JOT
JOURNAL OF
OBJECT TECHNOLOGY

Graph Rewriting for Testing Domain-specific Models: Dynamic Role-Based Access Control

Issam Al-Azzoni[*], **Reiko Heckel**[†], **and Zobia Erum**[‡]
[*]Al Ain University, United Arab Emirates
[†]University of Leicester, United Kingdom
[‡]Birmingham City University, United Kingdom

ABSTRACT Domain-specific languages (DSLs) express requirements or designs through visual abstractions. To support complex development tasks such as code generation, testing and analysis, DSLs need semantic foundations. This paper introduces such a semantic framework for DSLs based on graph rewriting.

We apply our framework to a DSL for defining multi-party dynamic role-based access control policies for smart contracts. Role-based access control models (RBACMs) express constraints on who can access which resources. Dynamic RBACMs allow a dynamic role membership. Access control policies, in particular for smart contracts, can involve multiple parties such as members of different groups or organisations, combining complex logical and dynamic constraints, and hence are hard to design, understand, validate and test at code level.

Our diagrammatic notation supports complex authorisation patterns, including alternatives and multiplicities, to address nuanced access control requirements. Defining the operational semantics for RBACMs by graph rewriting, we let the Groove model checker produce traces for actions where access is granted or denied and generate tests for smart contracts in the Digital Asset Modelling Language (DAML). We validate dynamic access control scenarios generated by ChatGPT for use as test cases or advising users at runtime. Such scenarios represent business workflows interleaved with operations to add or remove role members. They are expressed as Groove control programs and are also verified by its model checker.

KEYWORDS Smart Contracts, DAML, Multi-party Role-based Access Control, Domain-specific Languages, Graph Rewriting, Groove, Model-based Testing, Scenario Validation.

1. Introduction

Smart contracts support workflows of multiple parties without the need for a central authority (Tapscott & Tapscott 2016). They are used across blockchain platforms for applications including commercial and financial transactions, legal processes, data sharing, and supply chain management (Mougayar 2016). To secure applications, smart contracts implement platform-specific access control mechanisms, including sophisticated concepts such as role-based access control (RBAC) and multi-party authorisation, which are difficult to understand and verify against security requirements at the code level, leaving applications open to data leaks and unauthorised access (Vivar et al. 2020). In this paper, we propose a model-based approach to testing and validating access control policies (ACPs) while presenting a recipe for implementing behavioural DSLs based on graph rewriting as a semantics engine.

1.1. Model-based Access Control for Smart Contracts

Consider a transaction for homework grading authorised by an individually assigned grader together with either a second named grader or two members of a role of evaluators. The first participant may be a member of that role, too, but for the

JOT reference format:
Issam Al-Azzoni, Reiko Heckel, and Zobia Erum. *Graph Rewriting for Testing Domain-specific Models: Dynamic Role-Based Access Control*. Journal of Object Technology. Vol. 24, No. 2, 2025.
http://dx.doi.org/10.5381/jot.2025.24.2.a4

authorisation two separate participants are required. This is a variant of Dynamic Separation of Duty (DSD) where a user may not function in more than one role during a single authorisation, particularly relevant if role membership can change over time (Ultra & Pancho-Festin 2017).

To support such scenarios, we define modelling concepts for roles, role membership, alternatives, and cardinalities for authorisation relations. While existing RBAC models are static (Penelova 2021), we support dynamic RBAC, where dedicated role owners can add and remove members at runtime. An extension of the smart contract modelling language *iContractML* 2.0 (Hamdaqa et al. 2022) by RBAC and multi-party authorisation allows us to create, test and verify ACPs across platforms. The diagrammatic language helps developers to design and review RBAC models and we map to the smart contract languages Solidity and DAML[1] to generate and execute tests. DAML supports multi-party authorisation but no roles nor alternatives. Other languages such as Solidity use assertions for access control but do not support multi-party authorisations. Our model-based approach expresses these differences at a higher level and supports code generation to DAML and Solidity.

Any code implementing critical security properties must be tested. This also applies to access control code, even if we can generate such code from the model directly because (1) patterns and libraries used may not be compatible with technical requirements, (2) code generation for just one aspect, such as access control, is necessarily partial, so code will have to be completed manually and (3) code will be modified during software maintenance and evolution, introducing the risk of errors. Moreover, if models are created as documentation for existing implementations, they need to be confirmed as accurate through testing. To support model-based testing, we generate test scripts that include assertions as oracles to run automatically and fail when encountering access control errors.

We realise this model-based approach to analysing and testing ACPs by modelling the operational semantics of our DSL as a graph rewriting system.

1.2. Graph Rewriting as Semantic Engine

Our role-based access control models (RBACMs) describe complex requirements for the sets of authorisers that can accept or reject an access request, subsuming propositional logic over basic propositions of the form "participant P must authorise action A" but going beyond this with statements with multiplicities such as "n members of role R must authorise action A" and demanding that each participant can only act as authoriser once. The semantic information, which sets of authorisers are sufficient based on a given model, is obtained from a graph rewriting system operationalising the process of requesting authorisation for an action, computing a permissible set of authorisers, allowing them to confirm their authorisation and accepting or rejecting the request. Accepting or rejecting traces are produced by the Groove model checker (Ghamarian et al. 2012) using a start graph and temporal logic formulas derived from the RBACM. Test scripts in DAML are produced based on a test model using a Java EMF application and Acceleo templates. We evaluate the approach and tool chain using model-based mutation testing: We seed errors in the RBACM, generate implementations and apply our tests to discover the errors. We detail this process for the Homework Grading model described above and report on a second case study on managing financial transactions.

Our tests discover design-level faults in access control code. Design-level faults are deviations from the access control requirements expressed in an RBACM, that can arise from implementing (correctly) a faulty model, not from arbitrary implementation errors. This limitation is in line with related work on testing access control policies (Daoudagh et al. 2015; Martin & Xie 2007) which must be complemented by code-based techniques for comprehensive testing.

Dynamic access control is supported by a graph rewriting model that can add and remove role members at runtime. Apart from generating test cases, the operational semantics can answer questions on the model, such as: *Who can authorise a transaction if a participant is away?* or *If a given sequence of actions cannot be authorised, can I enable the process by adding a new member to a role?* where adding a role member itself requires authorisation. As a use case, we generate scenarios responding to such questions using OpenAI's o1-preview and o1-mini to validate them with the Groove model checker, combining the domain knowledge of LLMs with the safety of a formal model. We demonstrate the generation of complex dynamic test scenarios and discuss an intelligent help function to advise users on how to work within a given RBAC policy to achieve their goals.

1.3. A Recipe for Behavioural DSLs

While addressing a logically challenging problem, we present the approach as a blueprint for testing and validation based on domain-specific languages with complex graph-like structure and operational semantics. The general recipe is as follows:

1. define DSL using a metamodel and diagrammatic syntax
2. define operational semantics for the DSL by graph rewrite rules over a type graph extending the metamodel by dynamic elements to represent the runtime state
3. translate DSL instances (models) into a start graph (an instance of the type graph) to be rewritten by the rules
4. use a graph rewriting model checker to generate scenarios for temporal logic formulas describing the desired test cases
5. use the model checker to validate scenarios suggested by LLMs for testing or helping users at runtime
6. create executable tests from validated scenarios through a test model and code generation

Using metamodels to define and implement DSL is a standard approach, well supported by tools to create model editors and compilers. An operational semantics is needed for testing and behavioural analyses, so it is best to make this semantics explicit and use it as the engine to generate or validate test scenarios.

[1] Digital Asset Modelling Language: https://www.digitalasset.com/developers

Our RBACMs define behaviour indirectly by constraints on the possible actions. Hence their operational semantics is given in a structure-driven way (enabling actions based on the structure of the current state) rather than through control or message flow (deriving traces or transition systems from a program, state machine, or communication model).

Representing model and runtime states by graphs, we use graph rewriting to define operational semantics of modelling languages (Engels et al. 2000; Durán et al. 2017). In particular, making the model a part of the start graph we achieve two objectives: Our rules describe a fixed model interpreter that works across all models, and the model itself can be changed at runtime, for example by adding or removing role members, softening the distinction between model and data state.

The innovation in this paper is to use the graphical operational semantics to generated tests from counterexamples to LTL formulas that describe the scenarios to be validated through testing. By repeatedly invoking the model checker with augmented formulas we generate all relevant traces, which are then translated into test cases by way of an intermediate test model. This allows for the reuse of tools and offers a justification for the correctness of the approach, which would be hard to establish for any ad-hoc generation of test cases from a model.

Hence, while the various languages and tools used for operational semantics, mappings, and model checking are exchangeable, they represent a template for a general approach to testing and behavioural analyses for DSLs with graphical operational semantics.

We describe the approach in more detail in the next section before presenting the evaluation results, discussing related and work and drawing conclusions.

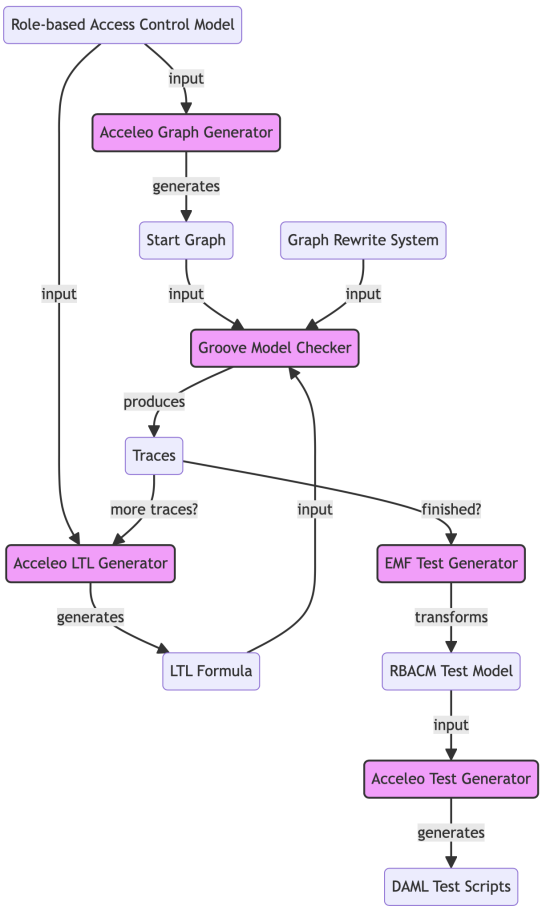

Figure 1 Outline of our approach

2. Testing from Operational Semantics

This section describes RBACMs, defines their operational semantics by graph rewriting and explains the generation of DAML tests. To represent RBACMs the *iContractML* metamodel (Hamdaqa et al. 2022) was extended by concepts such as roles and authorisations (Al-Azzoni & Iqbal 2023) including an extension to the visual notation. Multi-party authorisation and code generation to Solidity and DAML were added in (Al-Azzoni & Heckel 2023).

A sample RBACM is shown in Fig. 2. Transactions are represented by λ symbols with dashed arrows to their contracts and participants shown as human figures. Roles are black circles inside white squares, their membership shown by blue arrows and authorisation relations by red arrows. The latter are labelled by cardinalities indicating how many members are required to authorise the transactions. The language also supports alternative authorisations indicated by a $\vee$ (or) symbol within a white square (Al-Azzoni & Heckel 2023; Al-Azzoni & Iqbal 2023). From an instance of this extended metamodel, we generate test scripts in DAML as outlined in Fig. 1.

In real RBAC policies, roles can have many members. A school may have hundreds of homework submitters, and the membership can change over time. Hence, our RBACMs only represent sample members for analysing or testing policies.

A DAML test script invokes smart contract operations called *choices*. To determine the order in which these choices should be called to realise scenarios ending in (correctly or incorrectly) granted or rejected access requests, we created a graph rewriting system specifying the operational behaviour of RBACMs with basic access control operations such as for participants to authorise access and requests being granted or rejected based on the set of authorisers.

The RBACM is translated into a start graph by model-to-text transformation templates in Acceleo[2], generating the graph in Groove's textual Graph Exchange Language (GXL)[3] representation. Templates also generate a set of Linear Temporal Logic (LTL) formulas to search for particular traces realising the desired scenarios. To generate all traces satisfying a certain condition, Groove is invoked repeatedly with augmented LTL formulas to rule out all traces already generated.

A Java EMF[4] application transforms traces in Groove GXL format into corresponding test cases, adding new test-specific information to the RBACM. From this test model we generate the DAML test scripts using another set of Acceleo templates.

All steps are fully automated. Sect. 2.1 presents an RBACM as an *iContractML* metamodel instance. Sect. 2.2 discusses

[2] Acceleo: https://eclipse.dev/acceleo/
[3] GXL: https://userpages.uni-koblenz.de/~ist/GXL/index.php
[4] Eclipse Modeling Framework: https://eclipse.dev/modeling/emf/

the Groove graph rewriting system, including the start graph and trace generation. Sect. 2.3 presents the extension to the *iContractML* metamodel to support test script generation.

2.1. Access Control Models to Smart Contracts

To demonstrate our approach, we use the RBACM in Fig. 2. The contract defines who can submit and grade a homework. For testing purposes there are five sample participants: *Alice*, *Bob*, *Chris*, *Doug*, and *Emma*. The contract defines two roles: *HWSubmitter* and *HWEvaluator*. Authorisation by at least one member in the *HWSubmitter* role is required to execute the *submitHW* transaction. The *gradeHW* transaction requires authorisation by *Alice* in addition to *Bob* or two members distinct from *Alice* of the *HWEvaluator* role. The requirement that authorisers from the *HWEvaluator* role be distinct from *Alice* exemplifies the Dynamic Separation of Duty (DSD) principle where a user may not function in more than one role during an authorisation (Ultra & Pancho-Festin 2017).

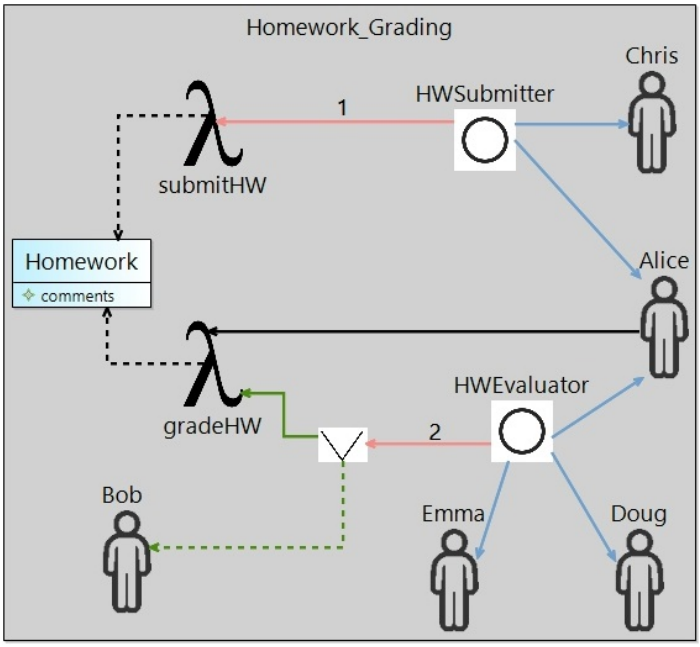

Figure 2 A sample RBACM: The Homework Grading contract

From this RBACM, the following DAML code is created by our code generation templates, defining the *HWEvaluator* role.

```
-- Template defining (and creating) role
-- with list of parties as initial members
template HWEvaluator_Role
  with
    default_owner : Party    -- Initial role owner
    owner : Party            -- Current role owner
    members : [Party]        -- Initial members
  where
    signatory default_owner  -- Must create contract
    observer owner           -- Can observe changes

    -- Choice allowing current to assign new owner
    choice SetNewOwner_HWEvaluator_Role
      : ContractId HWEvaluator_Role
      with
        newOwner : Party     -- Party to be new owner
      controller owner       -- Current owner can
      do
        assertMsg "newOwner cannot be equal to owner"
          (owner /= newOwner) -- No redundant change
        create this with     -- Updated contract
          owner = newOwner   -- with new owner

    -- Choice allowing owner to update members
    choice SetMembers_HWEvaluator_Role
      : ContractId HWEvaluator_Role
      with
        newMembers : [Party] -- New members list
      controller owner       -- Current owner can
      do
        create this with       -- Updated contract
          members = newMembers -- with new members
```

A DAML template is similar to a class and its instances are contracts. The main difference is the explicit handling of ownership and permissions to observe and control the execution of choices (operations). To support immutability, whenever a choice changes the state of a contract, a new version is created.

The signatory *default_owner* of the contract is the party that deploys the contract. The owner can be changed by the choice *SetNewOwner_HWEvaluator_Role*. The *owner* can set the members of the role *HWEvaluator_Role* through the choice *SetMembers_HWEvaluator_Role*, or by setting the contract's argument *members* when the *HWEvaluator_Role* template is instantiated.

The template below implements the *Homework_Grading* contract.

```
template Homework_Grading
  with
    -- Define contract owner, roles, parties
    -- owner creates; parties, role members observe
    owner : Party
    hWSubmitterRole: HWSubmitter_Role
    hWEvaluatorRole: HWEvaluator_Role
    alice : Party
    bob : Party
    chris : Party
    doug : Party
    emma : Party
    comments : Text
  where
    signatory owner
    observer alice,bob,chris,doug,emma,
      hWSubmitterRole.members,hWEvaluatorRole.members

    -- hWSubmitter can submit homework with comments
    choice SubmitHW
      : ContractId Homework_Grading
      with
        hWSubmitter : Party
        new_comments : Text
      controller hWSubmitter
      do
        assert (hWSubmitter `elem` -- Is role member
          hWSubmitterRole.members)
        create this with           -- Update contract
          comments = new_comments  -- with comments

    -- alice and graders in p1 can grade
    choice GradeHW
      : ContractId Homework_Grading
```

```
      with
        p1 : [Party]
        new_comments : Text
      controller alice, p1
      do
        let set1 = DA.Set.fromList p1 -- Graders set
        let set2 = DA.Set.fromList
          (hWEvaluatorRole.members)    -- Role set
        assert  (bob `elem` p1  ||     -- Bob or
          (DA.Set.isSubsetOf set1 set2 &&
          DA.Set.size set1 >= 2))    -- 2+ evaluators
        create this with             -- Update
          comments = new_comments
```

A choice's controllers are the parties whose authorisation is required to exercise the choice. Using DAML assertions, it is possible to fail the transaction corresponding to the choice if the assertion's predicate does not hold. Hence, for example, the choice *SubmitHW* requires the authorisation of one member in the *HWSubmitter_Role* role. Also, the choice *GradeHW* requires the authorisation of *Alice*, in addition to a list of parties *p1* that should include *Bob* or at least two members in the role *HWEvaluator_Role*. Note that in DAML, if several parties are listed as controllers for a choice, they have to be distinct, and *set2* is the set of *HWEvaluator_Role* members to which *Alice* does not belong. The effect is that this choice requires authorisation by *Alice* in addition to *Bob* or at least two members of the *HWEvaluator_Role* excluding *Alice*, as specified by the RBACM.

2.2. Operational Semantics via Graph Rewriting

The operational semantics of an RBACM is defined by translating it into a graph that, extended to represent runtime state information, is the start graph for a Groove graph rewrite system. The type graph for this system is shown in Fig. 3 and a start graph derived from the RBACM in Fig. 2 is shown in Fig. 4.

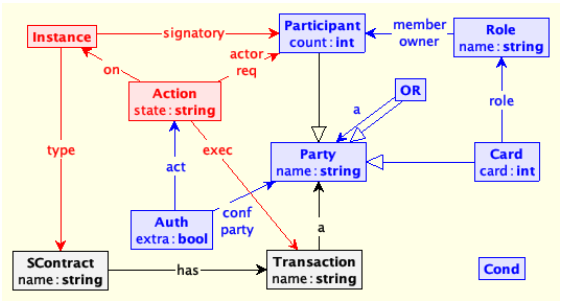

Figure 3 Type graph of Groove model

Grey elements derive from the *iContractML* metamodel, blue ones represent access control data and red elements are the runtime state. The start graph shows the *Homework_Grading* contract with its two transactions. It assigns access rights for *submitHW* to a single ($card \to 1$) member of the *HWSubmitter* role. An instance of the contract is shown with *Alice* as a signatory. The *gradeHW* transaction is authorised by *Alice* and either *Bob* or 2 members ($card \to 2$) of the *HWEvaluator* role. This applies to type and instance graphs, but not to rules where colours are used to distinguish pre- and post-conditions.

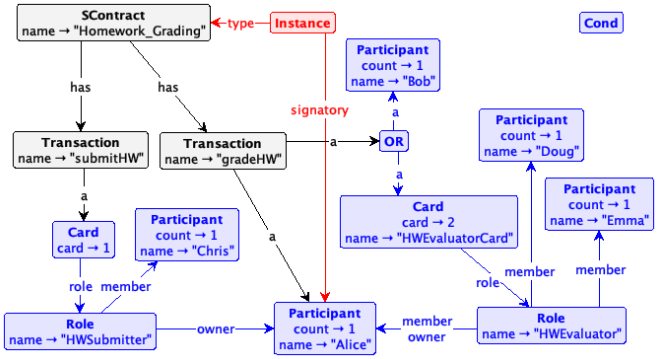

Figure 4 Start graph of Groove model

The first rule (*createAction* shown in Fig. 5) can create actions which, once authorised, will execute a transaction. Authorisations are then computed in two phases. First, a permissibly subtree is selected of the and/or tree describing the authorisation requirements of the transaction. Three of the rules implementing this recursive descent are shown in Fig. 6. The selection works by creating an *Auth* object for all required parties at the top level and then moving the *party* link down to alternatives and role members.

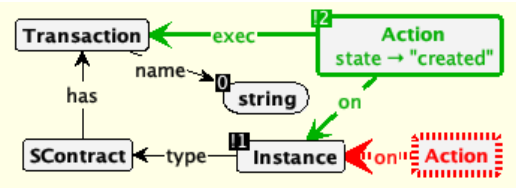

Figure 5 Rule *createAction*

In Groove's visual notation for rules, grey elements are required and preserved, green (with bold orders) represents creation of nodes or edges or updates of attributes, and blue (dashed lines or borders) means deletion. The $\forall$ node linked to the *Party* and *Auth* nodes represents a multi rule, applied once for each *Party* node to create a separate *Auth* node. The red/green-dashed edge labelled *party* in the middle rule *selectAlternative* says that this edge is only created if it does not already exist. This is a combination of edge creation with an application condition. Another negative condition is shown in red in the bottom rule *selectMember*, which also uses a $\forall$ node q to count the number of role members already assigned as $q.count$ in the *Card* node. There is a similar rule *selectMember* with an attribute condition $card \to g.count + 1$ that deletes the *party* edge from the *Auth* node, finishing the selection of role members.

Once selection is complete, as shown in Fig. 7, all selected participants are referenced by *party* links from *Auth* nodes. They then have an option to confirm authorisation, replacing the *party* by a *conf* edge, so we can decide if the action should be accepted.

Finally, we accept or reject the execution. Note that the rules in Fig. 9 do not check if the action was authorised correctly because we want to create traces to test for both correctly and incorrectly accepted and rejected actions. Correctly rejected traces do not have authorisation and end in rejection. We also

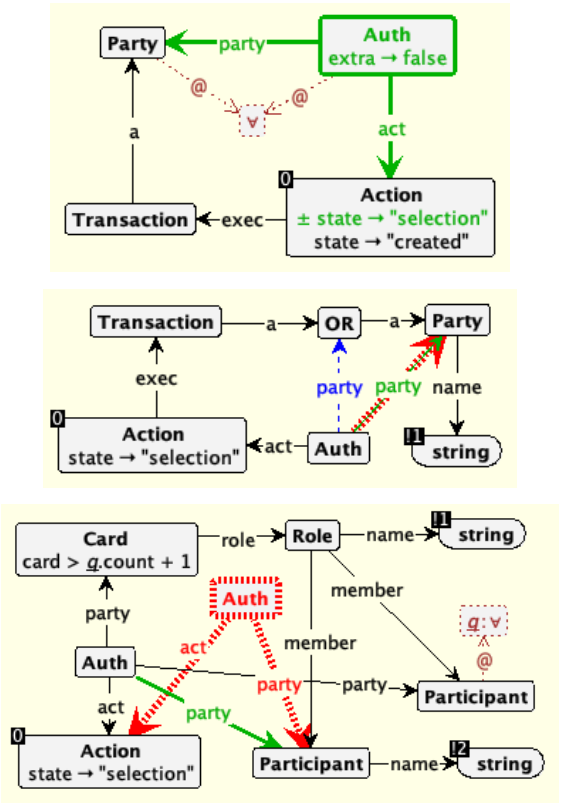

Figure 6 Top to bottom: rules *selectAllParties, selectAlternative* and *selectMember*

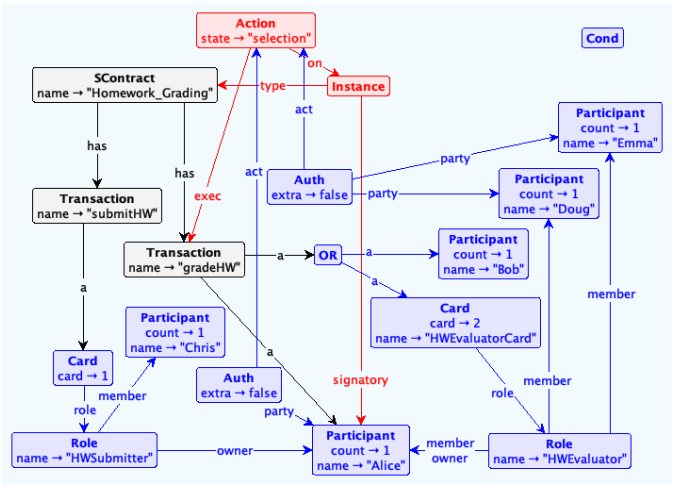

Figure 7 Sample state graph after selection of authorisers

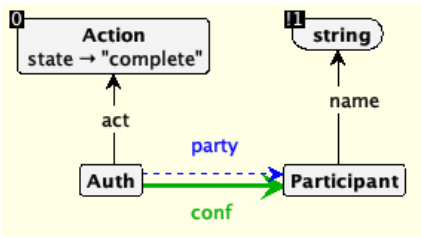

Figure 8 Rule *confirmSelection*

need test cases for incorrect behaviours, where authorisation should be granted but is rejected, and vice versa. Hence our rewrite system allows correct as well as incorrect behaviours which are filtered by LTL formulas to generate the desired traces. For example the following safety condition will generate incorrect accept traces,

```
F actionForTransaction("gradeHW") &
G (actionAccepted("gradeHW") ->
   allConfirmed("gradeHW"))
```

while this one represents lifeness, generating incorrect reject traces.

```
F actionForTransaction("gradeHW") &
G (allConfirmed("gradeHW") ->
F actionAccepted("gradeHW"))
```

The use of both lifeness and safety reflects complementary security properties such as Confidentiality and Authenticity which are supported by granting access only when correctly authorised (a safety property), while Availability requires that that access is always granted if it is correctly authorised (a lifeness property).

Basic propositions such as *actionForTransaction("gradeHW")* and *actionAccepted("gradeHW")* are implemented as graph patterns, *i.e.*, rules without effect that are used to label the states of the transition system generated by the Groove model checker. In particular, *allConfirmed("gradeHW")* checks if all authorisation requirements have been satisfied for the transaction, *i.e.*, there are no pending requirements, as expressed by the forbidden *Auth* node linked to a *Party* node via a *party* edge for its action.

To generate test cases representing correct behaviours, our LTL formulas are negated to create correct traces as counterexamples. The following formula generates correct accept traces. Using Groove's model checker, it took 401 milliseconds to generate a counterexample,

```
! ( F actionForTransaction("gradeHW") &
    G (
        ( actionAccepted("gradeHW") ->
          allConfirmed("gradeHW") ) &
        ( allConfirmed("gradeHW") ->
          F actionAccepted("gradeHW") )
    ) &
    F allConfirmed("gradeHW")
)
```

while this one generates correct reject traces in 236 milliseconds:

```
! ( F actionForTransaction("gradeHW") &
    G (
        ( actionAccepted("gradeHW") ->
          allConfirmed("gradeHW") ) &
        ( allConfirmed("gradeHW") ->
          F actionAccepted("gradeHW") )
    ) &
    ! F allConfirmed("gradeHW")
)
```

To generate all traces violating a certain condition, we augment the LTL formulas so that previously generated traces are ruled out. At this stage, we are not interested in the order in

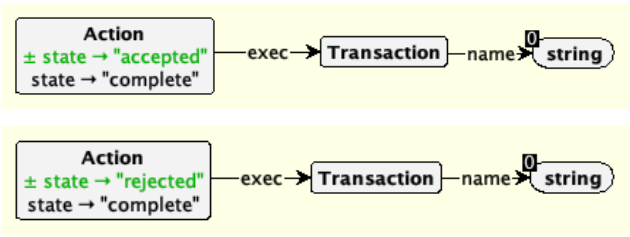

Figure 9 Top to bottom: rules *acceptAction* and *rejectAction*

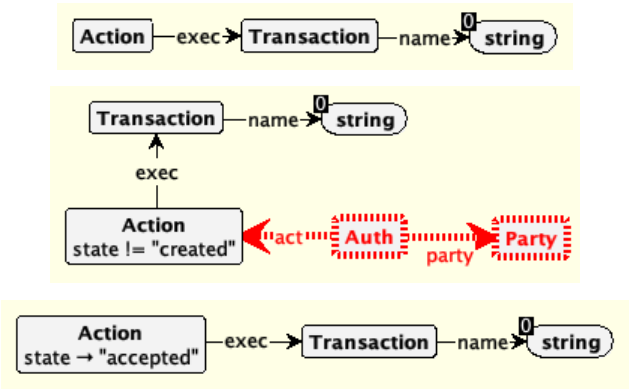

Figure 10 Top to bottom: patterns *actionForTransaction*, *allConfirmed*, and *actionAccepted*

which the authorisation requirements are confirmed, so we just rule out the particular subset of confirmations for each trace.

For example, the last formula above generates the trace shown in Fig 11 representing a counterexample with confirmations by {*Emma, Doug, Alice*}. To avoid generating this trace again it is encoded as

```
F ( confirmed(_,"Alice") & confirmed(_,"Emma") &
    confirmed(_,"Doug") )
```

which is added in negated form to the formula to prevent this counterexample from being returned again. This process is iterated until there are no further correct reject traces with new subsets of confirmations. Analogously, all correct accept traces are generated, while incorrect rejects/accepts are derived from correct accepts/rejects by swapping the final decision.

2.3. Model-based Test Script Generation

We adopt a model-based approach to generate the DAML test scripts from the traces of the Groove model checker, extending the RBACM metamodel to support the representation of test cases as a basis for code generation. In particular, a *TestCase* metaclass is added related to a *Transaction* and composed of a set of *Authorizations* which denote the participants or the number of role members who provide their authorisation for the execution of the related *Transaction*.

Instances of the RBACM test metamodel are created by a Test Generator implemented in Java EMF. The generator can insert authorisations not present in the traces. This is useful because the system under test may incorrectly require authorisation by a different participant than specified in the model, which corresponds to an incorrect accept trace with that participant

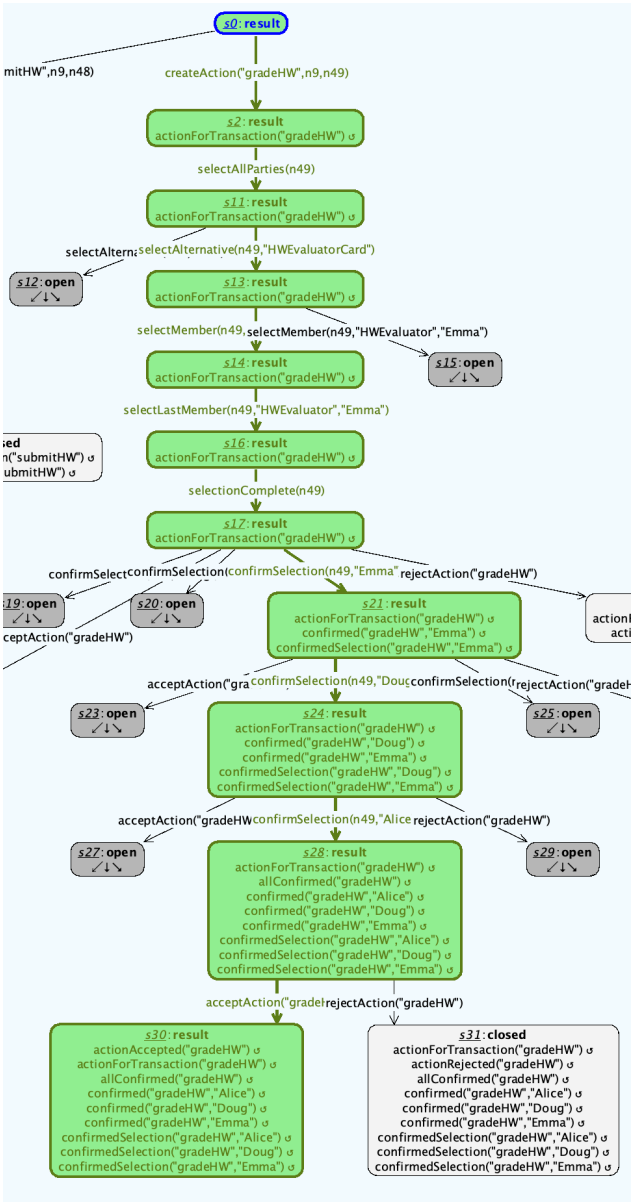

Figure 11 Visualisation of trace generated by Groove

added. We can also create a specified number of permutations of the required (and additional) authorisations, which should not produce different results in correct implementation but may help discover errors if the implementation enforces a certain order. The DAML scripts are generated from the RBACM test model by means of Acceleo templates.

Consider a test case for transaction *GradeHW* and two participants, *Alice* and *Bob*, who have provided their authorisation for the transaction. The generated DAML test script is:

```
-- Collect the required authorisations by Alice
-- and Bob
pending <- submit account1 do
  createCmd GradeHW_Attempt with
    alreadySigned =  [account1]
    finalContract = newCrontact
    p1 = [account1,account2]
```

```
  pending <- submit account2 do
    exerciseCmd pending Sign_GradeHW_Attempt with
      signer = account2

  -- Invoke the GradeHW transaction
  newContractid <- submit account2 do
    exerciseCmd pending Finalize_GradeHW_Attempt with
      signer = account2
      comments = "some comments"
      ghcId = newContractid
```

Note the attempt to instantiate the template *GradeHW_Attempt*. This template is also generated by our Acceleo Test Generator based on the test model (See Fig. 1). This template defines two choices to implement the Multiple Party Agreement pattern (*Online Source* The Multiple Party Agreement Pattern – DAML) in DAML by collecting the necessary authorisations. Once all required authorisations are collected, the choice *GradeHW* is invoked via the *Finalize_GradeHW_Attempt* choice.

The following is an Acceleo code to generate the LTL formulas corresponding to the *TestCases* of the *SContract* in an RBACM to identify the correct accept traces:

```
[let testcases : OrderedSet(TestCase) = con.testcase]
[for (tc: TestCase | testcases)]
[let t:Transaction = tc.transaction]
! ( F actionForTransaction("[t.name/]") &
    G (
        ( actionAccepted("[t.name/]") ->
          allConfirmed("[t.name/]") ) &
        ( allConfirmed("[t.name/]") ->
          F actionAccepted("[t.name/]") )
    ) &
    F allConfirmed("[t.name/]")
)
[let auths: OrderedSet(Authorization) =
 tc.authorization]
[for(auth:Authorization | auths)]
[let parts:OrderedSet(Participant) =
 auth.participant]
&
!  (
[for(part:Participant | parts)]
F (confirmed(_,"[part.name/]")
  [if (i<parts->size())]&[/if]
[/for]
)
[/let]
[/for]
[/let]
)
[/let]
[/for]
[/let]
```

Note the use of Acceleo control structures such as loops and *if* conditions. Variables in Acceleo are declared using the *let* blocks. The Acceleo code supports the iterative process to discover all correct accept traces as described at the end of Sect. 2.2.

Finally, DAML test scripts are executed against the templates in DAMLs Sandbox. DAML is a smart contract language running on a variety of blockchains and database platforms. The Sandbox is a testing tool included in the DAML SDK. It simulates a DAML ledger, allowing developers to test their applications and get quick feedback within DAML Studio.

3. Evaluation

We evaluate the effectiveness of the generated test cases for discovering design-level faults in access control code. Design-level faults are defined as deviations from the access control requirements expressed in a RBACM, that can arise from implementing a faulty (modified) model. The process is illustrated in Fig. 12.

We generate faulty access control code in the form of DAML templates by introducing faults into the RBACM used to generate DAML test scripts in Sect. 2. The resulting RBACM mutants are input to an Acceleo script generating DAML templates invoked when the test scripts are executed by the DAML Sandbox test ledger. We explain the steps in detail below.

3.1. Model-based Mutation Testing

To validate our ability to detect design-level faults, we use model-based mutation testing. In mutation testing, the effectiveness of a test suite is measured by its ability to detect artificial faults. Such faults (called mutations) are created by introducing small changes to the original program (Sánchez et al. 2022), usually directly in the source code (Gómez-Abajo et al. 2021). Instead, being interested in design-level faults, we modify the model from which the faulty code (mutant) is generated. The effectiveness of the test case set (its mutation score) is measured by the proportion of mutants detected. To kill (uncover) a model mutation, at least one test case should fail on the generated source code (D. Xu et al. 2012).

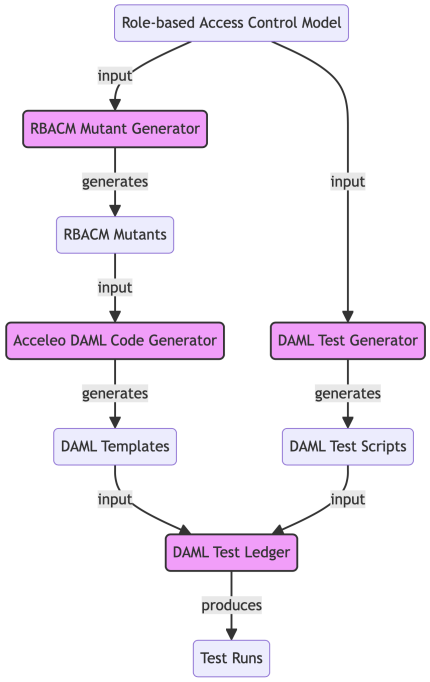

Figure 12 Evaluation process

We define the following basic rules for model mutation. They cover all primitive changes to concepts in the RBAC metamodel.

1. Add or remove a role member.
2. Add or remove a role to/from a transaction.
3. Increase or decrease a role relation cardinality.
4. Add or remove a participant to/from a transaction.
5. Add or remove a participant to/from an *OR* element.
6. Add or remove a role to/from an *OR* element.

To help the evaluation we developed a Java application to create mutants from a given RBACM. It randomly selects applicable elements while applying the specified rule. For example, for the first mutation rule, a random member in a randomly selected role is removed. Using this application, we have first created six mutants (one for each mutation rule):

1. Add participant *Frank* to role *HWSubmitter*.
2. Remove participant *Doug* from role *HWEvaluator*.
3. Change role cardinality of role *HWEvaluator* from 2 to 1.
4. Add participant *Doug* to transaction *submitHW*. (Hence, *submitHW* now requires the authorisation by *Doug* in addition to one member of role *HWSubmitter*.)
5. Add an *OR* element with participant *Bob* to transaction *submitHW*. (Hence, *submitHW* now requires the authorisation by one member in role *HWSubmitter* or *Bob*.)
6. Add role *HWEvaluator* to transaction *submitHW*. Hence, *submitHW* requires the authorisation by one member in role *HWSubmitter* in addition to one member in role *HWEvaluator*.

For each mutant, we generated the DAML code as described in Sect. 2. The generated DAML templates compile successfully.

As an example, consider the mutation to the RBACM in Fig. 2 in which *Frank* is added to the role *HWSubmitter*. In the generated DAML test script, the mutation is implemented as follows:

```
hWSubmitter_Role_Proposal <- submit account0 do
  createCmd HWSubmitter_Role_Proposal with
    hWSubmitter_Role = HWSubmitter_Role with
      default_owner = account0
      owner = account3
      members = [account1,account3,account6]
```

Note the insertion of *account6* which corresponds to *Frank* as a new member to the *HWSubmitter* role.

In addition, we manually created four more mutants by combining multiple mutations on the same contract:

7. Add participant *Frank* to role *HWSubmitter*, and remove participant *Chris* from the same role.
8. Add participant *Doug* to transaction *gradeHW*, and remove participant *Alice* from the same transaction.
9. Add an *OR* element with participant *Frank* to transaction *gradeHW*, and remove the original *OR* element.
10. Add role *HWEvaluator* to transaction *submitHW*, and remove the original role *HWSubmitter* as required authoriser to the same transaction.

These additional mutants can reveal defects due to incorrect access control implementations substituting rather than just adding or removing authorisation requirements. This can happen when new code is created by modifying copied existing code. Hence, the evaluation created a total of 10 mutants, each tested with a suite of 18 test cases (explained in detail below). This represents a total of 180 transaction attempts on the deployed contracts. After deploying the contracts on a DAML test ledger and running the test scripts, the results were examined using DAML Studio[5].

3.2. Test Case Generation and Execution

To create the test cases, we employed our tool chain, including Groove's model checker to generate the traces and Acceleo to generate DAML scripts. A total of 18 traces were created that cover all correct accept and reject scenarios. More test cases could have been defined by creating different permutations of the order in which participants authorise actions, or by including extra (unnecessary) authorisers; however, as the results will confirm, the 18 test cases were sufficient to kill all mutants.

Table 1 lists the test cases corresponding to the identified traces. For each mutant, at least one of the test cases listed in the table failed. For example, for the first mutant in which participant *Frank* is added to role *HWSubmitter*, Test Case 8, in which the expected outcome is *reject*, fails since the observed outcome was *accept* due to the mutation. The conclusion of the experiment is that the generated test cases killed all the mutants, producing a perfect mutation score.

3.3. FTA Case Study

To extend the evaluation, we explored the use of ChatGPT for generating RBACMs and workflows with dynamic access control. First, we asked ChatGPT to generate examples of multi-party RBACMs[6]. In the prompt, we required that models reflect real-world scenarios of complex automated workflows with multiple participants. This produced four models from which we selected the Financial Transaction Approval (FTA) model shown in Fig. 13 for further analysis. The RBACM defines three transactions and four parties, with more complicated authorisation requirements than in the homework grading RBACM.

Then, based on this model, we asked ChatGPT to create executable scenarios in the notation of Groove control programs, which constrain the order of rule applications for the model checker. We created a prompt explaining, by means of the Homework Grading example, the structure and semantics of RBACMs, provided the Financial Transaction Approval model

[5] Test Templates Using DAML Script: https://docs.daml.com/daml/intro/2_DamlScript.html#run-the-scripts

[6] Chat at https://chatgpt.com/share/656532a4-c238-43c0-964e-f8acb2ed507c

Table 1 Test Cases assessing correctness of *accept* and *reject* decisions for different transactions and sets of authorisers

Number	Transaction	Authorizers	Expected Outcome
1	*submitHW*	*Chris*	Accept
2	*submitHW*	*Alice*	Accept
3	*gradeHW*	*Alice*, *Bob*	Accept
4	*gradeHW*	*Alice*, *Bob*, *Emma*	Accept
5	*submitHW*	*Doug*	Reject
6	*submitHW*	*Emma*	Reject
7	*submitHW*	*Bob*	Reject
8	*submitHW*	*Frank*	Reject
9	*gradeHW*	*Chris*	Reject
10	*gradeHW*	*Alice*	Reject
11	*gradeHW*	*Bob*	Reject
12	*gradeHW*	*Doug*	Reject
13	*gradeHW*	*Emma*	Reject
14	*gradeHW*	*Doug*, *Emma*	Reject
15	*gradeHW*	*Doug*, *Bob*	Reject
16	*gradeHW*	*Alice*, *Doug*	Reject
17	*gradeHW*	*Alice*, *Emma*	Reject
18	*gradeHW*	*Frank*	Reject

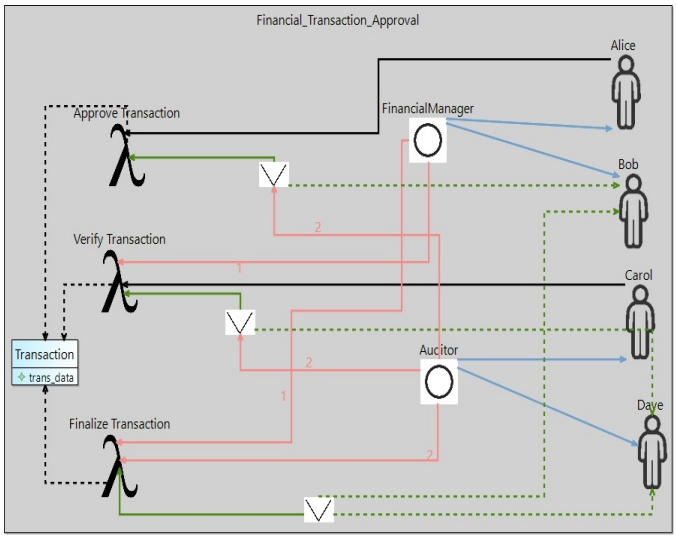

Figure 13 The Financial Transaction Approval (FTA) model

and asked for access control scenarios containing certain sequences of actions. The responses were inserted into the control of a Groove model with rules for adding and removing role members (if authorised by the roles' owners) to verify if these scenarios can indeed be executed under the operational semantics of the model. Hence, instead of focussing on individual actions, we are executing workflows with dynamic role membership. This is a planning problem of finding sequence of actions to achieve a given outcome.

The results show that o1-preview was able to correctly generate scenarios with dynamic role membership, while more basic models such as o1-mini fail to realise the condition that every participant can only be used once in authorising an action. For example, we asked: *What process do we need to execute if Dave is on holiday to perform the Approve, Verify, and Finalize actions?* while allowing Bob and Emma to be added as new members to the Auditor role.[7] The reply was in the form of the Groove control fragment shown below which, when inserted into the Groove model, was verified by the model checker.

```
GPT o1-preview:
  // Step 1: Add Bob and Emma as Auditors
  executeRoleAction("Bob", "Auditor", "addMember");
  // authorized by [Carol]
  executeRoleAction("Emma", "Auditor", "addMember");
  // authorized by [Carol]
  // Step 2: Perform Approve Transaction
  executeBusinessAction(instance,
    "Approve Transaction");
  // authorized by [Alice, Bob]
  // Step 3: Perform Verify Transaction
  executeBusinessAction(instance,
    "Verify Transaction");
  // authorized by [Carol, Alice, Bob, Emma]
  // Step 4: Perform Finalize Transaction
  executeBusinessAction(instance,
    "Finalize Transaction");
  // authorized by [Alice, Carol, Emma, Bob]
```

The operation *Verify Transaction* requires approval by, in logical terms, *(Carol* $\wedge$ *(1, Financial Manager)* $\wedge$ *(Dave* $\vee$ *(2, Auditors)))*. However, despite being instructed to check for

[7] Chat at https://chatgpt.com/share/66fc8bd0-1110-8010-a167-2021a8bd8d3e

duplicates in the prompt, in the generated control code below, o1-mini used *Bob* twice as authoriser of *Finalise Transaction*, after reusing *Carol* as authoriser for *Verify Transaction*.[8]

```
GPT o1-mini:
 executeRoleAction("Emma", "Auditor", "addMember");
 // authorised by [Carol]
 executeBusinessAction(instance,
  "Approve Transaction");
 // authorised by [Alice,Bob] or [Alice,Carol,Emma]
 executeBusinessAction(instance,
   "Verify Transaction");
 // authorised by [Carol, Alice, Carol, Emma]
 executeBusinessAction(instance,
  "Finalize Transaction");
 // authorised by [Alice or Bob, Carol, Emma, Bob]
```

Both chats had the same prompt, and while o1-preview took 30 seconds to answer, o1-mini needed less than 10 seconds. As expected, the Groove model checker verified the first scenario while rejecting the second. This shows that it is feasible to use LLMs to generate solutions to planning problems in this domain, but also that such solutions need to be verified by tools relying on formal (*e.g.*, graphical) representations of the problem.

When applying our methodology to the FTA RBACM, which features a more complex access control logic, a total of 69 test cases were generated and executed over 20 created mutants, resulting in 1380 tests. As the authorisation requirements become more complex, they become harder to satisfy, resulting in only five correct accept cases (where an action executed is accepted because its authorisation requirements are met), while the remaining test cases are all correct rejects. This test set is sufficient to kill all 20 mutants. The DAML contracts and test scripts are available in the paper's GitHub repository[9].

3.4. Threats to Validity

The evaluation results indicate that the approach is adequate for generating test cases to detect faults in the DAML access control code. The evaluation considered only two RBACMs. However, they each cover all features of the access control modelling language, so provide a good indication of the functional correctness of the test case generation process. In the following, we discuss the potential limitations of our evaluation in relation to scalability, the order of authorisations and additional authorisers, and the level and number of mutants considered for testing.

3.4.1. Scalability There are three ways in which RBAC policies could face scalability challenges.

1. The total number of roles, such as, in extreme cases, one role per access device and operation, or one per employee.
2. The number of role members, such as each employee in a large organisation.
3. The logical complexity and size of the access control requirement per operation or resource, *e.g.*, where many layers of authorisation or a majority of the members of a large role are required.

There is evidence that 1 and 2 can be large, but we were unable to find examples of 3 involving more than 4 authorisers (roles or individuals per operation).[10] This is credible because a single action requiring more than a handful of authorisation steps would be difficult to complete in practice. In particular, referring to the three dimensions of scale above.

1. Our approach handles access control requests per operation or resource, so the total number of roles in the organisation matters only for the purpose of visualisation in a graphical model, not for analysis or testing, as long as they are used in different access requests.
2. A larger number of members per role can lead to an explosion in the number of possible authorisation scenarios, but unless a large number is actually required for authorisation, each individual scenario will be small. If a larger number of role members are involved in approving a single action (*e.g.* a majority of board members must approve the appointment of a new chair), this is a voting system rather than what would usually be considered access control. In this paper, our models are used for generating test cases to validate logical correctness, not the scalability or performance, of the access control implementation. For this purpose, the use of a small number of role members for testing is in line with the usual practice of test data.
3. Our FTA case study uses 3 operations with 4 roles of individuals, each required per operation, which is at the upper limit of any concrete examples we could find.

3.4.2. Order of authorisations and additional authorisers We did not consider the order of authorisations for each test case, nor include authorisers beyond those required, even if both options are available in the test case generation tool, because the existing test cases were enough to kill all mutants. This is partly due to the model-based way mutants are created, *i.e.*, by introducing faults at model level and generating faulty access control code, we rely on our code generation templates that treat the required authorisers as a set rather than an ordered list. One could go beyond model-based mutation testing, applying mutations at the code level to incorrectly force a certain order of authorisation, and then test cases with different permutations could reveal different results. To avoid an exponential increase in the number of tests, a random selection of permutations could be chosen.

3.4.3. Level and number of mutants We apply mutation operators at model level, hence the generated mutants are not representative of all errors one could introduce at code level but only of design-level errors. This limits the scope of our evaluation, and of our approach, to testing for errors arising from correctly implementing an erroneous RBACM. This seems appropriate given the domain-specific nature of our models, and complementary methods can be used to test smart contracts comprehensibly at code level. Related work on mutation testing for access control policies (Daoudagh et al. 2015; Martin & Xie 2007) has similar scope and limitations.

[8] Chat at https://chatgpt.com/share/66fc8ea8-6618-8010-b7f4-78838fcf9101
[9] https://github.com/issamma1/Dynamic_Role_Based_Access_Control_Paper
[10] See chat: https://chatgpt.com/share/67fa0f46-e9bc-8010-87eb-6b887988069c

Another potential factor limiting the validity of the evaluation is the small number of mutants considered. For the purpose of testing the functional correctness of the test generation process and tool chain, this can be seen as sufficient because it covers all possible elementary changes to the model as well as some more complex changes. In addition we used a number of models of increasing complexity to validate separately both the correctness of the Groove graph rewrite system and generated LTL formulas, and of the various translations employed in the process. Hence, the end-to-end evaluation based on model-based mutation testing addresses the integration of the tool chain, while individual components have been validated separately.

It is also worth recalling that the main correctness argument is based on the fact that we use the operational semantics to generate the traces. Hence, assuming the correctness of the Groove model checker, our traces comply with the semantics of RBACMs *by construction*: A correct accept trace generated really represents a scenario where an action is sufficiently authorised and hence executed.

4. Related Work

This section discusses work related to modelling smart contracts and role-based access control.

4.1. Access Control for Smart Contracts

Model-based development has been proposed as a way to make smart contracts safer and more reliable in general (Velasco et al. 2023; Sánchez-Gómez et al. 2020) but without offering support for testing access control policies. In (Heckel et al. 2022; X. Xu et al. 2019), the authors use UML to model the architecture and business processes of blockchain applications. The *iContractML* language for smart contracts proposed in (Hamdaqa et al. 2020) is used in (Qasse et al. 2021) together with *iContractBot* for contract modelling using a chatbot. As domain-specific language, *iContractML* supports smart contract structure and code generation for multiple platforms, including Ethereum, Microsoft Azure, and Hyperledger Composer using the Acceleo model-to-text transformation language (Hamdaqa et al. 2020; Jurgelaitis et al. 2022). Our approach uses a similar DSL and suite of tools for representing and translating models but targets DAML while focussing on access control and testing. The authors of (Vandenbogaerde 2019; Sánchez-Gómez et al. 2021) present a metamodel and graph-based framework for Solidity smart contracts containing elements such as functions, events, and data structures, without considering other possible platforms, which support neither access control nor testing.

Several methodologies for model-based development of smart contracts include access control policies and the subsequent generation of Solidity source code. To model access control for smart contracts, (Brousmiche et al. 2018; Töberg et al. 2022; Al-Azzoni & Heckel 2023; Al-Azzoni & Iqbal 2023) introduce model elements to describe RBAC policies such as organisations, users, and roles. Permissions are checked at runtime using additional smart contracts for users and roles to implement access control. While we use similar concepts to generate access control code, we go beyond these approaches to support test case generation and analysis. Other access control approaches for smart contracts (Achour et al. 2021; Cruz et al. 2018) target Ethereum's smart contract language Solidity and are not model-based.

4.2. Graph Rewriting Models for Access Control

Modelling access control policies through graph rewriting, we benefit from their visual nature and their formal and executable semantics. Graphs representing resources, agents, access rights, and constraints enable precise rule-based definitions of authorisation processes and can utilise theory and tool support to analyse such definitions. This approach was first used in (Koch et al. 2002) to model RBAC policies and was applied to access control for business workflows in (Wei et al. 2008). We follow a similar philosophy using graph rewriting as the main semantic engine but provide a DSL defining multiparty dynamic RBAC and support testing and analysis based on model checking.

4.3. Dynamic and Multi-party Access Control

RBAC specifies and enforces enterprise security policies that naturally correspond to an organisation structure (Samarati & De Vimercati 2000). Access is decided by the roles users have in the company. These roles represent the different jobs and responsibilities of individuals. Usually, only the system administrator has the right to control system security and assign roles to users (Kashmar et al. 2021; Boadu & Armah 2014). We follow this model but, in order to reflect existing features in DAML and organisational practices, add support for multi-party, alternatives, and cardinalities for authorisations. Moreover, in our graph rewrite models, role membership is dynamic, *i.e.*, a dedicated participant, the owner of a role, is able to add and remove members. This is an important feature in the workflows of real organisations facing changes in staff turnover and availability. In (D. Xu et al. 2012), Petri nets are used to model RBACs and generate tests. Access control operations are also into workflows, but there is no support for multi-party RBAC and role membership is static.

4.4. LLMs for Test Generation

In Sect. 3.3 we describe the use of ChatGPT to generate RBACMs and test scenarios. Recent research on LLMs for software testing is surveyed in (J. Wang et al. 2024). (Schäfer et al. 2024) presents an empirical evaluation of the effectiveness of LLMs for automated unit test generation without additional training. In the field of autonomous driving systems, (Chang et al. 2024) proposes an LLM-driven scenario generation framework designed for training and testing critical modules. The framework is shown to identify corner scenario cases; a very important feature to increase the reliability and usability of self-driving vehicles. Our use of GPT follows the line of (Schäfer et al. 2024) using a standard LLM with a prompt explaining the necessary concepts and tasks, in our case the structure of RBACMs, to generate models and test scenarios.

4.5. Test case generation for smart contracts

There exist several automated approaches for test case generation for smart contracts. In (Driessen et al. 2024), the authors introduce a tool for the automatic generation of test suites for Solidity smart contracts optimised for branch coverage using a genetic algorithm or a fuzzing-based approach. The authors in (X. Wang et al. 2024) propose a test generation approach based on data flow analysis for Ethereum smart contracts. This approach can automatically generate smart contract test cases with high branch coverage. In (Górski 2024), the author presents a test suite reduction method for smart contracts. Neither of them specifically test for compliance to access control policies.

5. Conclusion

We presented a method and tool chain to generate test cases and validate scenarios based on a DSL for multi-party role-based access control in smart contracts. The approach is driven by an operational semantics of the DSL defined by a graph rewriting and uses the Groove model checker to generate and validate access control traces. Building directly on the operational semantics of the DSL, we can be confident in generating traces that comply with this semantics.

The evaluation used model-based mutation testing, introducing random faults in the model and generating correspondingly faulty implementations as mutants to be discovered by the test cases. We applied this approach to two models of moderate complexity, covering all concepts of the language.

Our methodology for model-based testing is applicable to all behavioural models whose runtime states and transitions can be represented by graphs and graph rewriting. We discussed the features of such languages and outlined the general recipe.

Although all steps in the generation and evaluation are automated, a more comprehensive assessment requires a larger collection of models that reflect access control policies in different applications. We experimented with ChatGPT to generate "real-world" RBACMs in different domains. This approach could be used to generate a set of benchmark models.

The verification of dynamic RBAC scenarios generated by ChatGPT shows another direction of future work, where such a facility could be integrated into an application's help function to provide verified AI-based advice to users who may struggle to understand and realise complex access control requirements.

References

Achour, I., Idoudi, H., & Ayed, S. (2021). Automatic generation of access control for permissionless blockchains: Ethereum use case. In *Proceedings of the IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (pp. 45–50). doi: 10.1109/WETICE53228.2021.00020

Al-Azzoni, I., & Heckel, R. (2023). Modelling multi-party role-based access control policies for iContractML smart contracts. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)* (pp. 100–104). doi: 10.1109/ASEW60602.2023.00018

Al-Azzoni, I., & Iqbal, S. (2023). Model-driven approach for generating smart contracts for access control. In *Proceedings of the International Conference on Blockchain Computing and Applications (BCCA)* (pp. 112–115). doi: 10.1109/BCCA58897.2023.10338863

Boadu, E. O., & Armah, G. K. (2014, 09). Role-based access control (RBAC) based in hospital management. *International Refereed Journal of Engineering and Science*, *3*, 53–67. Retrieved from http://irjes.com/Papers/vol3-issue9/H395367.pdf

Brousmiche, K.-L., Mercenne, L., & Ben-Hamida, E. (2018). Blockchain Studio: A role-based business workflows management system. In *Proceedings of the IEEE Information Technology, Electronics and Mobile Communication Conference (IEMCON)* (pp. 1215–1220). doi: 10.1109/IEMCON.2018.8614879

Chang, C., Wang, S., Zhang, J., Ge, J., & Li, L. (2024). LLM-Scenario: Large language model driven scenario generation. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*. doi: 10.1109/TSMC.2024.3392930

Cruz, J. P., Kaji, Y., & Yanai, N. (2018, 03). RBAC-SC: Role-based access control using smart contract. *IEEE Access*, *PP*, 185-197. doi: 10.1109/ACCESS.2018.2812844

Daoudagh, S., Lonetti, F., & Marchetti, E. (2015). Assessment of access control systems using mutation testing. In *Proceedings of the International Workshop on Technical and Legal Aspects of Data Drivacy and Security* (pp. 8–13). doi: 10.1109/TELERISE.2015.10

Driessen, S., Di Nucci, D., Tamburri, D., & van den Heuvel, W. (2024). SolAR: Automated test-suite generation for Solidity smart contracts. *Science of Computer Programming*, *232*, 103036. doi: 10.1016/j.scico.2023.103036

Durán, F., Moreno-Delgado, A., Orejas, F., & Zschaler, S. (2017). Amalgamation of domain specific languages with behaviour. *J. Log. Algebraic Methods Program.*, *86*(1), 208–235. Retrieved from https://doi.org/10.1016/j.jlamp.2015.09.005 doi: 10.1016/J.JLAMP.2015.09.005

Engels, G., Hausmann, J. H., Heckel, R., & Sauer, S. (2000). Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, & B. Selic (Eds.), *«UML» 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings* (Vol. 1939, pp. 323–337). Springer. Retrieved from https://doi.org/10.1007/3-540-40011-7_23 doi: 10.1007/3-540-40011-7_23

Ghamarian, A. H., de Mol, M. J., Rensink, A., Zambon, E., & Zimakova, M. V. (2012, February). Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, *14*(1), 15–40.

Gómez-Abajo, P., Guerra, E., Lara, J. d., & Merayo, M. G. (2021). Wodel-Test: a model-based framework for language-independent mutation testing. *Software and Systems Modeling (SoSyM)*, *20*(3), 767–793. doi: 10.1007/s10270-020-00827-0

Górski, T. (2024). SmarTS: A Java package for smart contract test suite generation and execution. *SoftwareX*, *26*, 101698.

doi: 10.1016/j.softx.2024.101698

Hamdaqa, M., Met, L. A. P., & Qasse, I. (2022). iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Information and Software Technology*, *144*, 106762. doi: 10.1016/j.infsof.2021.106762

Hamdaqa, M., Metz, L. A. P., & Qasse, I. (2020). iContractML: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. In *Proceedings of the System Analysis and Modelling Conference* (pp. 34–43). doi: 10.1145/3419804.3421454

Heckel, R., Erum, Z., Rahmi, N., & Pul, A. (2022). Visual smart contracts for DAML. In N. Behr & D. Strüber (Eds.), *Graph transformation* (pp. 137–154). Cham: Springer International Publishing.

Jurgelaitis, M., Ceponiene, L., & Butkienė, R. (2022, 01). Solidity code generation from UML state machines in model-driven smart contract development. *IEEE Access*, *10*, 3-3. doi: 10.1109/ACCESS.2022.3162227

Kashmar, N., Adda, M., & Ibrahim, H. (2021, 12). Access control metamodels: Review, critical analysis, and research issues. *Journal of Ubiquitous Systems and Pervasive Networks*, *16*. doi: 10.5383/JUSPN.16.02.006

Koch, M., Mancini, L., & Parisi Presicce, F. (2002, 08). A graph-based formalism for RBAC. *ACM Transactions on Information and System Security*, *5*, 332–365. doi: 10.1145/545186.545191

Martin, E., & Xie, T. (2007). A fault model and mutation testing of access control policies. In *Proceedings of the International Conference on World Wide Web* (p. 667—676). doi: 10.1145/1242572.1242663

Mougayar, W. (2016). *The business blockchain: Promise, practice, and application of the next internet technology*. Wiley.

Online source. (The Multiple Party Agreement Pattern – DAML). Retrieved from https://docs.daml.com/daml/patterns/multiparty-agreement.html

Penelova, M. (2021, 12). Access control models. *Cybernetics and Information Technologies*, *21*, 77-104. doi: 10.2478/cait-2021-0044

Qasse, I., Mishra, S., & Hamdaqa, M. (2021). iContractBot: A chatbot for smart contracts' specification and code generation. In *Proceedings of the ieee/acm international workshop on bots in software engineering (botse)* (pp. 35–38). doi: 10.1109/BotSE52550.2021.00015

Samarati, P., & De Vimercati, S. C. (2000). Access control: Policies, models, and mechanisms. In R. Focardi & R. Gorrieri (Eds.), *International school on foundations of security analysis and design* (pp. 137–196). Springer. doi: 10.1007/3-540-45608-2_3

Sánchez, A. B., Delgado-Pérez, P., Medina-Bulo, I., & Segura, S. (2022). Mutation testing in the wild: findings from GitHub. *Empirical Software Engineering*, *27*(6). doi: 10.1007/s10664-022-10177-8

Sánchez-Gómez, N., Torres-Valderrama, J., García-García, J. A., Gutiérrez, J. J., & Escalona, M. J. (2020). Model-based doftware design and testing in blockchain smart contracts: A systematic literature review. *IEEE Access*, *8*, 164556–164569. doi: 10.1109/ACCESS.2020.3021502

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, *50*(1), 85–105. doi: 10.1109/TSE.2023.3334955

Sánchez-Gómez, N., Torres, J., RISOTO, M., & Garrido, A. (2021, 11). Blockchain smart contract meta-modeling. *Journal of Web Engineering*, 11. doi: 10.13052/jwe1540-9589.2073

Tapscott, D., & Tapscott, A. (2016). *Blockchain revolution: How the technology behind bitcoin is changing money, business, and the world*. Portfolio.

Töberg, J.-P., Schiffl, J., Reiche, F., Beckert, B., Heinrich, R., & Reussner, R. (2022). Modeling and enforcing access control policies for smart contracts. In *Proceedings of the IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)* (pp. 38–47). doi: 10.1109/DAPPS55202.2022.00013

Ultra, J., & Pancho-Festin, S. (2017). A simple model of separation of duty for access control models. *Computers & Security*, *68*, 69-80. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167404817300652 doi: https://doi.org/10.1016/j.cose.2017.03.012

Vandenbogaerde, B. (2019). A graph-based framework for analysing the design of smart contracts. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 1220–1222). doi: 10.1145/3338906.3342495

Velasco, G., Alves Vieira, M., & Carvalho, S. (2023). Evaluation of a high-level metamodel for developing smart contracts on the Ethereum Virtual Machine. In *Proceedings of the Anais do VI Workshop em Blockchain: Teoria, Tecnologias e Aplicações* (pp. 29–42). doi: 10.5753/wblockchain.2023.757

Vivar, A. L., Castedo, A. T., Orozco, A. L. S., & Villalba, L. J. G. (2020). An analysis of smart contracts security threats alongside existing solutions. *Entropy*, *22*(2), 203. doi: 10.3390/e22020203

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, *50*(4), 911–936. doi: 10.1109/TSE.2024.3368208

Wang, X., Yang, Y., Liu, L., Chen, Z., & Huang, S. (2024). Test case generation for Ethereum smart contracts based on cross-contract data flow analysis. *IEEE Transactions on Reliability*, 1–14. doi: 10.1109/TR.2024.3494798

Wei, Y., Wang, C., & Peng, W. (2008). Graph transformations for the specification of access control in workflow. In *Proceedings of the International Conference on Wireless Communications, Networking and Mobile Computing* (pp. 1–5). doi: 10.1109/WiCom.2008.2938

Xu, D., Thomas, L., Kent, M., Mouelhi, T., & Le Traon, Y. (2012). A model-based approach to automated testing of access control policies. In *Proceedings of the ACM Symposium on Access Control Models and Technologies* (p. 209—218). doi: 10.1145/2295136.2295173

Xu, X., Weber, I., & Staples, M. (2019). Blockchain in software architecture. In *Architecture for blockchain applications* (pp.

83–92). Cham: Springer International Publishing. Retrieved from https://doi.org/10.1007/978-3-030-03035-3_5 doi: 10.1007/978-3-030-03035-3_5

Issam Al-Azzoni is an associate professor in the College of Engineering at Al Ain University. His research interests include Model-Driven Engineering (MDE) with applications in software verification and validation. He has published several articles in journals and peer-reviewed conferences on MDE. Contact him at issam.alazzoni@aau.ac.ae, or visit https://aau.ac.ae/en/staff/issam-al-azzoni.

Reiko Heckel is a leading researcher and professor in software engineering at the University of Leicester, with a focus on graph transformation theory and its applications in model-based development, software re-engineering, testing, and the semantics of modeling languages. Heckel has co-authored the textbook Graph Transformation for Software Engineers and plays an active role in the academic community through leadership in conferences and associations. His work bridges formal methods and practical software engineering, advancing the automation and analysis of complex software systems. Contact him at rh122@leicester.ac.uk, or visit https://le.ac.uk/people/reiko-heckel.

Zobia Enum is a Visiting Lecturer at Birmingham City University, where she teaches Web Application Development. She was previously working as a Project Supervisor at the University of Leicester. Her research focuses on Model Driven Engineering (MDE), particularly the visual modelling of DAML smart contracts using class diagrams and graph transformation. She has also worked as a software developer, building scalable web applications aligned with modern industry practices. You can contact her at erumzobia@gmail.com.