracts". In: *11th International Conference on Verified Software, Theories, Tools and Experiments (VSTTE 2019). Proceedings: 13-14 July 2019* (New York City, NY, USA,). Ed. by Supratik Chakraborty and Jorge A. Navas. Cham: Springer International Publishing, 2019, pp. 161–179. ISBN: 978-3-030-41600-3.

[5] Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. "A comprehensive modeling framework for role-based access control policies". In: *Journal of Systems and Software* 107 (2015), pp. 110–126. ISSN: 01641212. DOI: 10.1016/j.jss.2015.05.015.

[6] Tachio Terauchi and Alex Aiken. "Secure Information Flow as a Safety Problem". In: *12th International Static Analysis Symposium (SAS 2005). Proceedings: 07-09 September 2005* (London, United Kingdom,). Ed. by Chris Hankin and Igor Siveroni. Lecture notes in computer science, 0302-9743 3672. Berlin and London: Springer, 2005, pp. 352–367. ISBN: 978-3-540-28584-7. DOI: 10.1007/11547662_24.

[7] Josselin Feist, Gustavo Grieco, and Alex Groce. "Slither: A Static Analysis Framework for Smart Contracts". In: *2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB 19). Proceedings: 27. May 2019* (Montreal, QC, Canada,). IEEE, 2019, pp. 8–15. ISBN: 978-1-7281-2257-1. DOI: 10.1109/WETSEB.2019.00008.

[8] Colin Atkinson and Thomas Kühne. "Model-Driven Development: A Metamodeling Foundation". In: *IEEE Software* 20.5 (2003), pp. 36–41. DOI: 10.1109/MS.2003.1231149.

[9] Marco Brambilla, Manuel Wimmer, and Jordi Cabot. *Model-Driven Software Engineering in Practice*. Second Edition. Vol. 4. Synthesis lectures on software engineering. San Rafael, California: Morgan & Claypool, 2017. 191 pp. ISBN: 1627057080.

[10] Herbert Stachowiak. *Allgemeine Modelltheorie*. German. Wien: Springer, 1973. ISBN: 3-211-81106-0.

[11] Object Management Group. *OMG Meta Object Facility (MOF) Core Specification. Version 2.5.1*. 2019. URL: https://www.omg.org/spec/MOF/2.5.1/PDF (visited on 06/14/2021).

[12] Thomas Stahl et al. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. German. 2. aktualisierte und erweiterte Auflage. EBL-Schweitzer. dpunkt.verlag, 2007. ISBN: 9783898648813. URL: http://swb.eblib.com/patron/FullRecord.aspx?p=1049742.

[13] Dave Steinberg et al. *EMF - Eclipse modeling framework*. Second Edition. The eclipse series. Boston, Mass.: Addison-Wesley, 2009. ISBN: 0321331885. URL: http://bvbr.bib-bvb.de:8991/F?func=service&doc_library=BVB01&doc_number=014933667&line_number=0001&func_code=DB_RECORDS&service_type=MEDIA.

[14] Lars Vogel. *Eclipse Modeling Framework (EMF) - Tutorial*. 2019. URL: https://www.vogella.com/tutorials/EclipseEMF/article.html (visited on 06/09/2021).

[15] Philip Langer et al. "On the Usage of UML: Initial Results of Analyzing Open UML Models". In: *Modellierung 2014. Proceedings: 19-21 March 2014* (Vienna, Austria,). Ed. by Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer. Bonn: Gesellschaft für Informatik e.V, 2014, pp. 289–304.

[16] Jordi Cabot and Martin Gogolla. "Object Constraint Language (OCL): A Definitive Guide". In: *12th International School On Formal Methods For The Design Of Computer, Communication, and Software Systems (SFM 2012). Formal Methods for Model-Driven Engineering. Proceedings: 18-23 June 2012* (Bertinoro, Italy,). Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. LNCS sublibrary. SL 2, Programming and software engineering 7320. Heidelberg: Springer, 2012, pp. 58–90. ISBN: 978-3-642-30981-6. DOI: 10.1007/978-3-642-30982-3_3.

[17] Christian Damus et al. *OCL Documentation*. 2021. URL: https://download.eclipse.org/ocl/doc/6.15.0/ocl.pdf (visited on 01/11/2022).

[18] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. "Translating the Object Constraint Language into First-order Predicate Logic". In: *2nd Verification Workshop (VERIFY'02). Proceedings: 25-26 June 2002* (Copenhagen, Denmark,). Ed. by Serge Autexier and Heiko Mantel. Vol. 2. 2002, pp. 113–123.

[19] Wolfgang Ahrendt et al. *Deductive Software Verification - The KeY Book. From Theory to Practice*. Vol. 10001. LNCS Sublibrary: SL2 - Programming and Software Engineering. Cham: Springer, 2016. ISBN: 978-3-319-49811-9. URL: https://link.springer.com/book/10.1007/978-3-319-49812-6 (visited on 01/11/2022).

[20] Shane Sendall and Wojtek Kozaczynski. "Model Transformation: The Heart and Soul of Model-Driven Software Development". In: *IEEE Software* 20.5 (2003), pp. 42–45. DOI: 10.1109/MS.2003.1231150.

[21] *Xtend - Documentation*. Eclipse Foundation. 2020. URL: https://www.eclipse.org/xtend/documentation/index.html (visited on 11/01/2021).

[22] Pierangela Samarati and Sabrina Capitani de Vimercati. "Access Control: Policies, Models, and Mechanisms". In: *Foundations of Security Analysis and Design. Tutorial Lectures*. Ed. by Riccardo Focardi and Roberto Gorrieri. Lecture Notes in Computer Science 2171. New York: Springer, 2001, pp. 137–196. ISBN: 978-3-540-42896-1. DOI: 10.1007/3-540-45608-2_3.

[23] Khair Eddin Sabri. "Automated Verification Of Role-Based Access Control Policies Constraints Using Prover9". In: *International Journal of Security, Privacy and Trust Management* 4.1 (2015), pp. 01–10. ISSN: 23194103. DOI: 10.5121/ijsptm.2015.4101.

[24] Faouzi Jaidi, Faten Labbene Ayachi, and Adel Bouhoula. "A Comprehensive Formal Solution for Access Control Policies Management. Defect Detection, Analysis and Risk Assessment". In: *8th International Symposium on Symbolic Computation in Software Science (SCSS 2017). Proceedings: 06-09 April 2017* (Gammarth, Tunisia,). Ed. by Mohamed Mosbah and Michael Rusinowitch. EPiC Series in Computing. EasyChair, 2017, pp. 120–132. DOI: 10.29007/q916.

[25] Chung Tong Hu et al. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. Gaithersburg, MD: National Institute of Standards and Technology, 2014. URL: https://www.nist.gov/publications/guide-attribute-based-access-control-abac-definition-and-considerations (visited on 06/28/2021).

[26] Stefan Luber and Peter Schmitz. *Was ist eine Whitelist und Blacklist? Definition Whitelist / Blacklist*. German. 2017. URL: https://www.security-insider.de/was-ist-eine-whitelist-und-blacklist-a-667574/ (visited on 09/15/2021).

[27] Arvind Narayanan et al. *Bitcoin and Cryptocurrency Technologies. A Comprehensive Introduction*. Princeton University Press, 2016.

[28] Daniela Barbosa. *Staff Corner: Introducing Hyperledger Foundation*. Hyperledger Foundation. 2021. URL: https://www.hyperledger.org/blog/2021/10/27/staff-corner-introducing-hyperledger-foundation (visited on 11/24/2021).

[29] Orlenys López–Pintado et al. "Caterpillar: A business process execution engine on the Ethereum blockchain". In: *Software: Practice and Experience* 49.7 (2019), pp. 1162–1193. DOI: 10.1002/spe.2702.

[30] Dominik Harz and William Knottenbelt. "Towards Safer Smart Contracts: A Survey of Languages and Verification Methods". In: *Computing Research Repository (CoRR)* abs/1809.09805 (2018). URL: https://arxiv.org/pdf/1809.09805 (visited on 05/05/2021).

[31] Leonardo Alt and Christian Reitwiessner. "SMT-Based Verification of Solidity Smart Contracts". In: *8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Industrial Practice (ISoLA 2018). Proceedings: 05-09 November 2018* (Limassol, Cyprus,). Ed. by Tiziana Margaria-Steffen and Bernhard Steffen. LNCS sublibrary. SL 1, Theoretical computer science and general issues 11247. Cham, Switzerland: Springer, 2018, pp. 376–388. ISBN: 978-3-030-03426-9. DOI: 10.1007/978-3-030-03427-6_28.

[32] Gavin Zheng et al. *Ethereum Smart Contract Development in Solidity*. Singapore: Springer, 2021. ISBN: 978-981-15-6218-1. DOI: 10.1007/978-981-15-6218-1.

[33] Alexander Mense and Markus Flatscher. "Security Vulnerabilities in Ethereum Smart Contracts". In: *20th International Conference on Information Integration and Web-Based Applications & Services (iiWAS2018). Proceedings: 19-21 November 2018* (Yogyakarta, Indonesia,). Ed. by Maria Indrawan-Santiago et al. New York, NY, USA: ACM, 2018, pp. 375–380. ISBN: 9781450364799. DOI: 10.1145/3282373.3282419.

[34] Ákos Hajdu and Dejan Jovanović. "SMT-Friendly Formalization of the Solidity Memory Model". In: *29th European Symposium on Programming (ESOP 2020). Proceedings: 25-30 April 2020* (Dublin, Ireland,). Ed. by Peter Müller. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 224–250. ISBN: 978-3-030-44913-1. DOI: 10.1007/978-3-030-44914-8_9.

[35] Cypress Data Defense. *Differences Between Static Code Analysis and Dynamic Testing*. 2020. URL: https://cypressdatadefense.com/blog/static-and-dynamic-code-analysis/ (visited on 08/28/2021).

[36] Ákos Hajdu, Dejan Jovanović, and Gabriela Ciocarlie. "Formal Specification and Verification of Solidity Contracts with Events. Short Paper". In: *2nd Workshop on Formal Methods for Blockchains (FMBC 2020). Proceedings: 20-21 July 2020* (Online,). Ed. by Bruno Bernardo and Diego Marmsoler. Vol. 84. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 2:1–2:9. ISBN: 978-3-95977-169-6. DOI: 10.4230/OASIcs.FMBC.2020.2.

[37] Shaokai Lin et al. *SciviK: A Versatile Framework for Specifying and Verifying Smart Contracts*. 2021. URL: https://arxiv.org/pdf/2103.02209 (visited on 08/19/2021).

[38] Jamal Abd-Ali, Karim El Guemhioui, and Luigi Logrippo. "Metamodelling with Formal Semantics with Application to Access Control Specification". In: *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD). Proceedings: 09-11 February 2015* (Angers, Loire Valley, France,). Ed. by Slimane Hammoudi. Piscataway, NJ: IEEE, 2015, pp. 354–362. ISBN: 9789897581366.

[39] Tanveer Mustafa, Michael Drouineaud, and Karsten Sohr. “Towards Formal Specification and Verification of a Role-Based Authorization Engine using JML”. In: *2010 ICSE Workshop on Software Engineering for Secure Systems (SESS'10). Proceedings: 02 May 2010* (Cape Town, South Africa,). Ed. by Seok-Won Lee, Mattia Monga, and Jan Jürjens. New York, New York, USA: ACM Press, 2010, pp. 50–57. ISBN: 9781605589657. DOI: 10.1145/1809100.1809108.

[40] Zoubeyr Farah, Hania Gadouche, and Abdelkamel Tari. “A Correct-by-Design Role-Based Access Control Model for Healthcare Information Systems”. In: *CEEE (International Journal of Control, Energy and Electrical Engineering)* 7 (2019), pp. 12–17.

[41] Konstantine Arkoudas, Ritu Chadha, and Jason Chiang. “Sophisticated Access Control via SMT and Logical Frameworks”. In: *ACM Transactions on Information and System Security* 16.4 (2014), pp. 1–31. ISSN: 1094-9224. DOI: 10.1145/2595222.

[42] Grzegorz Kolaczek. “Specification and Verification of Constraints in Role Based Access Control for Enterprise Security System”. In: *12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2003). Proceedings: 09-11 June 2003* (Linz, Austria,). IEEE Computer Society. IEEE Computer Society, 2003, pp. 190–195. ISBN: 0-7695-1963-6. DOI: 10.1109/ENABL.2003.1231406.

[43] Chunyang Yuan et al. “A Verifiable Formal Specification for RBAC Model with Constraints of Separation of Duty”. In: *2nd SKLOIS International Conference on Information Security and Cryptology (Inscrypt 2006). Proceedings: 29 November - 01 December 2006* (Beijing, China,). Ed. by Helger Lipmaa, Moti Yung, and Dongdai Lin. Lecture notes in computer science, 0302-9743 4318. Berlin and London: Springer, 2006, pp. 196–210. ISBN: 978-3-540-49608-3. DOI: 10.1007/11937807_16.

[44] Samrat Mondal and Shamik Sural. “A Verification Framework for Temporal RBAC with Role Hierarchy (Short Paper)”. In: *4th International Conference on Information Systems Security (ICISS 2008). Proceedings: 16-20 December 2008* (Hyderabad, India,). Ed. by R. Sekar and Arun K. Pujari. Lecture notes in computer science, 0302-9743 5352. Berlin: Springer, 2008, pp. 140–147. ISBN: 978-3-540-89861-0. DOI: 10.1007/978-3-540-89862-7_11.

[45] Anastasia Mavridou and Aron Laszka. “Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach”. In: *22nd International Conference on Financial Cryptography and Data Security (FC 2018). Proceedings: 26 February - 02 March 2018* (Nieuwpoort, Curaçao,). Ed. by Sarah Meiklejohn and Kazue Sako. LNCS sublibrary. SL 4 - Security and cryptology 10957. Berlin: Springer, 2018, pp. 523–540. ISBN: 978-3-662-58386-9. DOI: 10.1007/978-3-662-58387-6_28.

[46] Jan Ladleif and Mathias Weske. “A Unifying Model of Legal Smart Contracts”. In: *38th International Conference on Conceptual modeling (ER 2019). Proceedings: 04-07 November 2019* (Salvador, Brazil,). Ed. by Alberto H. F. Laender et al. LNCS sublibrary. SL 3, Information systems and applications, incl. Internet/Web, and HCI 11788. Cham, Switzerland: Springer, 2019, pp. 323–337. ISBN: 978-3-030-33222-8. DOI: 10.1007/978-3-030-33223-5_27.

[47] Nejc Zupan et al. “Secure Smart Contract Generation Based on Petri Nets”. In: *Blockchain Technology for Industry 4.0. Secure, Decentralized, Distributed and Trusted Industry Environment*. Ed. by Rodrigo Rosa Da Righi, Antonio Marcos Alberti, and Madhusudan Singh. Blockchain Technologies. Singapore: Springer, 2020, pp. 73–98. ISBN: 978-981-15-1136-3. DOI: 10.1007/978-981-15-1137-0_4.

[48] Maximilian Wohrer and Uwe Zdun. “Domain Specific Language for Smart Contract Development”. In: *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). Proceedings: 02-06 May 2020* (Toronto, ON, Canada,). Ed. by Kostas Plataniotis and Andreas Veneris. IEEE, 2020, pp. 1–9. ISBN: 978-1-7281-6680-3. DOI: 10.1109/ICBC48266.2020.9169399.

[49] Henry Syahputra and Hans Weigand. “The Development of Smart Contracts for Heterogeneous Blockchains”. In: *Enterprise interoperability VIII. Smart Services and Business Impact of Enterprise Interoperability*. Ed. by Keith Popplewell et al. Vol. 9. Proceedings of the I-ESA Conferences Volume 9. Cham, Switzerland: Springer, 2019, pp. 229–238. ISBN: 978-3-030-13692-5. DOI: 10.1007/978-3-030-13693-2_19.

[50] Anastasia Mavridou et al. “VeriSolid: Correct-by-Design Smart Contracts for Ethereum”. In: *23rd International Conference on Financial Cryptography and Data Security (FC 2019). Proceedings: 18-22 February 2019* (Frigate Bay, St. Kitts and Nevis,). Ed. by Ian Goldberg and Tyler Moore. Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 446–465. ISBN: 978-3-030-32100-0. DOI: 10.1007/978-3-030-32101-7_27.

[51] Karthikeyan Bhargavan et al. “Formal Verification of Smart Contracts. Short Paper”. In: *2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS 2016). Proceedings: 24 October 2016*. 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS’16) (Vienna, Austria,). Ed. by Toby Murray and Deian Stefan. New York, NY, USA: ACM, 2016, pp. 91–96. ISBN: 9781450345743. DOI: 10.1145/2993600.2993611.

[52] Wolfgang Ahrendt et al. “Verification of Smart Contract Business Logic”. In: *8th International Conference on Fundamentals of Software Engineering (FSEN 2019). Proceedings: 01-03 May 2019* (Tehran, Iran,). Ed. by Hossein Hojjat and Mieke Massink. LNCS sublibrary. SL 2, Programming and software engineering 11761. Cham, Switzerland: Springer, 2019, pp. 228–243. ISBN: 978-3-030-31516-0. DOI: 10.1007/978-3-030-31517-7_16.

[53] Bernhard Beckert, Jonas Schiffl, and Mattias Ulbrich. “Smart Contracts: Application Scenarios for Deductive Program Verification”. In: *1st Workshop on Formal Methods for Blockchains (FMBC 2019) at the 23rd Symposium on Formal Methods (FM 2019). Proceedings: 07-11 October 2019* (Porto, Portugal,). Ed. by Emil Sekerinski et al. Cham: Springer International Publishing, 2019, pp. 293–298. ISBN: 978-3-030-54994-7.

[54] Olivera Marjanovic and Zoran Milosevic. “Towards Formal Modeling of e-Contracts”. In: *5th IEEE International Enterprise Distributed Object Computing Conference (EDOC’01). Proceedings: 04-07 September 2001* (Seattle, WA, USA,). NW Washington, DC, USA: IEEE Computer Society, 2001, pp. 59–68. ISBN: 0-7695-1345-X. DOI: 10.1109/EDOC.2001.950423.

[55] Jonas Schiffl et al. “Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control”. In: *26th ACM Symposium on Access Control Models and Technologies (SACMAT'21). Proceedings: 16-18 June 2021* (Virtual Event, Spain,). Ed. by Jorge Lobo, Roberto Di Pietro, and Omar Chowdhury. New York, NY, USA: ACM, 2021, pp. 125–130. ISBN: 9781450383653. DOI: 10.1145/3450569.3463574.

[56] Solaman Baby, Prasad B. Honnavalli, and S. Rajashree Soman. “Identity & Access Management System Based on Blockchain”. In: *3rd International Conference on Innovative Computing & Communication (ICICC 2020). Proceedings: 21-23 February 2020* (New Delhi, India,). Ed. by Deepak Gupta et al. 2020, pp. 1–6. DOI: 10.2139/ssrn.3599868.

[57] Maryam Davari and Elisa Bertino. “Access Control Model Extensions to Support Data Privacy Protection based on GDPR”. In: *2019 IEEE International Conference on Big Data (IEEE BigData 2019). Proceedings: 09-12 December 2019* (Los Angeles, CA, USA,). Ed. by Roger Barga and Carlo Zaniolo. Piscataway: IEEE, 2019, pp. 4017–4024. ISBN: 978-1-7281-0858-2. DOI: 10.1109/BigData47090.2019.9006455.

[58] Kinoshita Hirotsugu and Morizumi Tetsuya. “Access Control Model for the Inference Attacks with Access Histories”. In: *41st IEEE Annual Computer Software and Applications Conference (COMPSAC). Proceedings: 04-08 July 2017* (Turin, Italy,). Piscataway: IEEE, 2017, pp. 498–503. ISBN: 978-1-5386-0367-3. DOI: 10.1109/COMPSAC.2017.41.

[59] Muhammad Yasar Khan et al. “An extended access control model for permissioned blockchain frameworks”. In: *Wireless Networks* 26.7 (2020), pp. 4943–4954. DOI: 10.1007/s11276-019-01968-x.

[60] OpenZeppelin. *Access Control - OpenZeppelin Docs*. URL: https://docs.openzeppelin.com/contracts/3.x/access-control (visited on 07/05/2021).

[61] Victor Amaral de Sousa, Corentin Burnay, and Monique Snoeck. “B-MERODE: A Model-Driven Engineering and Artifact-Centric Approach to Generate Smart Contracts”. In: *32nd International Conference on Advanced Information Systems Engineering (CAISE'20). Proceedings: 08-12 June 2020* (Grenoble, France,). Ed. by Camille Salinesi and Dominique Rieu. LNCS-Springer-Verlag, 2020, pp. 117–133. DOI: 10.1007/978-3-030-49435-3_8.

[62] Lucie Mercenne, Kei-Leo Brousmiche, and Elyes Ben Hamida. “Blockchain Studio: A Role-Based Business Workflows Management System”. In: *9th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEEE IEMCON 2018). Proceedings: 01-03 November 2018* (Vancouver, BC, Canada,). Ed. by Victor Leung, Son Vuong, and Satyajit Chakrabarti. Piscataway, New Jersey: IEEE, 2018, pp. 1215–1220. ISBN: 978-1-5386-7266-2. DOI: 10.1109/IEMCON.2018.8614879.

[63] Oliver Stengele et al. “Access Control for Binary Integrity Protection using Ethereum”. In: *24th ACM Symposium on Access Control Models and Technologies (SACMAT'19). Proceedings: 03-06 June 2019* (Toronto, ON, Canada,). Ed. by Florian Kerschbaum et al. ACM Digital Library. New York, NY, USA: ACM, 2019, pp. 3–12. ISBN: 9781450367530. DOI: 10.1145/3322431.3325108.

[64] Alex Beregszaszi and Christian Reitwiessner. *Solidity by Example*. 2019. URL: https://docs.soliditylang.org/en/develop/solidity-by-example.html# (visited on 07/22/2021).

[65] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22. ISSN: 01641212. DOI: 10.1016/j.jss.2008.03.066.

[66] Michael Dietrich and Frederik Reiche. *SolidityMetaModel*. 2021. URL: https://github.com/KASTEL-CSSDA/SolidityMetaModel (visited on 09/20/2021).

[67] Sebastian Krach and Stephan Seifermann. *Metamodel-Modeling-Foundations*. 2019. URL: https://github.com/MDSD-Tools/Metamodel-Modeling-Foundations (visited on 08/02/2021).

[68] Phillip Goldberg. *Smart Contract Best Practices Revisited: Block Number vs. Timestamp*. 2018. URL: https://medium.com/@phillipgoldberg/smart-contract-best-practices-revisited-block-number-vs-timestamp-648905104323 (visited on 12/07/2021).

[69] Josselin Feist et al. *Printer documentation. Slither*. 2021. URL: https://github.com/crytic/slither/wiki/Printer-documentation (visited on 12/10/2021).

[70] Josselin Feist. *Python API*. 2019. URL: https://github.com/crytic/slither/wiki/Python-API (visited on 11/28/2021).

[71] Erich Gamma et al. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Reading, Mass. and Wokingham: Addison-Wesley, 1995. ISBN: 0-201-63361-2.

[72] Josselin Feist. *Incorrect data dependency print information. Slither Issue #337*. 2019. URL: https://github.com/crytic/slither/issues/337 (visited on 11/29/2021).

[73] Paul Anderson. "The Use and Limitations of Static-Analysis Tools to Improve Software Quality". In: *CrossTalk: The Journal of Defense Software Engineering* 21.6 (2008), pp. 18–21.

[74] Michael Dietrich and Frederik Reiche. *SolidityCodeGenerator*. 2021. URL: https://github.com/KASTEL-CSSDA/SolidityCodeGenerator (visited on 09/20/2021).

[75] Michael Emmi, Ákos Hajdu, and Dejan Jovanović. *Unable to verify non-decreasing balance. Solc-Verify Issue #58*. 2019. URL: https://github.com/SRI-CSL/solidity/issues/58 (visited on 11/29/2021).

[76] Max Kramer et al. *Ecore2Txt*. 2021. URL: https://github.com/kit-sdq/Ecore2Txt (visited on 12/05/2021).

[77] Max Kramer and Erik Burger. *Generating code with Xtend and Xtext triggered from the Eclipse context menu*. 2021. URL: https://sdqweb.ipd.kit.edu/wiki/Generating_code_with_Xtend_and_Xtext_triggered_from_the_Eclipse_context_menu (visited on 12/05/2021).

[78] Tutorials Point. *Solidity - Style Guide*. 2021. URL: https://www.tutorialspoint.com/solidity/solidity_style_guide.htm (visited on 12/05/2021).

[79] Robert Heinrich et al. “Integrating Run-Time Observations and Design Component Models for Cloud System Analysis”. In: *9th Workshop on Models@run.time. Proceedings: 30 September 2014* (Valencia, Spain,). Ed. by Sebastian Götz, Nelly Bencomo, and Robert France. 2014, pp. 41–46.

[80] Robert Heinrich. “Architectural Run-Time Models for Performance and Privacy Analysis in Dynamic Cloud Applications”. In: *SIGMETRICS Perform. Eval. Rev.* 43.4 (2016), pp. 13–22. ISSN: 0163-5999. DOI: 10.1145/2897356.2897359.

[81] Vic Basili, Gianluigi Caldiera, and H. Dieter Rombach. “The Goal Question Metric Approach”. In: *Encyclopedia of Software Engineering*. Ed. by John J. Marciniak. 2nd Edition. John Wiley & Sons, Inc., 2002, pp. 1–10. ISBN: 0471028959. DOI: 10.1002/0471028959.sof142.

[82] Per Runeson and Martin Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164. DOI: 10.1007/s10664-008-9102-8.

[83] Marcel F. van Amstel and Mark G. J. van den Brand. “Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare”. In: *4th International Conference on Theory and Practice of Model Transformations (ICMT 2011). Proceedings: 27-28 June 2011* (Zurich, Switzerland,). Ed. by Jordi Cabot and Eelco Visser. LNCS sublibrary. SL 2, Programming and software engineering 6707. Heidelberg: Springer, 2011, pp. 108–122. ISBN: 978-3-642-21731-9. DOI: 10.1007/978-3-642-21732-6_8.

[84] Johanna Gustafsson. *Single case studies vs. multiple case studies: A comparative study*. Engineering and Science: Halmstad University, Halmstad, Sweden, 2017.

[85] Jack Peterson et al. *Augur: A Decentralized Oracle and Prediction Market Platform (v2.0)*. Whitepaper. Forecast Foundation, 2021. URL: https://github.com/AugurProject/whitepaper/releases/latest/download/augur-whitepaper-v2.pdf (visited on 05/31/2021).

[86] Laurent Benichou. *fizzy. Innovation at AXA*. 2017. URL: https://laurentbenichou.medium.com/fizzy-innovation-at-axa-2304bb71e0f7 (visited on 11/22/2021).

[87] Miranda Wood. *AXA withdraws blockchain flight delay compensation experiment*. 2019. URL: https://www.ledgerinsights.com/axa-blockchain-flight-delay-compensation/ (visited on 07/24/2021).

[88] Sebastian Friebe et al. “Coupling Smart Contracts: A Comparative Case Study”. In: *3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). Proceedings: 27-30 September 2021* (Paris, France,). Ed. by Andreas Veneris and Abdelkader Lahmadi. IEEE, 2021, pp. 137–144. ISBN: 978-1-6654-3924-4. DOI: 10.1109/BRAINS52497.2021.9569830.

[89] Richard Hobeck et al. “Process Mining on Blockchain Data: A Case Study of Augur”. In: *19th International Conference on Business Process Management (BPM 2021). Proceedings: 06-10 September 2021* (Rome, Italy,). Ed. by Artem Polyvyanyy. LNCS sublibrary, SL 3, Information systems and applications, incl. internet/web, and HCI 12875. Cham, Switzerland: Springer, 2021, pp. 306–323. ISBN: 978-3-030-85468-3. DOI: 10.1007/978-3-030-85469-0_20.

[90] Dejan Jovanović. *Unsupported features. Solc-Verify Issue #1.* 2019. URL: https://github.com/SRI-CSL/solidity/issues/1 (visited on 12/13/2021).

[91] Jan-Philipp Töberg and Frederik Reiche. *JanPhilippToeberg. Modelling and Enforcing Access Control Requirements for Smart Contracts.* 2021. URL: https://git.scc.kit.edu/i43/stud/abschlussarbeiten/masterarbeiten/janphilipptoeberg (visited on 12/12/2021).

A. Appendix

This appendix collects additional information to support the contents of the thesis. This includes the concrete implementations for the mentioned smart contracts as well as the created *Slither* printer. These implementations are written in Python and Solidity and can also be examined in our GitLab repository [91].

Appendix A.1 provides the manual implementation for the running example from Chapter 4. This implementation consists of the *SingleAuction* and the *MarketManagement* contract. We provide the Python implementation for the `influence-and-calls` *Slither* printer described in Section 8.2 in Appendix A.2. The complete results of this printer for the *SingleAuction* contract from Appendix A.1 are provided in Appendix A.3. During the generation, the generator creates an additional access control smart contract according to the explanations from Section 9.2.4. Appendix A.4 provides a complete access control contract for the auction use case from Chapter 4. As described in Section 11.3.5, we manually added violations to the Augur and Fizzy use case on the architectural level. Appendix A.5 summarizes the resulting log files containing the violations. Appendix A.6 categorizes the information flows between variables found during the case study according to the categories presented in Section 10.1.3.

A.1. Implementation of the Auction Use Case

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3
4  contract SingleAuction {
5        address payable private managingContract;
6        address payable private sellerAddress;
7        address private highestBidder;
8
9        bool private auctionClosed;
10       bool private moneyCollected;
11       uint private highestBid;
12       uint private bidderCounter;
13
14       mapping(uint => address payable) private bidders;
15       mapping(address => uint) private currentBids;
16
17       constructor(address seller) {
18             managingContract = payable(msg.sender);
```

```
19            sellerAddress = payable(seller);
20
21            auctionClosed = false;
22            moneyCollected = false;
23            highestBid = 0;
24            bidderCounter = 0;
25      }
26
27      function bid() external payable {
28            require(auctionClosed == false, "Auction already closed.");
29            require(currentBids[msg.sender] > 0, "Please withdraw your bid
    before staking a new one.");
30
31            bidders[bidderCounter] = payable(msg.sender);
32            currentBids[msg.sender] = msg.value;
33            bidderCounter++;
34
35            if(msg.value > highestBid) {
36                  highestBid = msg.value;
37                  highestBidder = msg.sender;
38            }
39      }
40
41      function close() public {
42            require(msg.sender == sellerAddress, "Only the seller can close
    the auction.");
43            require(auctionClosed == false, "Auction already closed.");
44
45            auctionClosed = true;
46      }
47
48      function collectMoney() external {
49            require(msg.sender == sellerAddress, "Only the seller can
    collect the money.");
50            require(auctionClosed == true, "Auction is still active.");
51            require(moneyCollected == false, "The money has already been
    collected.");
52
53            moneyCollected = true;
54            if(!sellerAddress.send(highestBid)) {
55                  moneyCollected = false;
56            }
57      }
58
59      function collectItem() external {
```

```
60            require(msg.sender == highestBidder, "Only the highest bidder
    can collect the item.");
61            require(auctionClosed == true, "Auction is still active.");
62
63            // sent the item, whatever form it may have
64      }
65
66      function withdrawMoney() external {
67            require(msg.sender != highestBidder, "The highest bidder cannot
    withdraw their money.");
68            require(currentBids[msg.sender] > 0, "No money to withdraw.");
69
70            uint amount = currentBids[msg.sender];
71            currentBids[msg.sender] = 0;
72            if(!msg.sender.send(amount)) {
73                  currentBids[msg.sender] = amount;
74                  assert(currentBids[msg.sender] == amount);
75            } else {
76                  assert(currentBids[msg.sender] == 0);
77            }
78      }
79
80      function emergencyShutdown() public {
81            require(msg.sender == managingContract, "Only the manager can
    start the emergency shutdown.");
82            auctionClosed = true;
83
84            for(uint i = 0; i < bidderCounter; i++) {
85                  uint amount = currentBids[bidders[i]];
86                  if(amount > 0) {
87                        currentBids[bidders[i]] = 0;
88                        if(!bidders[i].send(amount)) {
89                              currentBids[bidders[i]] = amount;
90                              assert(currentBids[bidders[i]] == amount);
91                        } else {
92                              assert(currentBids[bidders[i]] == 0);
93                        }
94                  }
95            }
96
97            selfdestruct(managingContract);
98      }
99  }
```

Listing A.1: *SingleAuction* contract from the auction use case in Chapter 4

```
// SPDX-License-Identifier GPL-3.0
pragma solidity =0.7.0 0.9.0;

import .SingleAuction.sol;

contract AuctionManagement {

      SingleAuction[] private auctions;
      uint private auctionCounter = 0;

      function createNewAuction() public {
          auctions[auctionCounter] = new SingleAuction(msg.sender);
          auctionCounter++;
      }
}
```

Listing A.2: *AuctionManagement* contract from the auction use case in Chapter 4

A.2. Complete Implementation of the Custom Slither Printer

```
"""
Module enhancing the data-dependency printer.
Detecting and Printing indirect influence as well as direct and transitive
    influence.
Additionally, it also calculates the transitive closure for function calls.

Indirect Influence Example:
    if(influencer > 5) {
        x = 20;
    }
=> Here, "influencer" indirectly influences "x"
"""

from typing import Union, Set, List
from slither.core.variables.variable import Variable
from slither.core.variables.state_variable import StateVariable
from slither.core.declarations import Contract, Function, Modifier
from slither.core.declarations.function_contract import FunctionContract
from slither.core.cfg.node import Node, NodeType
from slither.printers.abstract_printer import AbstractPrinter
from slither.analyses.data_dependency.data_dependency import get_dependencies
from slither.slithir.variables import TemporaryVariable, ReferenceVariable
from slither.utils.myprettytable import MyPrettyTable
```

```
23
24 # Dictionary mapping each variable name to the list of variables it is
       influenced by.
25 # It is used to keept track of the influence overview created at the
       beginning (over all functions).
26 influence_dict = dict()
27
28 """
29 This function maps each variable to its influencers once at the beginning of
       the printing process.
30 It is given the list of contracts to analyte and iterates over it to look at
       each contract independently.
31 For all variables in each contract that has not yet been considered, it
       finds all direct
32 (using the normal data-dependency printer through get_dependencies) and
       indirect dependencies.
33 They are returned, the transitive closure is calculated and saved in a local
       dictionary for later use.
34 """
35 def return_all_dependencies(cons):
36     temp_influence_dict = dict()
37
38     for c in cons:
39         for v in c.state_variables:
40             if v in temp_influence_dict:
41                 continue
42
43             #print('Var: ', v.name, 'Context: ', c.name, 'Type: ', type(c))
44             temp_influence_dict[v] = set()
45             for inf in get_dependencies(v, c):
46                 if isinstance(inf, StateVariable):
47                     if inf not in temp_influence_dict[v] and inf.name !=
       v.name:
48                         temp_influence_dict[v].add(inf)
49
50             for inf in get_indirect_dependencies(v, c):
51                 if isinstance(inf, StateVariable):
52                     if inf not in temp_influence_dict[v] and inf.name !=
       v.name:
53                         temp_influence_dict[v].add(inf)
54
55     return temp_influence_dict
56
57 """
```

```
58 This function takes all found dependencies, calculates the transitive
       closure and then saves
59 the results in the influence_dict dictionary to map each variable to its
       influencers.
60 To do so, it iterates over all state variables and calculates the transitive
       closure recursively.
61 """
62 def fill_dictionary(d):
63     for var in d:
64         already_considered = set()
65         already_considered.add(var)
66         influence_dict[var] = calculate_transitive_closure_recursively(d,
       var, already_considered)
67 
68 """
69 To calculate the transitive closure for the given variable, this function
       looks at all influencers
70 and their influencers recursively. While doing so, it keeps track of the
       already considered influencers
71 to prevent unnecessary recursive steps.
72 """
73 def calculate_transitive_closure_recursively(d, var, already_considered):
74     temp = set()
75     for inf in d[var]:
76         if inf not in already_considered:
77             temp.add(inf)
78             already_considered.add(inf)
79             temp.update(calculate_transitive_closure_recursively(d, inf,
       already_considered))
80 
81     return temp
82 
83 """
84 This function calculates the transitive closure for the function calls,
       beginning with the given function.
85 In each recursive call, it looks at the internal and external (= high_level)
       calls made by that function and
86 adds all functions that have not yet been considered to the result set. When
       no more functions can be traversed,
87 the function returns the set of functions.
88 """
89 def calculate_transitive_closure_for_function_recursively(func,
       already_considered):
90     temp = set()
91 
```

```
92      if not isinstance(func, FunctionContract):
93          return temp
94
95      for called in func.internal_calls:
96          if called not in already_considered and isinstance(called,
        FunctionContract):
97              temp.add(called)
98              already_considered.add(called)
99
        temp.update(calculate_transitive_closure_for_function_recursively(called,
        already_considered))
100
101     for called in func.high_level_calls:
102         if called not in already_considered and isinstance(called,
        FunctionContract):
103             temp.add(called)
104             already_considered.add(called)
105
        temp.update(calculate_transitive_closure_for_function_recursively(called,
        already_considered))
106
107     return temp
108
109 """
110 Function taken from the original data-dependency printer but adjusted to
        better incorporate
111 the additional functionality for indirect influence. The function is given a
        variable v and
112 either a contract or a function (represented by c). If a contract is given,
        the variables
113 that influence v are returned from the influence_dict dictionary. If a
        function is given,
114 the variables are directly collected and returned. To do so, for both cases,
        the results of
115 the normal data-dependency printer (get_dependencies) and our extension are
        combined.
116 """
117 def _get(v, c):
118     if isinstance(c, Contract):
119         return [inf.name for inf in influence_dict[v]]
120
121     elif isinstance(c, Function):
122         res = set()
123         for inf in get_dependencies(v, c):
124             if not isinstance(inf, (TemporaryVariable, ReferenceVariable)):
```

```
                if inf not in res and inf.name != v.name:
                    res.add(inf.name)

        for inf in get_indirect_dependencies(v, c):
            if not isinstance(inf, (TemporaryVariable, ReferenceVariable)):
                if inf not in res and inf.name != v.name:
                    res.add(inf.name)

        return list(res)

"""
This function returns the variables that indirectly influence the given
    variable var.
To do so, it differentiates between contracts and functions and either
    iterates over
all available functions to collect the variables or directly analyzes the
    given
function.
"""
def get_indirect_dependencies(
    var: Variable,
    cont: Union[Contract, Function]
) -> Set[Variable]:

    assert isinstance(cont, (Contract, Function))
    #print('Var: ', var.name, 'Context: ', cont.name, 'Type: ', type(cont))
    res = set()

    if isinstance(cont, Contract):
        for func in cont.functions:
            res.update(get_influencers_for_var_in_func(var, func))
        return res
    elif isinstance(cont, Function):
        return get_influencers_for_var_in_func(var, cont)

"""
This function takes a variable var and a function func and returns all
    variables that
influence var inside of the function. To do so, it uses the CFG by looking
    at all
conditional nodes in the function and checking if var is written inside its
    children
nodes.
"""
def get_influencers_for_var_in_func(
```

```
164     var: Variable,
165     func: Function
166 ) -> Set[Variable]:
167 
168     res = set()
169     # print('Func: ', func.name)
170     # print('Written: ', *func.variables_written)
171     # print('Read: ', *func.variables_read)
172     if var in func.variables_written:
173         condNodes = [n for n in func.nodes if n.is_conditional()]
174         for node in condNodes:
175             if check_children_for_write(var, node):
176                 res.update(node.variables_read)
177 
178     return res
179 
180 """
181 This function is used to check if the given variable is written by any
        children
182 of the given node. To do so, it checks all child nodes recursively. If any
        check is positive,
183 these results will be returned. If an END_IF node is found during the
        traversal of the CFG,
184 a further look at those children nodes is not necessary.
185 """
186 def check_children_for_write(
187     check_for: Variable,
188     node: Node
189 ) -> bool:
190 
191     for son in node.sons:
192         if check_for in son.variables_written:
193             return True
194         if son.type != NodeType.ENDIF:
195             if check_children_for_write(check_for, son):
196                 return True
197     return False
198 
199 """
200 This class is mainly the same as the printer class for the standard
        data-dependency printer.
201 The only changes are how the data is collected beforehand and the addition
        of a table summarizing
202 the transitive closure for function calls before the results for each
        function are displayed.
```

```
"""
class InfluenceAndCallsPrinter(AbstractPrinter):
    """
    Documentation
    """

    ARGUMENT = "influence-and-calls"
    HELP = "Prints all influencing relations between variables and the
    transitive closure for function calls."

    WIKI = "ToDo"

    def output(self, _filename):
        """
        _filename is not used
        Args:
            _filename(string)
        """

        fill_dictionary(return_all_dependencies(self.contracts))

        all_tables = []
        all_txt = ""

        txt = ""
        for c in self.contracts:
            if c.is_top_level:
                continue
            txt += "\nContract %s\n" % c.name
            table = MyPrettyTable(["Variable", "Influencers"])
            for v in c.state_variables:
                table.add_row([v.name, _get(v, c)])

            txt += str(table)

            txt += "\n"
            table = MyPrettyTable(["Function", "Internal & External Calls"])
            for f in c.functions_and_modifiers_declared:
                already_considered = set()
                already_considered.add(f)
                called_funcs =
    calculate_transitive_closure_for_function_recursively(f,
    already_considered)
                temp = set()
                for f2 in called_funcs:
```

```
                    if not isinstance(f2, tuple):
                        temp.add(f2.full_name)
                table.add_row([f.full_name, [cf for cf in temp]])

            txt += str(table)


            for f in c.functions_and_modifiers_declared:
                txt += "\nFunction %s\n" % f.full_name
                table = MyPrettyTable(["Variable", "Influencers"])
                for v in c.state_variables:
                    table.add_row([v.canonical_name, _get(v, f)])
                txt += str(table)

            self.info(txt)

            all_txt += txt
            all_tables.append((c.name, table))

        res = self.generate_output(all_txt)
        for name, table in all_tables:
            res.add_pretty_table(table, name)

        return res
```

Listing A.3: Complete implementation of the `influence-and-calls` printer from Section 8.2

A.3. Slither Printer Results for the SingleAuction Contract

Table A.1.: Results regarding variable influence and function calls for the *SingleAuction* contract from Listing A.1 using the `influence-and-calls` printer.

Variable	Influencers
managingContract	[]
sellerAddress	[]
highestBidder	[currentBids, highestBid, sellerAddress, auctionClosed, bidders, bidderCounter, managingContract]
auctionClosed	[sellerAddress, managingContract]
moneyCollected	[currentBids, highestBid, sellerAddress, auctionClosed, bidders, bidderCounter, managingContract, highestBidder]
highestBid	[currentBids, highestBidder, sellerAddress, auctionClosed, bidders, bidderCounter, managingContract]
bidderCounter	[currentBids, highestBidder, sellerAddress, auctionClosed, bidders, highestBid, managingContract]
bidders	[currentBids, highestBidder, sellerAddress, auctionClosed, bidderCounter, highestBid, managingContract]
currentBids	[bidders, highestBidder, sellerAddress, auctionClosed, bidderCounter, highestBid, managingContract]
bid()	[require(bool,string)]
close()	[require(bool,string)]
collectMoney()	[require(bool,string)]
collectItem()	[require(bool,string)]
withdrawMoney()	[assert(bool), require(bool,string)]
emergencyShutdown()	[assert(bool), require(bool,string), selfdestruct(address)]

A.4. Generated Access Control Contract for the Auction Use Case

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract AccessControlContract {
	enum Roles { SELLER, BIDDER, HIGHEST_BIDDER, MANAGER, ADMIN }

	mapping(address => mapping(Roles => bool)) private roleAssignment;
	uint256 private timeAtStart = 0;
	uint private sellerCounter = 0;
	uint private highestBidderCounter = 0;

	constructor(address admin) {
		roleAssignment[admin][Roles.ADMIN] = true;
		timeAtStart = block.timestamp;
	}

	function checkAccess(address entity, Roles role) public view
returns(bool result) {
		return roleAssignment[entity][role];
	}

	function checkTiming(bool upper, uint256 addition) public view
returns(bool result) {
		if(upper) {
			return block.timestamp >= timeAtStart + addition;
		} else {
			return block.timestamp <= timeAtStart + addition;
		}
	}

	/// @notice modifies roleAssignment[entity][Roles.SELLER]
	/// @notice modifies sellerCounter
	function changeSellerRoleForEntity(address entity, bool giveRole)
external onlyAdmin {
		if(giveRole) {
			require(!checkAccess(msg.sender, Roles.BIDDER), "The
address cannot be a Bidder as well.");
			require(!checkAccess(msg.sender, Roles.MANAGER), "The
address cannot be a Manager as well.");
			require(sellerCounter < 1, "There are already 1
Seller(s).");
```

```
36                sellerCounter++;
37          } else {
38                require(sellerCounter > 0, "There needs to be at least one
    Seller before any role can be removed.");
39                sellerCounter--;
40          }
41
42          roleAssignment[entity][Roles.SELLER] = giveRole;
43      }
44
45      /// @notice modifies roleAssignment[entity][Roles.BIDDER]
46      function changeBidderRoleForEntity(address entity, bool giveRole)
    external onlyAdmin {
47          if(giveRole) {
48                require(!checkAccess(msg.sender, Roles.SELLER), "The
    address cannot be a Seller as well.");
49          }
50
51          roleAssignment[entity][Roles.BIDDER] = giveRole;
52      }
53
54      /// @notice modifies roleAssignment[entity][Roles.HIGHEST_BIDDER]
55      /// @notice modifies highestBidderCounter
56      function changeHighestBidderRoleForEntity(address entity, bool
    giveRole) external onlyAdmin {
57          if(giveRole) {
58                require(checkAccess(msg.sender, Roles.BIDDER), "The
    address needs to be a Bidder as well.");
59                require(highestBidderCounter < 1, "There are already 1
    Highest Bidder(s).");
60                highestBidderCounter++;
61          } else {
62                require(highestBidderCounter > 0, "There needs to be at
    least one Highest Bidder before any role can be removed.");
63                highestBidderCounter--;
64          }
65
66          roleAssignment[entity][Roles.HIGHEST_BIDDER] = giveRole;
67      }
68
69      /// @notice modifies roleAssignment[entity][Roles.MANAGER]
70      function changeManagerRoleForEntity(address entity, bool giveRole)
    external onlyAdmin {
71          if(giveRole) {
```

```
72                  require(!checkAccess(msg.sender, Roles.SELLER), "The
    address cannot be a Seller as well.");
73              }
74
75              roleAssignment[entity][Roles.MANAGER] = giveRole;
76          }
77          /// @notice modifies roleAssignment[entity][Roles.ADMIN]
78          function changeAdminRoleForEntity(address entity, bool giveRole)
    external onlyAdmin {
79              roleAssignment[entity][Roles.ADMIN] = giveRole;
80          }
81
82          modifier onlyAdmin {
83              require(checkAccess(msg.sender, Roles.ADMIN), "Access denied due
    to missing role!");
84              _;
85          }
86  }
```

Listing A.4: Generated access control contract for the auction use case from Chapter 4

A.5. Results of the Soundness Check for the Malicious Models

```
1  There are violations in the selected AccessControlSystem 'Augur' and
       SmartContracts 'Market' 'MarketManagement':
2  1) Role 'Designated Reporter' can access function 'Report' indirectly
       through function 'Designated Report'
3  2) Role 'Shareholder' cannot modify variable 'Agree Counter' through any
       function
4  3) Role 'Market Creator' can illegally influence variable 'Designated
       Reporter' by modifying variable 'Reporter Set'
5  4) Role 'Market Creator' can illegally influence variable 'Designated
       Reporter' through function 'Set Reporter'
6  5) Role 'Shareholder' can illegally modify variable 'Market Counter' through
       function 'Dispute Outcome'
7  6) FunctionToStateVariableRelation 'Agree with the reported outcome' -
       Violation for OCL constraint
       'VariableTypeIsNoMappingWithAddressAsKeytype':
8  The referenced variable is not allowed to be mapping with 'address' as its
       key data type
9  7) FunctionToCsmRelation 'Outcome disputes are saved in CSM' - Violation for
       OCL constraint 'VariableTypeNeedsToBeMapping':
10 The referenced variable needs to be a mapping
```

```
8) FunctionToCsmRelation 'Outcome disputes are saved in CSM' - Violation for
    OCL constraint 'MappingKeyTypeNeedsToBeAddressOrAddressPayable':
The referenced mappings key needs to have the data type 'address' or
    'address payable'
```

Listing A.5: Violations.log file for the malicious Augur use case from Section 11.3.1

```
There are violations in the selected AccessControlSystem 'Fizzy' and
    SmartContracts 'Insurance' 'InsuranceManagement':
1) Role 'Insurance Company' - Violation for OCL constraint
    'CardinalityIsValid':
The role cardinality needs to be -1 or bigger than 0
2) Role 'Insurance Company' - Violation for OCL constraint
    'NoRoleCanBePrerequisiteForItself':
The prerequisites need to be different from the role itself
3) Role 'Insurance Company' - Violation for OCL constraint
    'NoRoleCanBeSuperiorToItself':
The superior roles need to be different from the role itself
4) Role 'Insurance Company' - Violation for OCL constraint
    'NoRoleInPrerequisiteAndSuperiorSet':
One role cannot be a prerequisite and a superior role
5) MutualRoleExclusion 'Exclusion of Insurant and Company' - Violation for
    OCL constraint 'RolesCannotBeInHierarchyOrPrerequisite':
The two mutually exclusive roles cannot be superiors or prerequisites to
    each other
6) FunctionToStateVariableRelation 'Change the insured account' - Violation
    for OCL constraint 'VariableTypeNeedsToBeBoolean':
The referenced variable needs to be a boolean variable
7) FunctionToStateVariableRelation 'Checking for a delay indirectly closes
    the insurance' - Violation for OCL constraint
    'ForbidTimeContextConditions':
FunctionToVariable relations are not allowed to have time-based constraints
8) Violation for OCL constraint 'NoMoreBalanceModificationsThanFunctions':
There are more function balance modifications than functions
9) Violation for OCL constraint
    'BalanceModificationsReferenceDifferentFunctions':
All balance modifications need to reference different functions
10) Violation for OCL constraint
    'BalanceModificationsRegardingThisContractNeedPayableFunction':
A function that is allowed to increase the balance of this contract needs to
    be marked as 'payable'
11) Violation for OCL constraint 'NoFunctionOverrides':
The generator does not support the usage of function overrides
12) Violation for OCL constraint 'NoAdditionalLocalFunctions':
Local functions defined not as 'Function' are not covered by the generator
```

Listing A.6: Violations.log file for the malicious Fizzy use case from Section 11.3.2

A.6. Analyzing the Tool Results during the Case Study

In the following, all found influence relations during the three use cases in the case study from Section 11.3 are enumerated. Each influence is analyzed according to the guidelines from Section 10.1.3 and the results are collected in Table 11.6. If an information flow is introduced through the calculation of the transitive closure, we briefly analyze whether the information flow occurs in the implementation. The reasoning for this analysis is provided in Section 10.1.3.

For each use case, a summary of the enumeration is provided in Tables A.2 to A.4. Here, the *Influencer* and the *Influenced* variable are stated. Additionally, the *Function* where the information flows is located in the source code is described. If no *Function* is stated, the information flow is introduced by the transitive closure. Lastly, the *Classification* is summarized. For this purpose, each information flow is numbered to show how many unique information flows occur in the contract.

A.6.1. Market contract of the Augur use case

1. `openReportAllowed` influences `openReporter` in the `claimReporterRole` function:
   The variable `openReportAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `openReporter` variable is modified by the *shareholder*.
   ⇒ Communicate that the *designated reporter*, *open reporter* and *manager* could influence `openReporter` illegally

2. `designatedReporterReported` influences `openReporter` through transitive closure:
   This information flow occurs in the implementation. The variable `designatedReporterReported` is modified by the *designated reporter* whereas the `openReporter` variable is modified by the *shareholder*.
   ⇒ Communicate that the *designated reporter* could influence `openReporter` illegally

3. `designatedReportAllowed` influences `openReporter` through transitive closure:
   This information flow occurs in the implementation. The variable `designatedReportAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `openReporter` variable is modified by the *shareholder*.
   ⇒ Communicate that the *designated reporter*, *open reporter* and *manager* could influence `openReporter` illegally

4. `tradingActive` influences `shares` in the `buyShares` and `sellShares` function:
   The variable `tradingActive` is modified by the *manager* and influenced by the *market creator* whereas the `shares` variable is modified by the *shareholder* and influenced by the *market creator*.
   ⇒ Communicate that the *manager* could influence `shares` illegally

Table A.2.: This table summarizes the classification of the 31 information flows found by the `influence-and-calls` printer for the *Market* contract from the Augur use case from Section 11.3.1.
From these 31 information flows that are not modelled with the ACM, one cannot occur, six occur due to a missing VariableToVariableRelation element in the ACM instances and 24 introduce insecure information flows. From these 24 violations, 14 are unique.

#	Influencer	Influenced	Function	Classification
1	`openReportAllowed`	`openReporter`	`claimReporterRole`	Insecure inf. flow #1
2	`designatedReporter-Reported`	`openReporter`		Insecure inf. flow #2
3	`designatedReport-Allowed`	`openReporter`		Insecure inf. flow #1
4	`tradingActive`	`shares`	`buyShares` & `sellShares`	Insecure inf. flow #3
5	`reporterSet`	`shares`		Missing element
6	`createdBonds`	`shares`		Missing element
7	`disputesAllowed`	`disagreeCounter`	`disputeOutcome`	Insecure inf. flow #4
8	`disputesAllowed`	`agreeCounter`	`disputeOutcome`	Insecure inf. flow #5
9	`disagreeCounter`	`reportedOutcome`	`closeDisputing-Window`	Insecure inf. flow #6
10	`agreeCounter`	`reportedOutcome`	`closeDisputing-Window`	Insecure inf. flow #6
11	`disputesAllowed`	`reportedOutcome`		Insecure inf. flow #7
12	`disputesAllowed`	`disputes`	`disputeOutcome`	Insecure inf. flow #8
13	`designatedReporter-Reported`	`designatedReport-Allowed`	`allowOpenReport`	Missing element
14	`designatedReport-Allowed`	`designatedReporter-Reported`	`designatedReport`	Insecure inf. flow #9
15	`designatedReporter-Reported`	`openReportAllowed`	`allowOpenReport`	Missing element
16	`designatedReport-Allowed`	`openReportAllowed`		Missing element
17	`disagreeCounter`	`settlementPhaseActive`	`closeDisputing-Window`	Insecure inf. flow #10
18	`agreeCounter`	`settlementPhaseActive`	`closeDisputing-Window`	Insecure inf. flow #10
19	`disputesAllowed`	`settlementPhaseActive`		Insecure inf. flow #11
20	`designatedReporter`	`creationBondPaidOut`	`sentCreationBond`	Insecure inf. flow #12
21	`creationBond`	`creationBondPaidOut`	`sentCreationBond`	Insecure inf. flow #12
22	`settlementPhaseActive`	`creationBondPaidOut`	`sentCreationBond`	Missing element
23	`openReporter`	`creationBondPaidOut`	`sentCreationBond`	Insecure inf. flow #13
24	`designatedReporter-Reported`	`creationBondPaidOut`	`sentCreationBond`	Nothing
25	`disputesAllowed`	`creationBondPaidOut`		Insecure inf. flow
26	`createdBonds`	`creationBondPaidOut`		Insecure inf. flow #12
27	`reporterSet`	`creationBondPaidOut`		Insecure inf. flow #12
28	`disagreeCounter`	`creationBondPaidOut`		Insecure inf. flow #13
29	`agreeCounter`	`creationBondPaidOut`		Insecure inf. flow #13
30	`designatedReport-Allowed`	`creationBondPaidOut`		Insecure inf. flow #14
31	`openReportAllowed`	`creationBondPaidOut`		Insecure inf. flow #14

5. `reporterSet` influences `shares` through transitive closure:
   This information flow occurs in the implementation. The variable `reporterSet` is modified by the *market creator*, who is allowed to influence the `shares` variable.
   ⇒ Missing VariableToVariableRelation element

6. `createdBonds` influences `shares` through transitive closure:
   This information flow occurs in the implementation. The variable `createdBonds` is modified by the *market creator*, who is allowed to influence the `shares` variable.
   ⇒ Missing VariableToVariableRelation element

7. `disputesAllowed` influences `disagreeCounter` in the `disputeOutcome` function:
   The variable `disputesAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `disagreeCounter` variable is modified by the *shareholder*.
   ⇒ Communicate that the *designated reporter*, *open reporter* and *manager* could influence `disagreeCounter` illegally

8. `disputesAllowed` influences `agreeCounter` in the `disputeOutcome` function:
   The variable `disputesAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `agreeCounter` variable is modified by the *shareholder*.
   ⇒ Communicate that the *designated reporter*, *open reporter* and *manager* could influence `agreeCounter` illegally

9. `disagreeCounter` influences `reportedOutcome` in the `closeDisputingWindow` function:
   The variable `disagreeCounter` is modified by the *shareholder* whereas the `reportedOutcome` variable is modified by the *designated reporter* and the *open reporter*.
   ⇒ Communicate that the *shareholder* could influence `reportedOutcome` illegally

10. `agreeCounter` influences `reportedOutcome` in the `closeDisputingWindow` function:
    The variable `agreeCounter` is modified by the *shareholder* whereas the `reportedOutcome` variable is modified by the *designated reporter* and the *open reporter*.
    ⇒ Communicate that the *shareholder* could influence `reportedOutcome` illegally

11. `disputesAllowed` influences `reportedOutcome` through transitive closure:
    This information flow occurs in the implementation. The variable `disputesAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `reportedOutcome` variable is modified by the *designated reporter* and the *open reporter*.
    ⇒ Communicate that the *manager* could influence `reportedOutcome` illegally

12. `disputesAllowed` influences `disputes` in the `disputeOutcome` function:
    The variable `disputesAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `disputes` variable is modified by the *shareholder*.
    ⇒ Communicate that the *designated reporter*, *open reporter* and *manager* could influence `disputes` illegally

13. `designatedReporterReported` influences `designatedReportAllowed` in the `allowOpenReport` function:
    The variable `designatedReporterReported` is modified by the *designated reporter*, who is allowed to influence the `designatedReportAllowed` variable.
    ⇒ Missing VariableToVariableRelation element

14. `designatedReportAllowed` influences `designatedReporterReported` in the `designatedReport` function:
    The variable `designatedReportAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `designatedReporterReported` variable is modified by the *designated reporter*.
    ⇒ Communicate that the *open reporter* and *manager* could influence `designatedReporterReported` illegally

15. `designatedReporterReported` influences `openReportAllowed` in the `allowOpenReport` function:
    The variable `designatedReporterReported` is modified by the *designated reporter*, who is allowed to influence the `openReportAllowed` variable.
    ⇒ Missing VariableToVariableRelation element

16. `designatedReportAllowed` influences `openReportAllowed` through transitive closure:
    This information flow occurs in the implementation. The variable `designatedReportAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager*, who are also allowed to influence the `openReportAllowed` variable.
    ⇒ Missing VariableToVariableRelation element

17. `disagreeCounter` influences `settlementPhaseActive` in the `closeDisputingWindow` function:
    The variable `disagreeCounter` is modified by the *shareholder* whereas the `settlementPhaseActive` variable is modified by the *manager*.
    ⇒ Communicate that the *shareholder* could influence `settlementPhaseActive` illegally

18. `agreeCounter` influences `settlementPhaseActive` in the `closeDisputingWindow` function:
    The variable `agreeCounter` is modified by the *shareholder* whereas the `settlementPhaseActive` variable is modified by the *manager*.
    ⇒ Communicate that the *shareholder* could influence `settlementPhaseActive` illegally

19. `disputesAllowed` influences `settlementPhaseActive` through transitive closure:
    This information flow occurs in the implementation. The variable `disputesAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `settlementPhaseActive` variable is modified by the *manager*.
    ⇒ Communicate that the *designated reporter* and the *open reporter* could influence `settlementPhaseActive` illegally

20. `designatedReporter` influences `creationBondPaidOut` in the `sentCreationBond` function:
    The variable `designatedReporter` is modified by the *market creator* whereas the `creationBondPaidOut` variable is modified by the *manager*.
    ⇒ Communicate that the *market creator* could influence `creationBondPaidOut` illegally

21. `creationBond` influences `creationBondPaidOut` in the `sentCreationBond` function:
    The variable `creationBond` is modified by the *market creator* whereas the `creationBondPaidOut` variable is modified by the *manager*.
    ⇒ Communicate that the *market creator* could influence `creationBondPaidOut` illegally

22. `settlementPhaseActive` influences `creationBondPaidOut` in the `sentCreationBond` function:
    The variable `settlementPhaseActive` is modified by the *manager* who is also allowed to influence the `creationBondPaidOut` variable.
    ⇒ Missing VariableToVariableRelation element

23. `openReporter` influences `creationBondPaidOut` in the `sentCreationBond` function:
    The variable `openReporter` is modified by the *shareholder* whereas the `creationBondPaidOut` variable is modified by the *manager*.
    ⇒ Communicate that the *shareholder* could influence `creationBondPaidOut` illegally

24. `designatedReporterReported` influences `creationBondPaidOut` in the `sentCreationBond` function:
    This influence does not occur since *creationBondPaidOut* is changed independent of *designatedReporterReported*'s value.
    ⇒ Nothing

25. `disputesAllowed` influences `creationBondPaidOut` through transitive influence:
    This information flow occurs in the implementation. The variable `disputesAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `creationBondPaidOut` variable is modified by the *manager*.
    ⇒ Communicate that the *designated reporter* and the *open reporter* could influence `creationBondPaidOut` illegally

26. `createdBonds` influences `creationBondPaidOut` through transitive influence:
    This information flow occurs in the implementation. The variable `disputesAllowed` is modified by the *market creator* whereas the `creationBondPaidOut` variable is modified by the *manager*.
    ⇒ Communicate that the *market creator* could influence `creationBondPaidOut` illegally

27. `reporterSet` influences `creationBondPaidOut` through transitive influence:
    This information flow occurs in the implementation. The variable `reporterSet` is

modified by the *market creator* whereas the `creationBondPaidOut` variable is modified by the *manager*.
⇒ Communicate that the *market creator* could influence `creationBondPaidOut` illegally

28. `disagreeCounter` influences `creationBondPaidOut` through transitive influence:
This information flow occurs in the implementation. The variable `disagreeCounter` is modified by the *shareholder* whereas the `creationBondPaidOut` variable is modified by the *manager*.
⇒ Communicate that the *shareholder* could influence `creationBondPaidOut` illegally

29. `agreeCounter` influences `creationBondPaidOut` through transitive influence:
This information flow occurs in the implementation. The variable `agreeCounter` is modified by the *shareholder* whereas the `creationBondPaidOut` variable is modified by the *manager*.
⇒ Communicate that the *shareholder* could influence `creationBondPaidOut` illegally

30. `designatedReportAllowed` influences `creationBondPaidOut` through transitive influence:
This information flow occurs in the implementation. The variable `designatedReportAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `creationBondPaidOut` variable is modified by the *manager*.
⇒ Communicate that the *designated reporter* and the *open reporter* could influence `creationBondPaidOut` illegally

31. `openReportAllowed` influences `creationBondPaidOut` through transitive influence:
This information flow occurs in the implementation. The variable `openReportAllowed` is modified by the *designated reporter*, the *open reporter* and the *manager* whereas the `creationBondPaidOut` variable is modified by the *manager*.
⇒ Communicate that the *designated reporter* and the *open reporter* could influence `creationBondPaidOut` illegally

A.6.2. Insurance contract of the Fizzy use case

1. `insuranceClosed` influences `insurant` in the `changeAccount` function:
The variable `insuranceClosed` is modified by the *insurance company* whereas the `insurant` variable is modified by the *insurant*.
⇒ Communicate that the *insurance company* could influence `insurant` illegally

2. `insuranceAmount` influences `insurant` through transitive closure:
This information flow occurs in the implementation. The variable `insuranceAmount` is not modified by any roles whereas the `insurant` variable is modified by the *insurant*.
⇒ Missing VariableToVariableRelation element

Table A.3.: This table summarizes the classification of the four information flows found by the `influence-and-calls` printer for the *Insurance* contract from the Fizzy use case from Section 11.3.2.
From these four information flows that are not modelled with the ACM, three occur due to a missing VariableToVariableRelation element in the ACM instances and one introduce an insecure information flow.

#	Influencer	Influenced	Function	Classification
1	`insuranceClosed`	`insurant`	`changeAccount`	Insecure information flow
2	`insuranceAmount`	`insurant`		Missing element
3	`insuranceAmount`	`insuranceClosed`	`payout`	Missing element
4	`insurant`	`insuranceClosed`	`payout`	Missing element

3. `insuranceAmount` influences `insuranceClosed` in the `payout` function:
   The variable `insuranceAmount` is not modified by any roles whereas the `insuranceClosed` variable is modified by the *insurant* and the *insurance company*.
   ⇒ Missing VariableToVariableRelation element

4. `insurant` influences `insuranceClosed` in the `payout` function:
   The variable `insurant` is modified by the *insurant* who is, together with the *insurance company*, allowed to influence the `insuranceClosed` variable.
   ⇒ Missing VariableToVariableRelation element

A.6.3. Software contract of the Palinodia use case

1. `accCtrl` influences `rootOwnerAddress` in the `changeRootOwner` function:
   The variable `accCtrl` is not part of the model but created during the generation, so it does not lead to anything illegally.
   ⇒ Nothing

2. `storageContracts` influences `platformsStore` in the `registerBinaryHashStorageContract` function:
   The variable `storageContracts` is modified by the *developer* who is, together with the *platform* and the *maintainer*, allowed to influence the `platformsStore` variable.
   ⇒ Missing VariableToVariableRelation element

3. `platformsStore` influences `storageContracts` in the `registerBinaryHashStorageContract` function:
   The variable `platformsStore` is modified by the *developer*, the *platform* and the *maintainer* whereas the `platformsStore` variable is modified by the *developer*.
   ⇒ Communicate that the *platform* and the *maintainer* could influence `storageContracts` illegally

Table A.4.: This table summarizes the classification of the 16 information flows found by the `influence-and-calls` printer for the Palinodia use case from Section 11.3.3. From these 16 information flows that are not modelled with the ACM, five cannot occur, six occur due to a missing VariableToVariableRelation element in the ACM instances and five introduce insecure information flows. From these five violations, three are unique.

Contract	#	Influencer	Influenced	Function	Classification
Software	1	`accCtrl`	`rootOwnerAddress`	`changeRootOwner`	Nothing
	2	`storageContracts`	`platformsStore`	`registerBinary-HashStorageContract`	Missing element
	3	`platformsStore`	`storageContracts`	`registerBinary-HashStorageContract`	Insecure information flow #1
Identity-Management-	1	`accCtrl`	`identitiesArray`	`changeRootOwner`	Nothing
	2	`accCtrl`	`identitiesMap`	`changeRootOwner`	Nothing
	3	`rootOwnerAddress`	`identitiesArray`	`resetIdentitySet`	Missing element
	4	`rootOwnerAddress`	`identitiesMap`		Nothing
	5	`identitiesArray`	`identitiesMap`	`addIdentity`	Missing element
Binary-Hash-Storage	1	`accCtrl`	`rootOwnerAddress`	`changeRootOwner`	Nothing
	2	`softwareContract`	`initializeStatus`	`setSoftwareContract`	Missing element
	3	`initializeStatus`	`publishCounter`	`publishHash`	Insecure information flow #2
	4	`softwareContract`	`publishCounter`		Insecure information flow #2
	5	`hashStore`	`publishCounter`	`publishHash`	Missing element
	6	`publishCounter`	`hashStore`	`publishHash`	Missing element
	7	`initializeStatus`	`hashStore`	`publishHash`	Insecure information flow #3
	8	`softwareContract`	`hashStore`		Insecure information flow #3

A.6.4. IdentityManagement contract of the Palinodia use case

1. `accCtrl` influences `identitiesArray` in the `changeRootOwner` function:
   The variable `accCtrl` is not part of the model but created during the generation, so it does not lead to anything illegally.
   ⇒ Nothing

2. `accCtrl` influences `identitiesMap` in the `changeRootOwner` function:
   The variable `accCtrl` is not part of the model but created during the generation, so it does not lead to anything illegally.
   ⇒ Nothing

3. `rootOwnerAddress` influences `identitiesArray` in the `resetIdentitySet` function:
   The variable `rootOwnerAddress` is modified by the *root owner* who is, together with the *developer* and the *maintainer*, allowed to influence the `identitiesArray` variable.
   ⇒ Missing VariableToVariableRelation element

4. `rootOwnerAddress` influences `identitiesMap` through transitive closure:
   Due to the structure of the contract, this case does not occur since the `rootOwnerAddress` is explicitly removed from the set of addresses that is added from the `identitiesArray` and thus from the `identitiesMap`.
   ⇒ Nothing

5. `identitiesArray` influences `identitiesMap` in the `addIdentity` function:
   The variable `identitiesArray` is modified by the same roles as the `identitiesArray` variable.
   ⇒ Missing VariableToVariableRelation element

A.6.5. BinaryHashStorage contract of the Palinodia use case

1. `accCtrl` influences `rootOwnerAddress` in the `changeRootOwner` function:
   The variable `accCtrl` is not part of the model but created during the generation, so it does not lead to anything illegally.
   ⇒ Nothing

2. `softwareContract` influences `initializeStatus` in the `setSoftwareContract` function:
   The variable `softwareContract` is modified by the same roles as the `initializeStatus` variable.
   ⇒ Missing VariableToVariableRelation element

3. `initializeStatus` influences `publishCounter` in the `publishHash` function:
   The variable `initializeStatus` is modified by the *developer*, the *maintainer* and the *root owner* whereas the `publishCounter` variable is modified by the *maintainer*.
   ⇒ Communicate that the *developer* and the *root owner* could influence `publishCounter` illegally

4. `softwareContract` influences `publishCounter` through transitive closure:
   This information flow occurs in the implementation. The variable `softwareContract` is modified by the *developer*, the *maintainer* and the *root owner* whereas the `publishCounter` variable is modified by the *maintainer*.
   ⇒ Communicate that the *developer* and the *root owner* could influence `publishCounter` illegally

5. `hashStore` influences `publishCounter` in the `publishHash` function:
   The variable `hashStore` is modified by the *maintainer* who is also allowed to influence the `publishCounter` variable.
   ⇒ Missing VariableToVariableRelation element

6. `publishCounter` influences `hashStore` in the `publishHash` function:
   The variable `publishCounter` is modified by the *maintainer* who is also allowed to influence the `hashStore` variable.
   ⇒ Missing VariableToVariableRelation element

7. `initializeStatus` influences `hashStore` in the `publishHash` function:
   The variable `initializeStatus` is modified by the *developer*, the *maintainer* and the *root owner* whereas the `hashStore` variable is modified by the *maintainer*.
   ⇒ Communicate that the *developer* and the *root owner* could influence `hashStore` illegally

8. `softwareContract` influences `hashStore` through transitive closure:
   This information flow occurs in the implementation. The variable `softwareContract` is modified by the *developer*, the *maintainer* and the *root owner* whereas the `hashStore` variable is modified by the *maintainer*.
   ⇒ Communicate that the *developer* and the *root owner* could influence `hashStore` illegally