Declarative Smart Contract Testing by Domain Experts

Kevin J. Purnell[1], Rolf Schwitter[2]

[1]School of Computing, Macquarie University, Balaclava Rd, Macquarie Park, NSW, 2109, Australia
[2]School of Computing, Macquarie University, Balaclava Rd, Macquarie Park, NSW, 2109, Australia

Abstract

We present a novel approach to testing Answer Set Programs (ASPs) in the context of a system designed to enable a domain expert to write, test and deploy legal smart contracts. Common practice is to use compiled imperative languages to write smart contracts which limits what can be achieved, and provides a clear opportunity for an approach that empowers domain experts. Our system supports the construction of declarative smart contracts by domain experts with the help of a smart user interface that communicates visually and verbally using domain expert level concepts. It captures the ontology and legal logic of a legal document in a model automatically constructed as an ASP program. This paper discusses a complementary approach to testing, achieved by structuring ASP rules and splitting testing into model validation and program verification. Holding ontology information about the application domain allows the approach to be highly automated, so that we achieve automatic discovery of all hypothetical scenarios and exhaustive testing for each rule. Our approach places the domain expert in a tight learning loop where the behaviour of each rule scenario can be understood from the visual and verbal feedback, and rule corrections can be made immediately as required.

Keywords

Answer Set Programming, Declarative Language, Legal Logic, Modelling, Ontology, Smart Contract, Validation, Verbalisation, Verification, Visualisation

1. Introduction

The Smart Document Editor (SDE) is a system designed to enable a domain expert to write, test and deploy legal smart contracts on an Ethereum blockchain [1]. It uses a smart user interface (UI) and five processes to extend a traditional paper format legal document into a smart contract where the legal logic is represented as an auto-generated Answer Set Program (ASP). The five process steps are: 1) ontology discovery; 2) logic modelling; 3) model validation; 4) smart contract creation; and, 5) program verification. This paper describes an approach to exhaustive testing (steps 3 and 5) by domain experts that complements the ontology discovery and logic modelling functionality previously developed [2, 3, 4]. It illustrates features using the 'Will and Testament' use case from the preceding papers. A declarative program describes what is required [5], and the purely declarative language we use, ASP, lends itself to a model-based specification methodology that is executable [6]. Use of model based testing processes (model

RuleML+RR'22: 16th International Rule Challenge and 6th Doctoral Consortium, September 26–28, 2022, Virtual
kevin.purnell@hdr.mq.edu.au (K. J. Purnell); rolf.schwitter@mq.edu.au (R. Schwitter)
0000-0001-6674-7332 (K. J. Purnell); 0000-0001-8998-7005 (R. Schwitter)

validation and program verification) with an executable model allows testing to be performed as a user-mediated simulation.

Model validation answers the question: "Are we building the correct product?" [7]. For software models, model validation is the process by which the domain expert who requested the model, checks if the model correctly reproduces the features and behaviours of the artefacts being modelled. In the context of our SDE editor, the requesting domain expert is the user; so a process that displays to this user all the features and exercises all the intended behaviours, satisfies the definition of model validation. Another requirement is to prove that the model operates correctly for all possible instantiations. This is straightforward for text data which is held as strings, but numeric data requires that exceptions like 'divide by zero' return 'undefined' [8], so for model validation we auto-populate these input fields with the type name and standard numeric values respectively. The SDE editor provides feedback to the user visually and via verbalisation, a feature made easier to achieve because elements of the ASP program have a formal correspondence with graphics and text. The representations used by the SDE editor also hold enough information to support a high level of automation which allows model validation to be reduced to a user-mediated simulation with visual and verbal feedback. Our approach forces rule-by-rule creation and testing and it limits the possible outcomes per rule to a manageable number, making exhaustive testing practical.

Program verification answers the question: "Are we building the product correctly?" [7]. In the case of the SDE editor, the declarative program has been generated by the domain expert who created the requirements specification. Splitting the testing effort into model validation and program verification reduces the effort required to achieve a form of program verification which is the equivalent of combining system testing and user acceptance testing for imperative languages. It can be assumed that the model validation process provides a program shown to work for test data and shown to be able to handle all possible instantiations including no data. This means that the state space for program verification relative to model validation is reduced as some input fields are fixed by the addition of facts (see Section 4). Program verification for the SDE editor looks similar to model validation, except that the scope is now the ASP program rather than one rule. This all-up test [9] is the most complex aspect of the SDE editor, and likely constrains what can be programmed with this approach.

1.1. Related Research

Literature describing approaches to testing ASP programs usually mention test cases [10, 11, 12], coverage, assertions and pre- and post-conditions inserted as meta information comments in ASP code [13, 14], often in relation to the integrated development environment ASPIDE [11, 12, 13, 15]. A recent paper refines these ideas and presents a programming environment "ASP-WIDE" that supports Test-Driven Development (TDD) [14]. These approaches aim for effective coverage of the state space rather than attempting exhaustive coverage. In contrast, the SDE system aims for exhaustive coverage by using a subset of ASP and limiting the complexity of each rule so that testing all possible outcomes is practical. Program expressability is instead achieved by stacking rules upon each other (see Section 2.2), a form of modularisation that suits ASP. This 'divide and conquer' strategy appears in the literature [16, 17], as does work on testing the small-scope hypothesis [18]; however, no similar focus on achieving exhaustive

testing through restrictions on coding style have surfaced. Furthermore, a recent paper by Kholkar [19] mentions many of the techniques we use but lists automatic generation of model testing data as future work. Also note that the modularity achieved with `#program, #include` [20] is not related to rule complexity.

2. Overview of the Smart Document Editor (SDE)

Traditionally, most legal documents were paper forms filled out and signed by hand, and this format is retained in the electronic documents used by word processors and document automation systems [21]. The SDE editor adds the ability to first understand the closed world of the legal document, then model and test the embedded logic. The SDE editor takes an existing legal document and allows it to be incrementally developed into a smart contract [4]. This workflow consists of a fixed sequence of steps that can be split into two main phases: 1) machine understanding of the legal document; and, 2) smart contract creation. Machine understanding of the legal document involves three steps: (i) modelling the ontology (ontology discovery); (ii) modelling the legal logic (logic modelling); and, (iii) validating that this model matches the user's understanding (model validation). Smart contract creation involves two steps: (i) entering actual data (contract creation); and, (ii) testing that the output is what is expected (program verification). The final product is the initial legal document completed with actual information and an embedded ASP program. Contracts that are legally binding and both human- and machine-readable are known as Ricardian contracts [22]. This paper discusses the techniques used for 'model validation', and introduces the techniques used for 'smart contract creation' and the subsequent 'program verification' step.

2.1. SDE Declarative Representations

The features achieved by the SDE editor rely on the ASP representations used, and these are structured from a subset of ASP and an adjunct grammar [3]. ASP programs are auto-assembled from only two components; ASPspec and ASPexpression (see Fig. 1) which are auto-generated under user guidance. ASPspec represents the ontology of the document and consists of a set of ASP-like strings, while ASPexpression is the set of executable ASP built-in and aggregate atoms. ASPspec is used to auto-generate ASPatomic (unground), ASPfact (ground) and ASPtest (ground) atoms (see Fig. 2). ASP rules (ASPrule) and rule heads (ASPanswer) are also auto-generated.

The ASP Subset used by the SDE editor: The ASP subset conforms to the definition of a "extended logic program" [23], where rule heads are restricted to classical literals, and terms (SDEterm) are restricted to safe functional terms with two parameters [24, 25]. The functor `<termId>` specifies the purpose of the term, while the parameters are data type (placeholderId) and instance (data).

`(1.1) SDEterm: <termId>(<placeholderId>,<data>)`

The SDE editor allows three literal types (SDEliteral): 1) ASPatomic are representations; 2) ASPanswer are rule generated; and, 3) ASPexpression are built-ins and aggregations (Fig. 1).

`(1.2) SDEliteral: ["not "]["-"]<predicateId>(<one or more SDEterm>)`

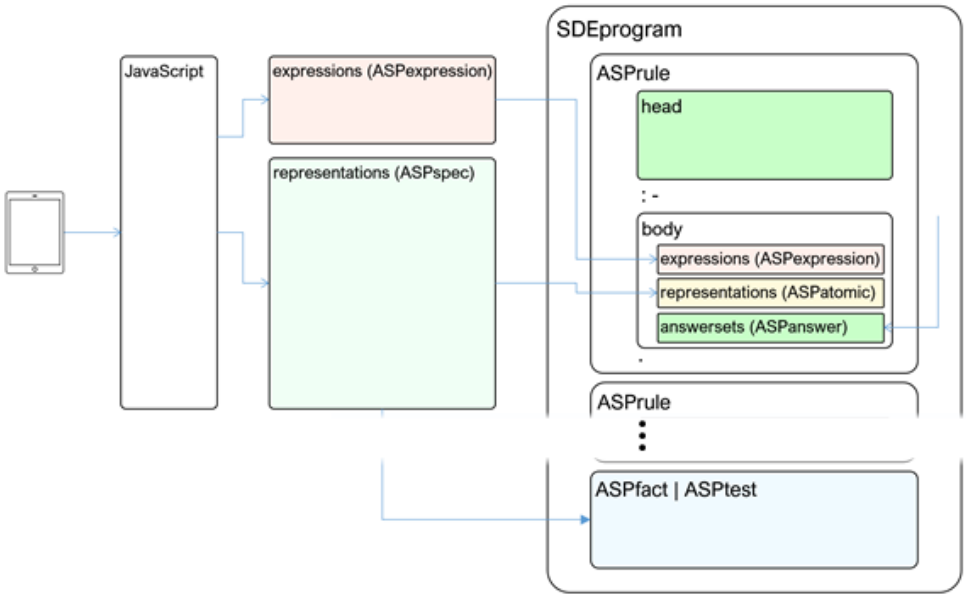

Figure 1: The Structure of an ASP program in the SDE editor.

The Adjunct Grammar: Artefacts being modelled are represented by literals with predicateId "represent" and auto-generated from 'ASPspec' (see Fig. 2).

Representations (**ASPatomic**) for artefacts 'thing', 'association', and 'event' are shown below in EBNF-like syntax (text enclosed by < > are SDEterms):

```
(2.1) represent(<thingId>,<thingKey>[,<zero or more properties>])
(2.2) represent(<assocId>,<attribute>,<ofKey>,<toKey>[,<zero or more properties>])
(2.3) represent(<eventId>,<timeKey>[,<agentKey>],<expnrKey>[,<modfrKey>][,<zero or mor...)
```

The structure of representations assists verbalisation; for example, the representation for an 'event' holds agent and experiencer information corresponding to subject and object in an English active sentence, while representations for 'thing' and 'association' verbalise as "there exists" (see Fig. 5). Other representations are likely required for different domains (e.g., fluents).

Rule heads (**ASPanswer**) are positive literals with predicateId "rulehead":

```
(2.4) rulehead(<rulehId>,<object>,if,<one or more SDEterm>)
```

The rule head has meaning relative to some object (the second term), usually the document being processed; for example, the object of rule 'is_witnessed' is the 'Will'. The 'if' marks the start of SDEterms copied from body literals and which are grouped by 'and'. The delimiters 'if' and 'and' aid verbalisation.

Expressions (**ASPexpression**) are also literals:

```
(2.5) <zero or more ASPexpression>
```

Expressions are always attached to an ASP rule, so they do not need identifiers.

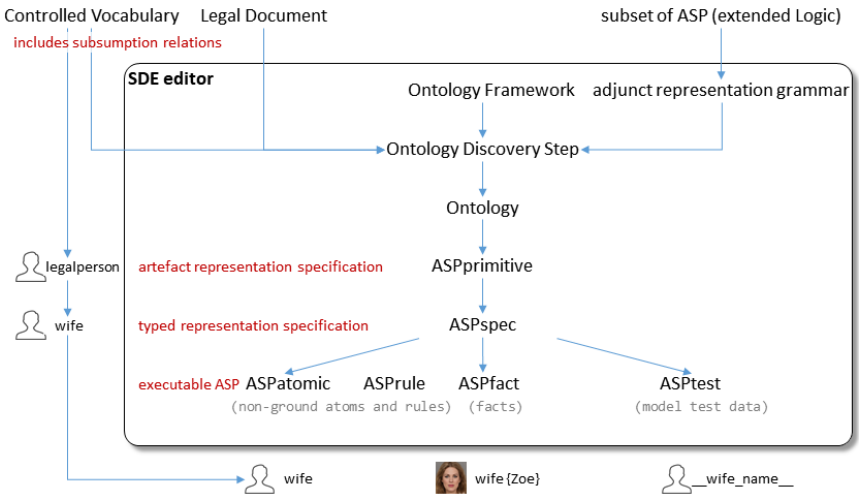

Figure 2: Derivation of ASP, Visual, and Text Representations in the SDE editor.

2.2. Discussion of Representation Design Choices

The design of the representations used by the SDE editor is driven by four factors: 1) ease of verbalisation; 2) understandability of auto-generated ASP code during construction of a proof-of-concept system; 3) alignment to artefacts and concepts that can be understood by domain experts; and, 4) achieving a practical exhaustive testing regime. If solver efficiency is required, techniques for minimising ASP programs, like moving string data to keyed tables and using meta-programming techniques may assist. Practical exhaustive testing requires limitations to the complexity of individual rules, but this does not limit stacking rule upon rule and transmitting information between rules via answer set atoms. For example; ten rules (m) with three literals (n) each (plus an extra literal for the linking answer set atom), covers $(2^n)^m$ or 8^{10} input combinations, yet exhaustively testing this large state space is not intractable.

3. Overview of SDE Model Validation

An SDE answer set program consists of rules and facts (see Fig. 1). Testing this program can be achieved by modifying facts, which for the SDE editor are also representations of: 1) things or people; 2) events that can happen (often to things or people); and, 3) relationships between things, people and events. These facts (ASPfact) have the same form as rule literals (ASPatomic) and are generated from the same representation specification (ASPspec) (see Fig. 2). This allows model test data and test scenarios to be automatically generated by reading rule body literals. The SDE editor uses a human-in-the-loop (mediated) simulation [26, 27] which allows the domain expert user to observe all the features and behaviours of the model with a minimum of effort, a good fit for the SDE editor's intended audience. Techniques used to achieve this simulation approach are outlined in the following sections.

3.1. Principles underlying SDE Model Validation

To be practical for domain experts, the user interface must communicate legal rather than technical concepts, and this implies the avoidance of a traditional testing methodology. The approach we take can be summarised by the following five principles: 1) communicate with the domain expert user by aligning the language and representations used to concepts understood by a domain expert; 2) decompose complex objects like rules where possible; 3) automate tasks like scenario generation where possible; 4) feedback to users via both visual and verbal methods; and, 5) create and test in a tight learning feedback loop.

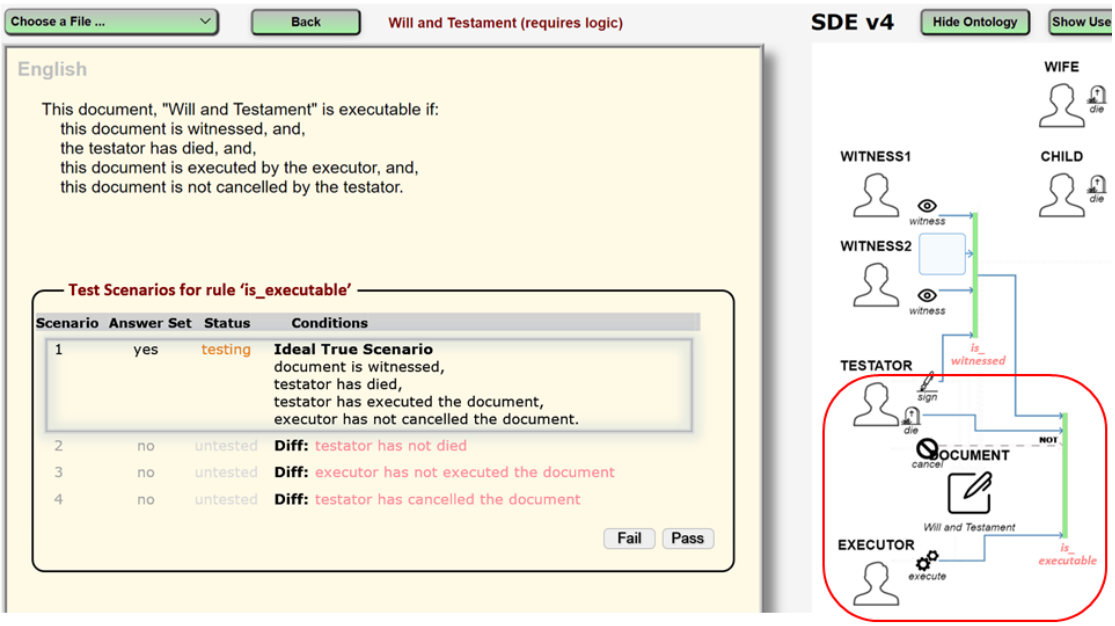

Figure 3: SDE Model Validation showing the 'true' Scenario for the Rule 'is_executable'.

3.2. The Model Validation Process

The SDE editor automatically assembles ASP rules under user guidance in a build-then-test rule-by-rule process such that the collection of these rules becomes the ASP program. SDE editor rules use three types of literal in a rule body; ASPatomic, ASPanswer and ASPexpression (see Fig. 1). Because facts always match the structure of ASPatomic, model validation test data (ASPtest) can be auto-generated. To instantiate ASPtest, the type is inserted into the data field for string data, and predefined numbers are inserted for numeric data, so that the ASPtest generated represent generalised facts. The system automatically develops model testing scenarios for each rule by simply reading the body literals of the rule, and executes these scenarios by manipulating the ASPtest (blue coloured box in Fig. 1). The process is: 1) auto-generate singleton facts; 2) auto-generate the exhaustive set of scenarios; 3) automatically step the user though each

scenario providing visual and verbal feedback; and, 4) allow user control via the selection of either the "Pass" or the "Fail" button (see Fig. 3).

3.3. User Feedback

This section illustrates our approach to user feedback during model validation using one of our use cases, the "Will and Testament". The three relevant components are displayed in Fig. 3: 1) the lower left pane shows the scenario presentation manager with the currently executing scenario highlighted (grey box); 2) the upper left pane shows the verbalisation of the relevant answer set atoms; and, 3) the lower right pane shows a visualisation of the rule (red outline).

1. Scenario Presentation Manager: The scenario presentation manager verbalises all test scenarios in a scrolling window with the true scenario displayed first. For subsequent scenarios, our approach changes only one variable at a time relative to the true scenario which is enough to stop the rule from producing an answer set. The changed variable is verbalised as a difference to the true solution, which is indicated by the prefix "Diff:". The user can move to the next scenario by pressing either the "Pass" or "Fail" button.

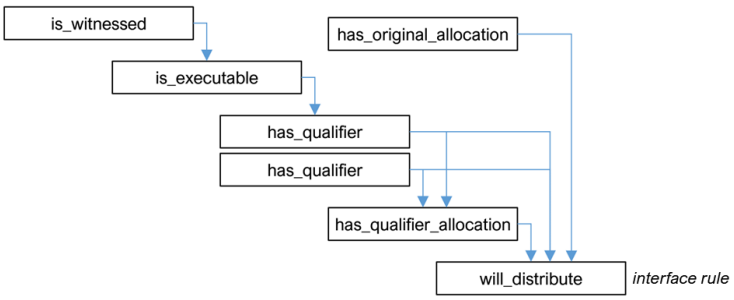

Figure 4: The main Rule Tree of the "Will and Testament" Use Case.

2. Verbalisation: Scenarios which produce answer sets verbalise the relevant atoms of the answer set to communicate that the rule has triggered and why it triggered. Representations lend themselves to verbalisation (see Section 2.1) and four formal correspondences have been achieved: 1) an icon corresponds to a text term which corresponds to an ASP representation; 2) a photo corresponds to an instance name which corresponds to an ASP fact; 3) parts of the graphic for a rule correspond to an English clause which corresponds to an ASP literal; and, 4) the graphic for a rule corresponds to a complete English conditional sentence which corresponds to an ASP rule. These correspondences allow both scenario and answer set verbalisations (see Fig. 3). Furthermore, because the SDE editor holds process state, it is able to verbalise using the correct tense for a particular stage of processing.

3. Dynamic Visualisation of the Rule: The right pane of Fig. 3 displays the visual representation of the rule 'is_executable'. The rule head is represented by the green vertical bar with 'is_executable' in small crimson text below. The literals that exist are shown as blue arrows, while those that do not exist are shown as dotted grey arrows. The arrows originating from small icons attached to larger icons shown in Fig. 3 are 'events'. The set of scenarios for 'is_executable' (Fig. 3) contains only four members because this rule has only four literals and no constraints

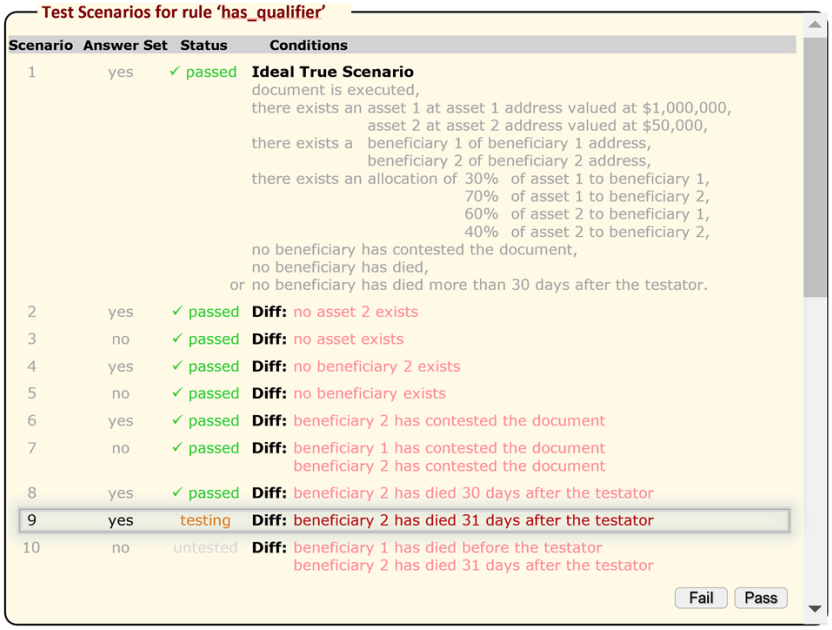

Figure 5: An example of a complex SDE editor rule and its model validation scenarios.

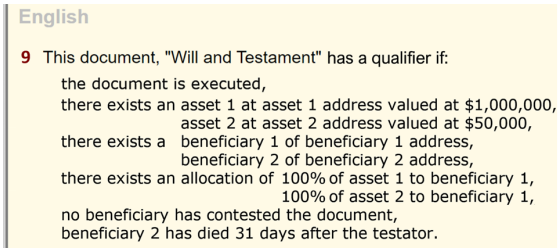

Figure 6: The verbalisation of scenario 9 of rule 'has qualifier'.

and ASP rules are conjunctions. In the scenario being tested, only the 'cancel' event has not happened (shown as a grey dashed line with text "NOT"), while the events that have happened are represented by blue arrows. This rule fires because all its conditions are satisfied, which is indicated by its rule head turning green, and a verbalisation being displayed in the top left pane of Fig. 3. To test the other scenarios listed in the bottom left pane of Fig. 3, ASPtest is manipulated to remove positive literals and add negative literals one at a time. In this example, these scenarios do not fire, so the verbalisation disappears and the rule head turns red. Finally, rules referenced in a following rule (via ASPanswer) are always set to their "true scenario" because they have already been tested.

4. Complex Rules: A more complex rule is 'has_qualifier' (see Fig. 5) which has six scenarios with answer sets, and is useful for illustrating the more complex features of the SDE editor model validation process. To aid orientation, the main rule tree of our use case, the 'Will and Testament' is shown in Fig. 4. The rule 'has_qualifier' implements an inclusive disjunction

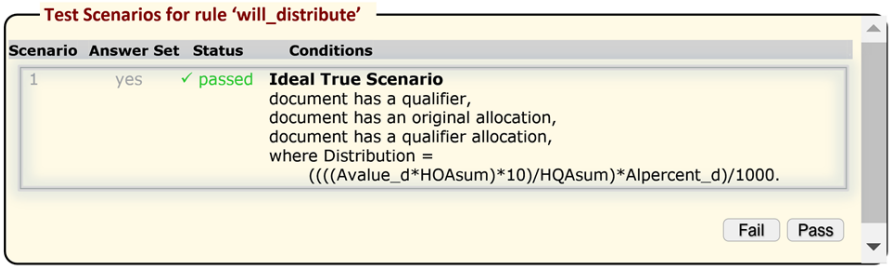

Figure 7: The interface rule 'will_distribute'

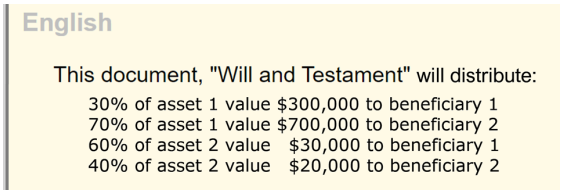

Figure 8: Verbalisation of the interface rule 'will_ distribute'

in the body so that it can identify both beneficiaries that are still alive and beneficiaries that outlasted the testator by at least 30 days, both of which qualify for a distribution. The method we use always lists the "true scenario" first and by definition this scenario will always have an answer set. The concept of "true scenario" is extended to "ideal true scenario" for rules that have multiple scenarios with answer sets. This distinction allows answer sets produced with all predicates valued at 'TRUE' to be differentiated from answer sets produced where some facts do not exist. Automatic generation of the "ideal true scenario" is achieved by reading all body literals (representations first and then expressions) of the rule and retrieving the matching ASPtest facts. Each grey line of text (see Fig. 5) of the "ideal true scenario" is a verbalisation of a body literal, elaborated from the matching fact. The answer set of the scenario with focus in Fig. 5 verbalises as shown in Fig. 6.

5. Interface Rules: Our design requires the interface rule, 'will_distribute' (see Fig. 7) to provide directions to the blockchain via an interface (not implemented) which generates Ethereum transactions. The verbalisation provides insight into both the minimum set we use to populate associations and the predefined numbers inserted for numeric data when ASPtest facts are auto-generated. The minimum set for an association is 2 of each foreign key being associated; so in this case 2 beneficiaries have 2 assets allocated between them. The allocation percentages are set at 30% and 70% for the first asset and 60% and 40% for the second asset, while the value of the first asset is set to $1,000,000 and $50,000 for the second asset. These numbers are hard coded into the system and have been chosen to provide both generality and confirmation at a glance for the domain expert, while also exercising the model adequately. The rule 'will_distribute' has only one scenario because the rule body uses only ASPanswer and ASPexpression literals, and the ASPexpression is an assignment (see subsection 3.4).

3.4. Body Literals in Detail

This section details how the components (literals) in the rule body of an SDE editor rule, are processed. These are: 1) representations of artefacts (ASPatomic); 2) answer set derived literals (ASPanswer); 3) comparison expressions; 4) assignment expressions; and, 5) aggregate expressions. These last three literals (ASPexpression) are similar but behave differently as described below.

1. ASPatomic: Artefacts are modelled by representations which are literals (see Section 2.1) and so are predicates. This means that the value of the rule containing only representations in the body can be calculated via a truth table and has at most 2^n (the power set) input combinations. However; ASP rule bodies are conjunctions, so all literals have to evaluate to 'true' for the rule to fire, which reduces the number of tests required to the number of literals. The method we use removes positive literals and adds negative literals one at a time.

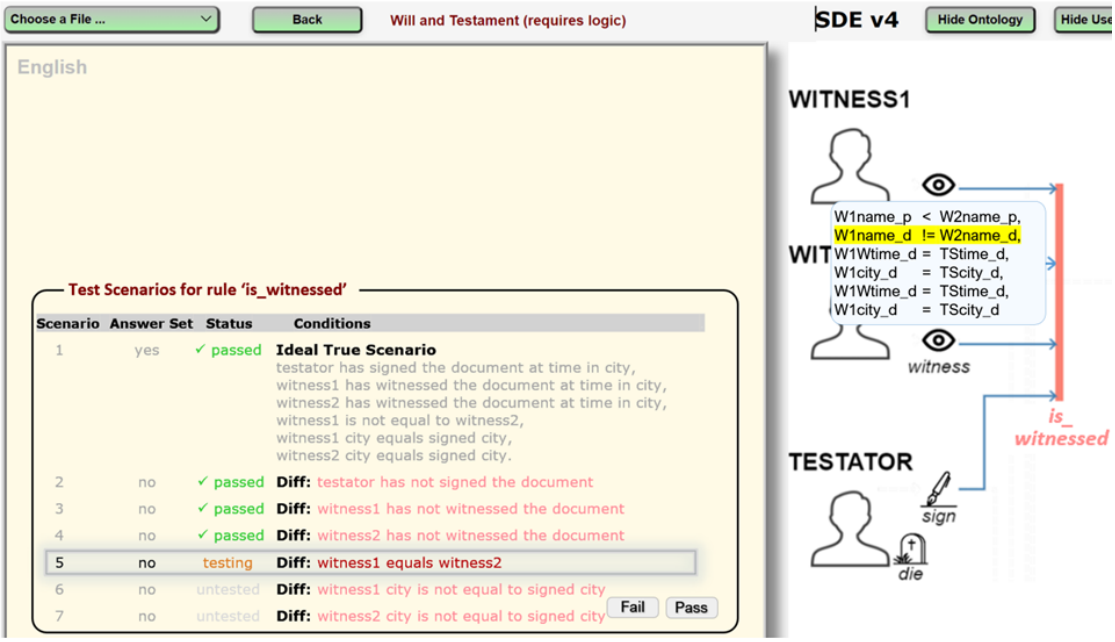

Figure 9: SDE editor screen showing a condition (enlarged) being tested.

2. ASPanswer: ASPanswer are literals derived from answer sets of non-interface rules. Because of the way the SDE editor stacks rules upon each other and requires rules to be validated immediately after they are created, ASPanswer encountered as literals in a rule body have already been tested and can be ignored by setting the test data to the "ideal true scenario".

3. ASPexpression - Comparison: Testing relational operators follows a similar approach, except that automatic scenario generation now modifies rather than adds or removes the ASPtest containing the compared variables. Three scenarios are required to exhaustively test a relational operator; for example, 1) $x > y$; 2) $x = y$; and, 3) $x < y$. The other operators are tested similarly

and comparison predicates can be both weakly and strongly negated.

4. ASPexpression - Assignment: Common ASP terminology refers to expressions as built-in atoms. In built-in atoms, the equal symbol ("=") has two meanings (see [20] page 26) that depend on the existence of the left variable. If the left variable exists, the equal symbol is treated as a relational operator, while if it does not exist, the variable is created and assigned the value calculated for the right side of the expression (both string and numeric). The SDE editor uses predefined numbers in ASPtest to provide the user with an indication of how an arithmetic formula behaves, but leaves most arithmetic testing to the program verification step. The SDE editor does not allow the use of assigned variables within the generating rule because this complicates testing, and assignments cannot be weakly negated.

5. ASPexpression - Aggregate: Aggregates are treated in a similar way to assignments; that is, no specific testing is performed because aggregates manipulate actual data. The predefined numbers in ASPtest creates aggregate values that indicate aggregate behaviour. Aggregates can be both weakly and strongly negated, and the use of aggregates; like assignments, is restricted to the following rule to avoid complicating testing.

4. Introduction to SDE Smart Contract Creation

The SDE editor creates smart contracts by adding facts to the validated logic model. This section provides examples of adding facts (ASPfact) and illustrates the use of instantiation placeholders (IPH) placed by the earlier ontology step to bind input data [2, 3].
Examples of representation specifications (ASPspec) at the model validation step are:

```
(3.1) represent(thingId(document,""),
     thingKey(__document_name__,"Will and Testament"),
     version(__document_version__,"v1.0"), ...)
(3.2) represent(thingId(legalperson,executor),
     thingKey(__executor_name__,""),
     address(__executor_address__,""), ...)
```

An example of an event fact (ASPfact) (derived from (3.2)) after the contract creation step is:

```
(3.3) represent(thingId(legalperson,executor),
     thingKey(__executor_name__,"James Stewart"),
     address(__executor_address__,"Macquarie Road, Springwood"), ...).
```

Some event representations are not created by the contract creation step. Example (3.4) is an event ASPfact generated by the interface between Ethereum and the ASP program by mining:

```
(3.4) represent(eventId(execute,executor),time(__executor_execute_time__,44599),[1]
     agent(__executor_name__,"James Stewart"),
     experiencer(__document_name__,"Will and Testament"), ...).
```

The Ethereum transaction which triggers mining and generates (3.4) is placed in the transaction pool by the executor (not implemented).

[1] Date is days since 1/1/1900

5. Introduction to SDE Program Verification

This section provides a short description of SDE program verification, as the ideas used are similar to those used by model validation; in particular, model validation of the interface rule (see Fig. 4). This process also automatically generates hypothetical scenarios by identifying which literals and variables change the output; however, the scope is now the entire ASP program.

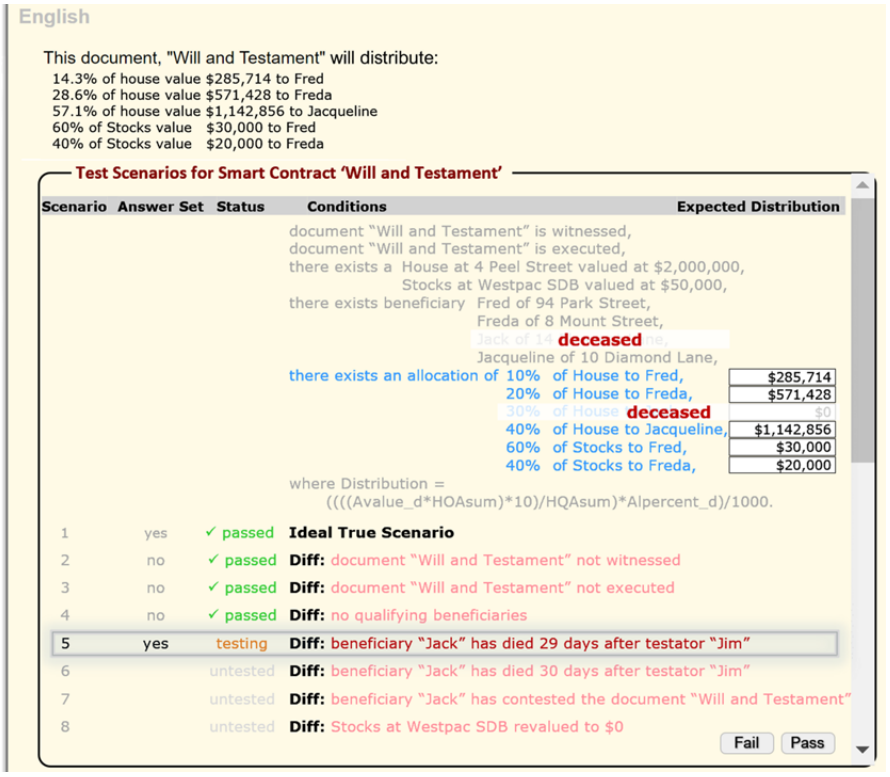

Figure 10: Scenario Manager and Verbalisation for Program Verification.

Changes which affect Smart Contract Outcomes: By the program verification stage, the logic model has been validated and some facts added, which fixes the value of some variables and reduces the state space. After this point, only events can change program outcomes, and these are: 1) the death of the testator, 2) the execution of the 'Will'; 3) the death of a beneficiary; 4) a beneficiary contests the 'Will'; and, 5) a revaluation of an asset to zero via transaction or oracle.

Aspects previously tested by Model Validation: Model validation tests that the system behaves as expected for both zero and one or more artefacts of a given type, and similarly for numerical values of zero, undefined, and non-zero.

Verbalising all Conditions affecting Smart Contract Outcomes: Because the SDE editor is targeted at domain experts, it is desirable that a complete verbalisation of the ASP program be displayed, even for events that have already happened (e.g., witnessing). Some simplifying techniques make this practical; for example, for rules with one answer set, the verbalisation of

all literals can be replaced one verbalisation of the rule, a form of abstraction. The interface rule 'will_distribute' can now be verbalised in an abbreviated form starting with "document is witnessed", and "document is executed". Rules with more than one answer set still require verbalisation at the literal level. These can be seen for rule 'has_qualifier' in Fig. 5. To further limit verbalisation length, we show events as overlays; for example, "deceased" is overlayed on beneficiary "Jack" (see Fig. 10). In summary: 1) for rules with one answer set, abstract with the rule; 2) for rules with more than one answer set, verbalise the non event literals and overlay the event literals; 3) for calculated variables like 'Distribution' (see Fig. 7), ask for test case data (white input fields in Fig. 10). Test case data is required for every scenario that has numeric calculations. This requires the domain expert to prepare independent calculations before conducting the verification. These input fields require manual calculation and input for each scenario (see Fig. 10 ... death of 'Jack').

Running the User-Mediated Simulation: The system displays each scenario in sequence pausing for either a "Pass" or "Fail" to be entered by the domain expert. For a scenario to pass, the calculated allocations must match the test case data entered. Because of previous testing during model validation and use of simplifying techniques, the verbalisation of the program conditions and the list of scenarios generated are both manageable. Again, the SDE editor achieves program verification in a way that is intuitive and usable by domain experts with the visual and verbal communication channels working synergistically.

6. Results, Usability and Future Work

The primary objective of this research was to identify a practical testing approach and user interface that complemented the understandability, usability and look of the 'ontology discovery' and 'logic modelling' process steps [4] for domain experts. The key techniques required for this achievement are demonstrated by a proof-of-concept system [28] and evidence of usability is provided by the following description. The first testing step (model validation) allows the user to exhaustively test individual rules by ctrl+click on the rule head. The system computes and verbalises all test scenarios then moves the user through each scenario one at a time, visually displaying how the rule works and verbalising the applicable atoms of the answer set if the rule fires. Progress to the next scenario requires the user to mark the scenario "Pass" or "Fail". Progress to the next rule requires failed scenarios to be corrected. The second testing step (program verification) functions similarly. The system computes and verbalises all test scenarios for the entire program, using the simplifying techniques discussed in Section 5. If the interface rule has numeric calculations, the user is asked to input test cases, but otherwise the process is similar to model validation. The system will only allow smart contract deployment if no "Fail" flags exist for the program. Both validation and verification are simple to use and require a minimum of training.

Areas requiring further research include: 1) investigate the range of use cases that can be modelled and tested; 2) investigate the boundaries of program expressability; 3) conduct a user study to determine usability by domain experts; and, 4) further develop the proof-of-concept system.

7. Conclusion

This paper describes research that has the objective of devising a practical approach to the testing of ASP programs in the context of declarative smart contracts that can be created, tested and deployed by domain experts. We have identified and described a practical approach applicable to the subset of ASP programs that can be generated by the SDE editor, namely the modelling of legal logic used by amenable legal contracts and other legal/normative documents. The auto-generation of ASP code by the SDE editor facilitates systematic automated testing approaches; and, adhering to model based testing methods allows the split into two testing phases to be exploited to lower complexity at each phase. This allows practical exhaustive testing without overly restricting what can be expressed by the program. To our knowledge, no other ASP testing approach achieves exhaustive testing in a practical way.

References

[1] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, Ethereum Project Yellow Paper (2014). URL: https://gavwood.com/paper.pdf.

[2] K. Purnell, R. Schwitter, Towards Declarative Smart Contracts, in: Proceedings of SDLT, 2019, pp. 18–21. URL: https://symposium-dlt.org/4th/SDLT2019-FinalProceedings.pdf.

[3] K. Purnell, R. Schwitter, User-defined smart contracts using answer set programming, AI 2021: Advances in AI, LNCS 13151 (2021) 291–303. doi:10.1007/978-3-030-97546-3_24.

[4] K. Purnell, R. Schwitter, User-Guided Machine Understanding of Legal Documents, in: Proceedings of JURISIN, 2021. URL: https://nuss.nagoya-u.ac.jp/s/ZtFe8PWatGA77Eo.

[5] J. W. Lloyd, Practical advantages of declarative programming, in: GULP-PRODE, 1994. URL: https://www.programmazionelogica.it/wp-content/uploads/2015/12/GP1994-I-000-031.pdf.

[6] G. Brewka, T. Eiter, M. Truszczynski, Answer Set Programming at a Glance, Communications of the ACM 54 (2011) 92–103. doi:10.1145/2043174.2043195.

[7] B. Boehm, Verifying and Validating Software Requirements and Design Specifications, IEEE Software Journal 1 (1984) 75–88. doi:10.1109/MS.1984.233702.

[8] V. Lifschitz, What is answer set programming?, in: Proceedings of AAAI, 2008, pp. 1594–1597. URL: https://dl.acm.org/doi/10.5555/1620270.1620340.

[9] M. Sharpe, Saturn and All-up Flight Testing: Saturn History Project, 1974. URL: http://heroicrelics.org/info/all-up/all-up-flight-testing.html.

[10] T. Janhunen, I. Niemelä, J. Oetsch, J. Pührer, H. Tompits, On testing answer-set programs, in: Proceedings of ECAI, 2010, p. 951–956. doi:10.3233/978-1-60750-606-5-951.

[11] O. Febbraro, N. Leone, K. Reale, F. Ricca, Unit Testing in ASPIDE, ArXiv (2011). doi:10.48550/arXiv.1108.5434.

[12] O. Febbraro, K. Reale, F. Ricca, ASPIDE: Integrated Development Environment for Answer Set Programming, in: Proceedings of LPNMR, 2011. doi:10.1007/978-3-642-20895-9_37.

[13] M. D. Vos, D. Kisa, J. Oetsch, J. Puhrer, H. Tompits, Annotating answer-set programs in lana, Theory and Practice of Logic Programming 12 (2012) 619–637. doi:10.1017/S1471068418000327.

[14] G. Amendola, T. Berei, F. Ricca, Testing in ASP: Revisited language and programming environment, in: Proceedings of JELIA, 2021, pp. 362–376. doi:10.1007/978-3-030-75775-5_24.

[15] C. Kloimüllner, J. Oetsch, J. Pührer, H. Tompits, Kara: A system for visualising and visual editing of interpretations for answer-set programs, in: Proceedings of INAP 2011 and WLP, 2011, pp. 325–344. doi:10.1007/978-3-642-41524-1_20.

[16] P. Cabalar, J. Fandinno, Y. Lierler, Modular Answer Set Programming as a Formal Specification Language, TPLP 20 (2020) 767–782. doi:10.1017/S1471068420000265.
[17] D. Desovski, B. Cukic, A Component-Based Approach to Verification and Validation of Formal Software Models, volume 4615, 2007. doi:10.1007/978-3-540-74035-3_5.
[18] J. Oetsch, M. Prischink, J. Pührer, M. Schwengerer, H. Tompits, On the small-scope hypothesis for testing answer-set programs, 13th International Conference on the Principles of Knowledge Representation and Reasoning, KR 2012 (2012) 43–53. doi:10.5555/3031843.3031849.
[19] D. Kholkar, D. Mulpuru, V. Kulkarni, Balancing model usability and verifiability with SBVR and answer set programming, in: Proceedings of MODELS 2018 Workshops (MoDeVVa), CEUR Workshop Proceedings, 2018. URL: http://ceur-ws.org/Vol-2245/modevva_paper_3.pdf.
[20] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, P. Wanko, Potassco User Guide 2.2.0, 2019. URL: https://github.com/potassco/guide/releases/.
[21] T. Reuters, HighQ Document Automation, 2022. URL: https://legal.thomsonreuters.com/en/products/highq/document-automation.
[22] I. Grigg, The ricardian contract, in: Proceedings of First IEEE International Workshop on on Electronic Contracting, 2004, 2004, pp. 25 – 31. doi:10.1109/WEC.2004.1319505.
[23] T. Eiter, G. Ianni, T. Krennwallner, Answer set programming: A primer, Lecture Notes in Computer Science 5689 (2009) 40–110. doi:10.1007/978-3-642-03754-2_2.
[24] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, Asp-core-2 input language format, Theory and Practice of Logic Programming 20 (2019) 294–309. doi:10.1017/S1471068419000450.
[25] Y. Lierler, V. Lifschitz, One more decidable class of finitely ground programs, in: P. M. Hill, D. S. Warren (Eds.), Proceedings of Logic Programming. ICLP 2009, Springer Berlin Heidelberg, 2009, pp. 489–493. doi:10.1007/978-3-642-02846-5_40.
[26] I. Salik, J. Ashurst, Closed Loop Communication Training in Medical Simulation, StatPearls Publishing, 2021. URL: www.statpearls.com/articlelibrary/viewarticle/63796/.
[27] G. E. Paul, Modeling and Simulation of Human Systems, John Wiley & Sons, Ltd, 2021, pp. 704–735. doi:10.1002/9781119636113.ch27.
[28] K. Purnell, Smart Document Editor Proof-Of-Concept, 2022. URL: http://130.56.246.229/.