Visual Smart Contracts: A Doodle in DAML

Reiko Heckel[1], Zobia Erum[1], Nitia Rahmi[2], and Albert Pul[3]

[1] School of Comp. and Math. Sci., University of Leicester, UK, rh122|ze19@le.ac.uk
[2] PT Bank Rakyat Indonesia (Persero) Tbk, Indonesia, nr185@student.le.ac.uk
[3] betterECO GmbH, Germany pulalbert2@gmail.com

Abstract. We present a case study in the use of visual contracts and graph transformations for representing DAML smart contacts, their operational semantics, and analysis.

1 Introduction

The Doodle example is used in introductory papers [3,4] to illustrate a range of DAML's features. It provides a good opportunity for evaluating the feasibility of our model-based approach consisting of extended (DAML-specific) class diagrams to declare DAML templates with their attributes and choices, smart visual contracts to specify the behaviour of choices, and graph transformation rules to capture their operational semantics.

A doodle is a voting system to schedule meetings, where an *organiser* invites *voters* to vote on a set of *options*, recording their preferences in *voting slots*. Everyone can vote at most once for each option, and votes are visible to all.

2 From DAML Data Structures and Templates to Class Diagrams and Typed Graphs

In this section, we introduce DAML and establish a link with object-oriented concepts by "reverse engineering" DAML code into UML class diagrams.

A class can be seen as a template for creating an object with certain features. In DAML, a *template* describes the features of a *contract*, including its ownership and access rights, attributes, and operations called *choices*.

2.1 DAML Code

The DAML code [5] has one data structure and two contract templates: As shown in the listing below, a `VotingSlot` record represents the data about an option, including the vote count and the list of parties who voted for it. The line `deriving (Eq, Show)` states that records' equality is based on identity and they support serialization. A `Doodle` contract (created by the organiser as signatory) offers choices (i.e., operations invoked by the specified controllers) to add and remove voters and issue invites (organiser), and to cast votes (voters). Individual `DoodleInvite` contracts will allow voters to access the `CastVote` choice.

II

```
data VotingSlot = VotingSlot
  with
    count : Int
    voted : [Party]
      deriving (Eq, Show)

template Doodle
  with
    name: Text
    organizer: Party
    voters: [Party]
    options: [Text]
    votes: TextMap VotingSlot
    open: Bool
  where
    signatory organizer
    observer voters
    ensure (unique voters) && (unique options)
    key (organizer, name): (Party, Text)
    maintainer (fst key)

   choice AddVoter : ContractId Doodle
      with
        voter: Party
      controller organizer
        do
          assertMsg
                "this doodle has been opened for voting, cannot add voters"
                (not open)
          create this with voters = voter::voters

    choice RemoveVoter : ContractId Doodle
      with
        voter: Party
      controller organizer
        do
          assertMsg
                "this doodle has been opened for voting, cannot remove voters"
                (not open)
          create this with voters = DA.List.delete voter voters

    preconsuming choice IssueInvites : ContractId Doodle
      controller organizer
        do
          assertMsg
```

```
          "this doodle has been opened for voting,
             cannot issue any more invites"
          (not open)
     DA.Traversable.mapA
          (\voter -> create DoodleInvite with doodleName = this.name,
             organizer = this.organizer, voter = voter)
          voters
     -- archive self
     create this with open = True

preconsuming choice CastVote: ContractId Doodle
   with
     voter: Party
     option: Text
     inviteId: ContractId DoodleInvite
   controller voter
     do
       invite <- fetch inviteId
       assertMsg
          "this invite was issued for a different doodle"
          (invite.doodleName == name)
       assertMsg
          "the voter casting the vote does not match the voter
          who received the invite"
          (invite.voter == voter)
       assertMsg
          "the organizer who issued the invite is not the one
          who created this doodle"
          (invite.organizer == organizer)
       assertMsg "this doodle not is open" open
       assertMsg "voters is not one of the invited voters"
                       (elem voter voters)
       assertMsg "this is not a valid option " (elem option options)
       let
         crtVotes = fromOptional
            (VotingSlot with count = 0, voted = [])
            (DA.TextMap.lookup option this.votes)
         updatedVotes = DA.TextMap.insert option
            (VotingSlot with count = crtVotes.count + 1,
              voted = voter :: crtVotes.voted)
            this.votes
       assertMsg
          "each voter is only allowed to cast one vote per option"
          (notElem voter crtVotes.voted)
       create this with votes = updatedVotes
```

2.2 Class Diagram

We illustrate the mapping by creating a class diagram from the DAML code. The result is shown in Fig. 1. A template maps to a class with the same name, and template parameters become its attributes or associations. In particular, attributes of party type are associations to the single party actor in the diagram, labelled by the appropriate prototypes. In the `Doodle` template, the organiser is the signatory and the voters are observers.

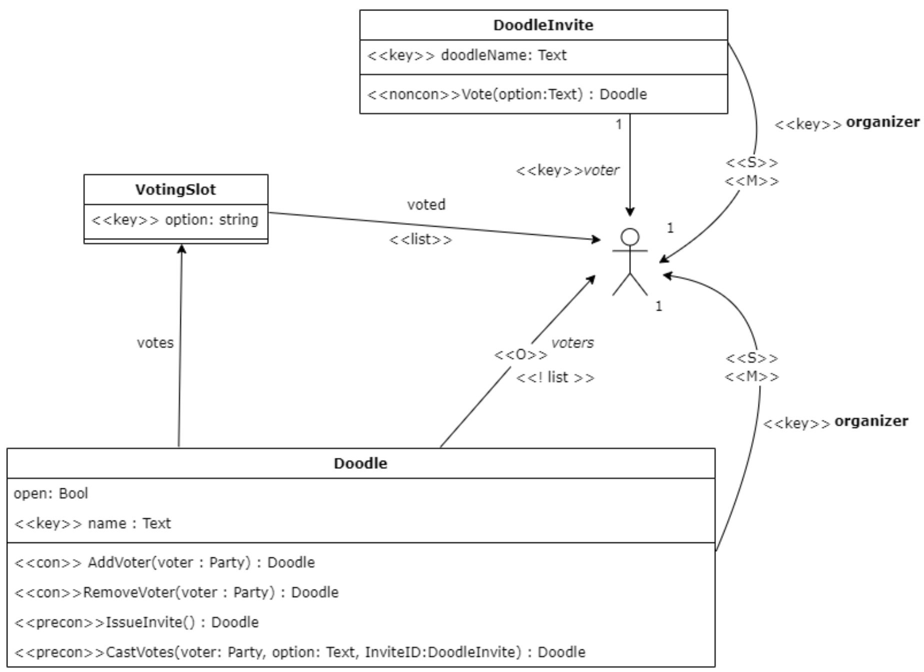


Fig. 1: Class diagram for Doodle case study

2.3 Type and Instance Graphs

Disregarding operations, a class diagram can be seen as an attributed type graph with inheritance [2]. Such a type graph defines a set of attributed instance graphs representing object structures, the possible data states of our system.

The type graph in Fig. 2 has node types for all contract and data structures with attributes as defined by the class diagram. It also defines a <op>Call node for each operation (choice or contract constructor) <op>with arguments and returns as attributes and edges.

An instance graph with one call node for the Doodle constructor, three parties and two voting slots is shown in Fig. 3.

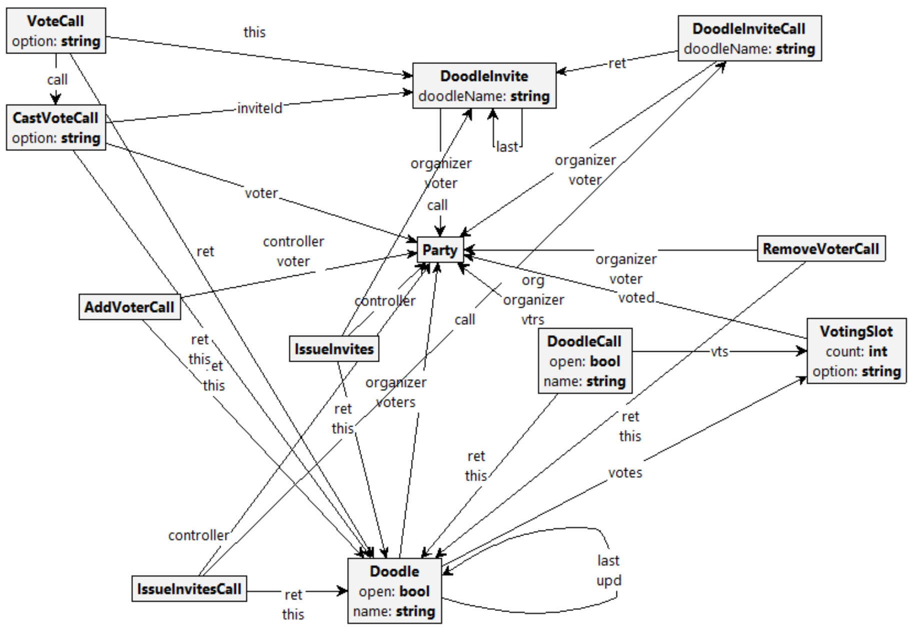


Fig. 2: Type graph for Doodle case study

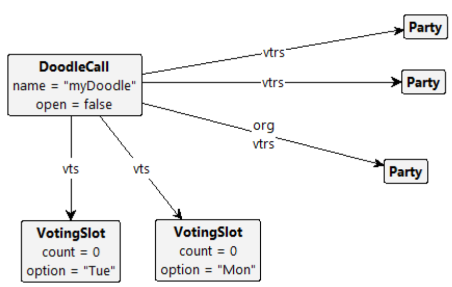


Fig. 3: Sample instance Graph for Doodle case study

3 From Template Constructors and Choices to Smart Visual Contracts and Graph Transformation Rules

A `Doodle` contract (created by the `organiser` as signatory) offers choices (i.e., operations invoked by the specified controllers) to add and remove `voters` and issue invites (`organiser`), and to cast their votes (`voters`).

3.1 Doodle Template

Constructor Doodle Fig. 4 shows how the template translates to a smart visual smart contract (VC) modelling the constructor. We use green colour for all elements created by the operation, i.e., the Doodle contract its references to parties organizer, voter and to VotingSlotss.

DAML

```
data VotingSlot = VotingSlot
  with
    count : Int
    voted : [Party]
      deriving (Eq, Show)

template Doodle
  with
    name: Text
    organizer: Party
    voters: [Party]
    options: [Text]
    votes: TextMap VotingSlot
    open: Bool
  where
    signatory organizer
    observer voters
    ensure (unique voters) && (unique
options)
key (organizer, name): (Party, Text) maintainer (fst key)
```

UML

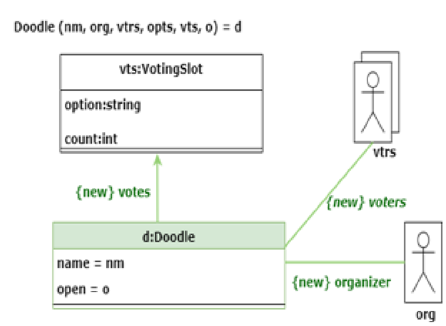


Fig. 4: Template `Doodle` as visual contract

Fig. 5 shows the corresponding semantic graph transformation rules, one for calling the choice, one for executing it and one for the return from the operation.

The first rule creates a DoodleCall node representing the call, which points to the last version of the contract `Doodle` (indicated by the last loop), and the input parameters org, vtr. The execution rule performs the operation, updates the `Doodle` contract and creates a ret link to the new version to be returned, and the return rule deletes the call node.

Choice RemoveVoter In the VC in Fig. 6 the label of the diagram: org > Doodle.RemoveVoter(vtr) = d' means that the actor org is the organizer of the contract. The execution of the choice archives the current version d and replaces

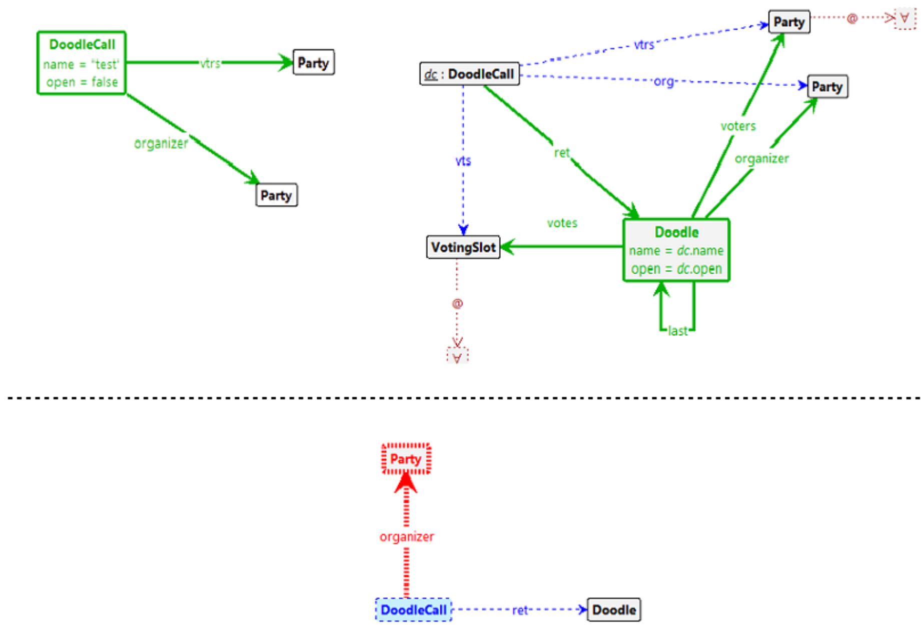


Fig. 5: Semantic rules for the call, execution and return of Doodle

it with a new version d' indicated by object id d -> d'. The choice removes the given vtr from the voters list, as represented by the deletion of the voters link.

Fig. 7 shows its corresponding semantic graph transformation rules, one for calling the choice, one for executing it and one for the return from the operation. The call rule represents the fact that `RemoveVoter` can be called at any time. It is part of the interface the `Doodle` contract offers to the organizer of the poll. The rule creates a RemoveVoterCall node representing the call, which points to the latest version of the this contract d, indicated by the last loop, and the input parameters org, vtr.

The precondition of the execution rule, shown in solid black (required but unchanged) and dashed blue (required and deleted) specifies the structure that must exist before the operation, including the call node. The post condition, which specifies the changes to the graph, is shown in solid black (unchanged) and dashed blue (deleted).

In the execution rule, the call node is unlinked from its input parameters and linked to its return, the new version d' of the contract. The blue dashed line from call node to doodle represents the removal of the voter from d that caused d to be archived and replaced by d'. Data and links are copied from the old to the new version and any updates, such as the removal of the link to the voter, are applied to the new version d'. The return rule just defines the deletion of the call node which, in this case, could have been done in the same rule because there are no nested calls.

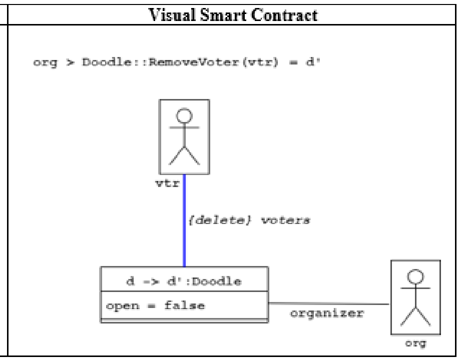
DAML	Visual Smart Contract
choice RemoveVoter : ContractId Doodle with voters: Party controller organizer do create this with voters = DA.List.delete voter voters	

Fig. 6: Choice `RemoveVoter` as visual contract

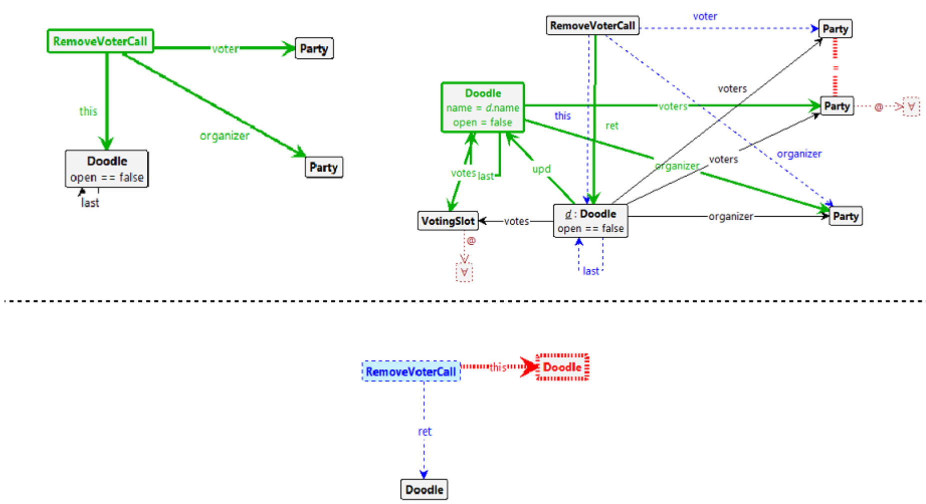


Fig. 7: Semantic rules for the call, execution and return of RemoveVoter

Choice AddVoter Fig. 8 shows the choice `AddVoter` in DAML and modelled as a visual smart contract. Fig. 9 shows its corresponding semantic graph transformation rules, one for calling the choice, one for executing it and one for the return from the operation.

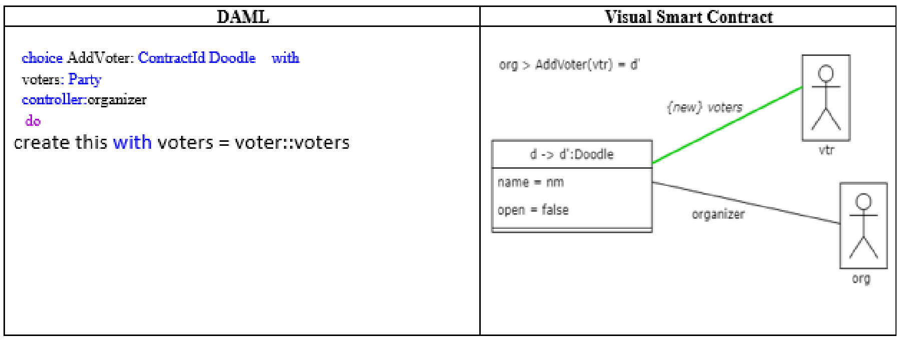


Fig. 8: Choice `AddVoter` as visual contract

Choice IssueInvites Fig 10 shows choice `IssueInvites` as a rule and modelled in visual smart contracts. Fig. 11 shows its corresponding semantic graph transformation rules, one for calling the choice, one for executing it and one for the return from the operation.

Choice CastVote Fig. 12 shows choice `CastVote` as a rule and modelled in visual smart contracts. Fig. 13 shows its corresponding semantic graph transformation rules, one for calling the choice, one for executing it and one for the return from the operation.

3.2 DoodleInvite Template

Constructor DoodleInvite In Fig. 14, the arrow labelled DoodleInvite(d.name,org, vtr) represents the invocation of the constructor for DoodleInvite contracts. According to the DAML code, the visual contract has to create the DoodleInvite object, initialise its attributes and associations according to the input parameters and return the new contract's id di. Like the one earlier, this default constructor VC is wholly derivable from the class definition and would not have to be defined explicitly. Fig. 15 shows its corresponding semantic graph transformation rules, one for calling the choice, one for executing it and one for the return from the operation.

From the code in Fig. 16, we see that when `Vote` is called, it exercises choice `CastVote`. Thus, the CastVote VC is represented with the call function notation.

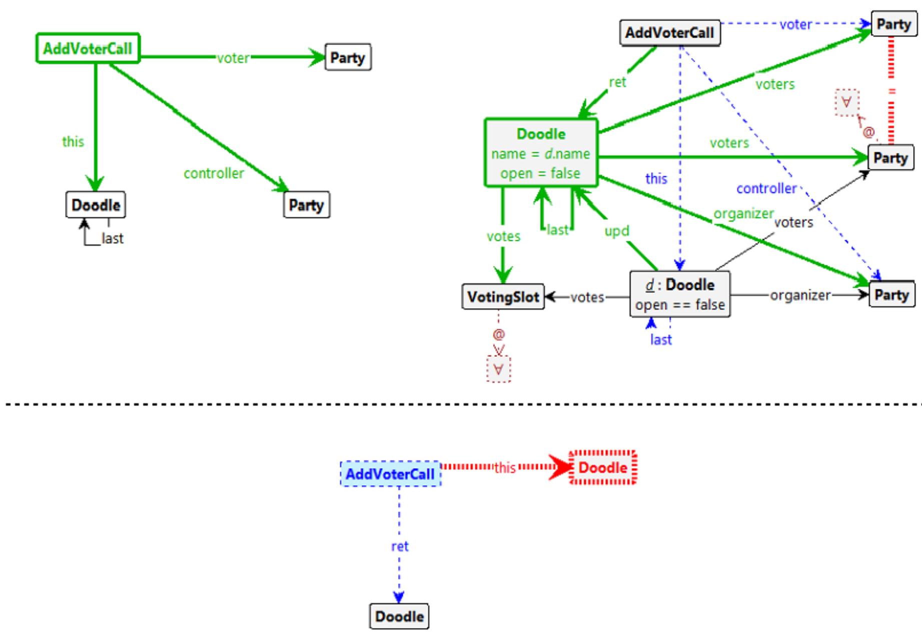


Fig. 9: Semantic rules for the call, execution and return of AddVoter

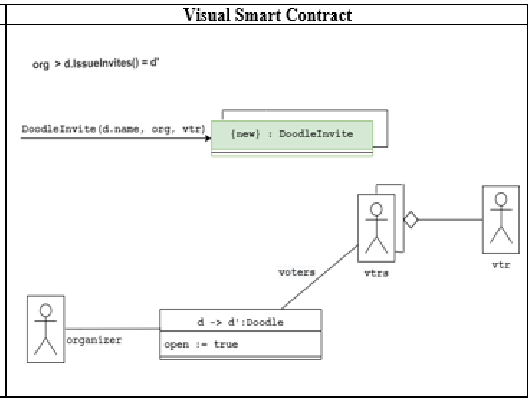
<table>
<tr><th>DAML</th><th>Visual Smart Contract</th></tr>
<tr><td>preconsuming choice IssueInvites : ContractId Doodle
controller
organizer do
assertMsg "this doodle has been opened for voting, cannot issue any more invites"
(not open)
DA.Traversable.mapA (\voter ->
create DoodleInvite with doodleName = this.name, organizer
= this.organizer, voter = voter) voters
-- archive self
create this with open = True</td><td>
</td></tr>
</table>

Fig. 10: Choice `IssueInvites` as visual contract

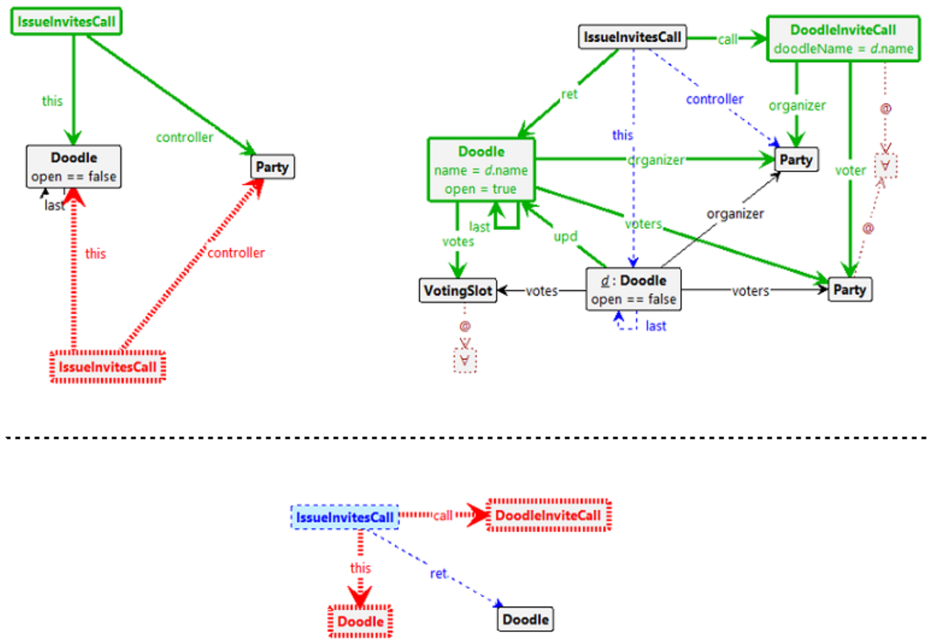


Fig. 11: Semantic rules for the call, execution and return of IssueInvites

DAML	Visual Smart Contract

```
preconsuming choice CastVote: ContractId Doodle with
  voter: Party
  option: Text
inviteId: ContractId DoodleInvite
 controller voter
          do
            invite <- fetch inviteId
              assertMsg "this invite was issued for a
 different doodle" (invite.doodleName == name) assertMsg
                     "the voter casting the vote
 does not match the voter who received the
 invite" (invite.voter == voter)
            assertMsg "the organizer who issued
 the invite is not the one who created this doodle"
 (invite.organizer == organizer)
            assertMsg "this doodle not is open"
 open
            assertMsg "voters is not one of the
 invited voters" (elem voter voters)
            assertMsg "this is not a valid option "
 (elem option options)
            let
              crtVotes = fromOptional (VotingSlot
 with count = 0, voted = []) (
              DA.TextMap.lookup option this.votes)
              updatedVotes = DA.TextMap.insert
 option (VotingSlot with count = crtVotes.count
 + 1, voted = voter :: crtVotes.voted)
 this.votes
            assertMsg "each voter is only allowed to
 cast one vote per option" (notElem voter
crtVotes.voted)
```

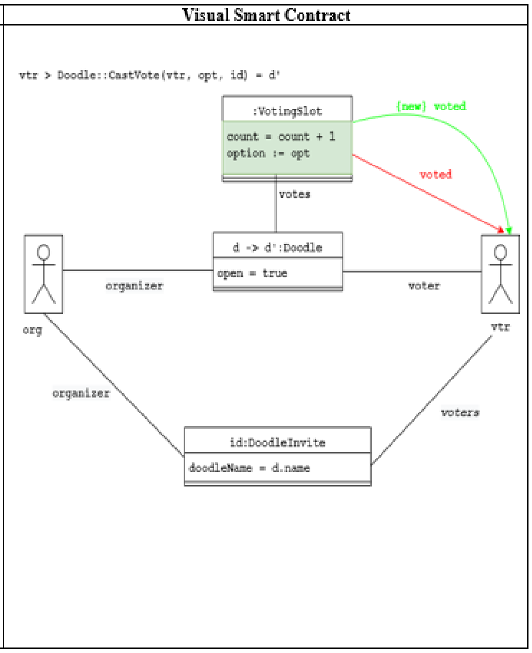


Fig. 12: Choice CastVote as visual contract

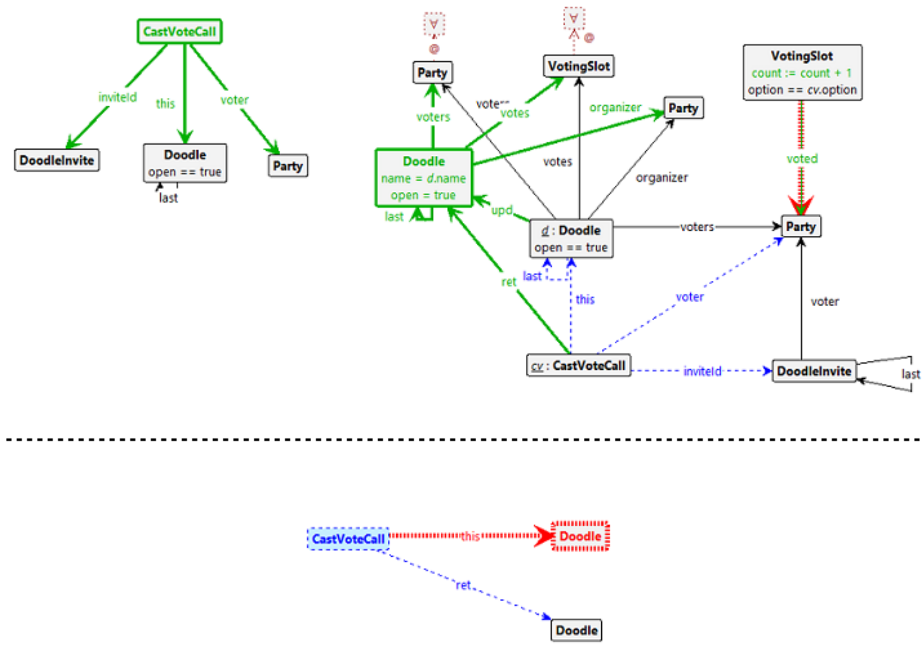


Fig. 13: Semantic rules for the call, execution and return of CastVote

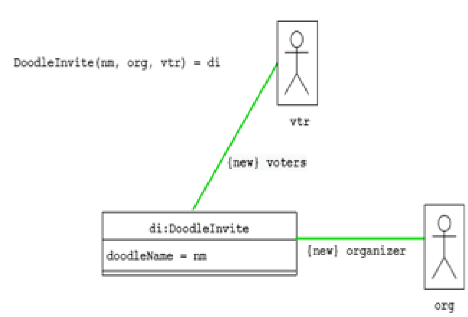
<table>
<tr><th>DAML</th><th>UML</th></tr>
<tr><td>template DoodleInvite
with
doodleName: Text
organizer: Party
voters: Party
where
signatory organizer
observer voter
key (organizer, voter, doodleName): (Party, Party, Text)</td><td>
</td></tr>
</table>

Fig. 14: Template DoodleInvite as Visual Smart Contract

Moreover, since this choice is nonconsuming, this choice can be called more than once without archiving the old contract.

Fig. 17 shows its corresponding semantic graph transformation rules, one for calling the choice, one for executing it and one for the return from the operation.

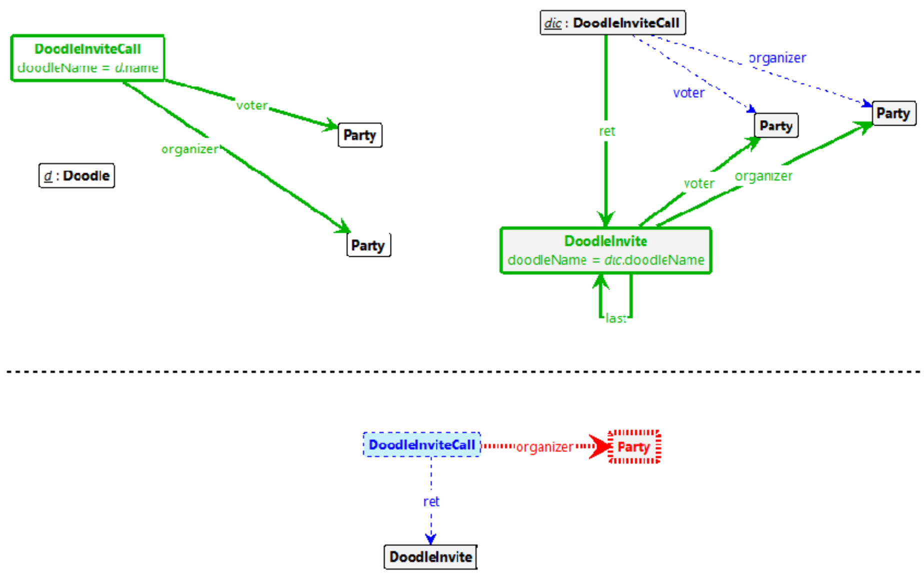


Fig. 15: Semantic Rule for the call, execution and return of `DoodleInvite`

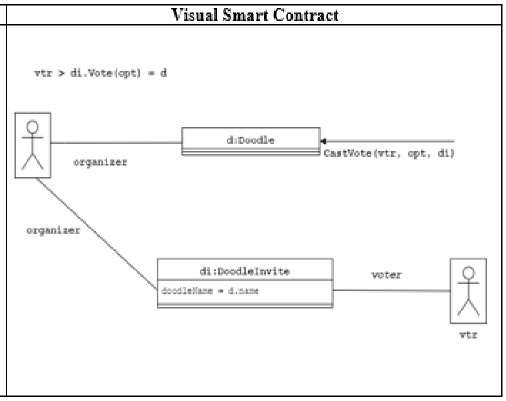
DAML	Visual Smart Contract
nonconsuming choice Vote : ContractId Doodle with option: Text controller voter do doodleContractId <- lookupByKey @Doodle (organizer, doodleName) doodleContract <- fetch (fromSome doodleContractId) assertMsg "this party does not own the doodle " (organizer == doodleContract.organizer) exercise (fromSome doodleContractId) CastVote with voter = this.voter, option = option, inviteId = self	

Fig. 16: Choice as a rule `Vote` as visual contract

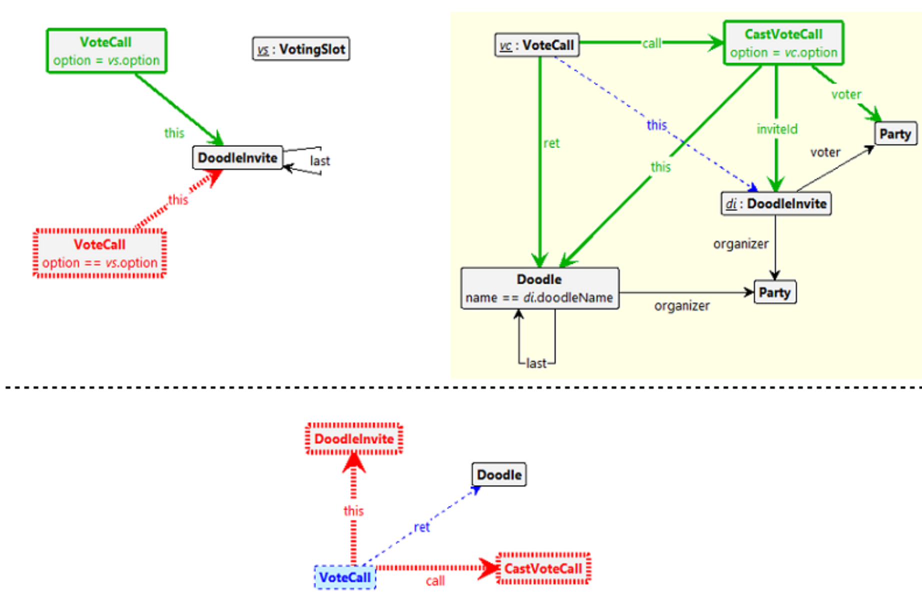


Fig. 17: Semantic Rule for the call, execution and return of `Vote`

4 Verification

We validated the model in Groove [1] both to test the soundness of the overall approach to mapping smart VCs into graph transformation systems and to experiment with model checking to analyse different types of properties. In particular, we analysed safety properties deriving from constraints declared in the class diagram, such as key properties for DoodleInvite (doodleName, organizer, voter are jointly unique), Doodle (name, organizer are jointly unique), and VotingSlot (option is unique), uniqueness of associations (voters represents a list with unique entries). From the logic of the problem domain we derive requirements such as: A party can vote at most once for each voting slot.

We formalised these constraints as property rules (without effect) expressing the forbidden patterns in their precondition. They are shown in Fig. 18.

We verified them in Groove as an LTL formula G !propNotDoodleInviteKey & !propNotDoodleKey & !propNotVotingSlotKey & !propNotUniqueVoters & !propNotUniqueVote. In addition we checked the lifeness property, that it is always possible to reach a state where cast vote is not enabled any more (because all invited parties have voted for all possible voting slots), in LTL GF !execCastVote.

Checks were executed on a state space of 3379 states and 22948 transitions generated from a start graph with a single Doodle contract, three parties and two voting slots while disabling the AddVoter and RemoveVoter. This shows that model checking is feasible on graph transformation models derived from visual smart contracts, with many of the properties defined directly by the constraints in the class diagram and other safety and lifeness properties expressing suitable requirements from the problem domain.

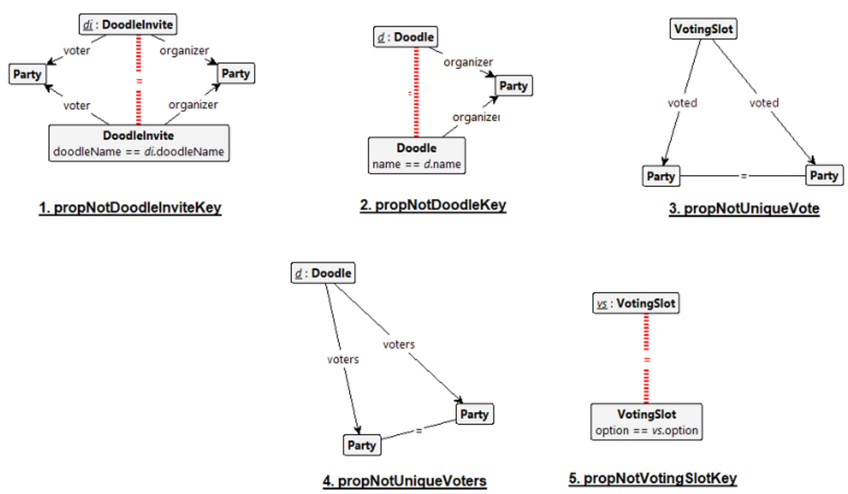


Fig. 18: Property Rules

References

1. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using groove. International journal on software tools for technology transfer 14(1), 15–40 (2012), https://doi.org/10.1007/s10009-011-0186-x
2. Heckel, R., Taentzer, G.: Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering. Springer (2020), https://doi.org/10.1007/978-3-030-43916-3
3. Kirschner, E.: A doodle in DAML - Part 1. Medium.com (October 2020), https://entzik.medium.com/a-doodle-in-daml-part-1-d2ef18bbf7e8
4. Kirschner, E.: A doodle in DAML - Part 2. Medium.com (November 2020), https://entzik.medium.com/a-doodle-in-daml-part-2-910614d94c62
5. Kirschner, E.: Github (11 2020), https://github.com/entzik/daml-examples/blob/master/doodle/daml/Com/Thekirschners/Daml/Doodle.daml