Smart Tests for a Smart Contract Language

Miguel Valido
NOVA School of Science and Technology, Caparica, Portugal

António Ravara
NOVA School of Science and Technology, Caparica, Portugal

Abstract

Smart contracts are high-stakes software: their immutable, publicly accessible, code may govern assets worth millions, meaning that even minor defects can have severe consequences. The most used techniques to ensure smart contract correctness are testing and formal verification. Testing is almost always employed but is often restricted to unit tests (which often miss edge cases) and has limited coverage, while formal verification can provide strong guarantees but is often costly and complex to apply, demanding substantial time and expertise. Property-based testing bridges this gap by exploring large input spaces and shrinking failures to minimal counterexamples, helping uncover defects early in development. Formal verification can be left to critical features once testing has filtered out common issues.

To add to the challenges smart contract developers face, most languages used were not designed with safety and security guarantees built-in. Daml is a smart contract language designed with correctness in mind, featuring a strong static type system, functional programming paradigms, and built-in abstractions for common smart contract patterns. However, Daml currently lacks support for property-based testing, limiting developers' ability to systematically explore input spaces and verify contract properties.

This paper introduces Hypothesis2Daml, an open-source library that brings property-based testing to the Daml ecosystem by connecting the Hypothesis testing framework with the Daml JSON API. Hypothesis2Daml enables developers to specify invariants, preconditions, and stateful workflows over realistic ledger interactions, while providing automatic input generation, shrinking, and isolation of ledger state between test cases. The approach is evaluated using a benchmark consisting of eight contracts, three Daml templates, and twenty-eight property-based tests covering happy paths, negative cases, and alternative interaction orders. The results show that property-based testing is feasible for Daml smart contracts, can systematically expose violated properties with minimal counterexamples, and supports effective debugging of realistic, stateful workflows.

2012 ACM Subject Classification Software and its engineering → Empirical software validation

Keywords and phrases Smart contracts, Blockchain, Daml, Property-based testing, Hypothesis

Digital Object Identifier 10.4230/OASIcs.FMBC.2026.5

Supplementary Material
Software (Source Code): `https://github.com/miguelvalido02/Hypothesis2Daml` [34]
archived at `swh:1:dir:42946bbde2f05418d3a606258a87246557211531`

Funding We acknowledge the support of NOVA LINCS (Grant no. UID/04516/2025) with the financial support of FCT.IP and EU Horizon Europe project TaRDIS (Grant Agreement no. 101093006).

Acknowledgements We would like to thank the anonymous reviewers for their constructive feedback.

1 Introduction

Testing the reliability of smart contracts is essential to making them secure against malicious actors. These contracts often manage and interact with protocols that hold millions of dollars, making them high-value targets for hackers. Once a smart contract is deployed, it becomes public on the network, allowing anyone to interact with it. Without thorough testing,

rigorous auditing, and ultimately the verification of the correctness of key functionalities, vulnerabilities in the code can result in catastrophic losses for both protocols and their users [21]. For example, the Ronin Network hack in March 2022 resulted in $625 million worth of Ether (cryptocurrency worth over $2000) and USDC (stablecoin worth $1) being stolen, marking the largest cryptocurrency hack in history. Similarly, the Poly Network exploit in August 2021 saw over $600 million stolen, exploiting a single vulnerability in its system. Even Binance, a prominent cryptocurrency exchanges, suffered a $569 million loss in October 2022 due to a bug in a smart contract on its BSC Token Hub. These recurrent situations underscore the financial and reputational risks associated with exploited vulnerabilities, highlighting the necessity of ensuring contracts are as secure and fault-proof as possible [17].

To mitigate these risks, developers typically invest in writing unit tests for individual contract functions and often extend testing to scenario-based approaches. Scenario testing involves creating multiple scenarios with varying inputs to evaluate the functionality of the contract as a whole. While these strategies validate individual components and their interactions, they are inherently limited. Since test cases are manually crafted, they leave room for unexplored edge cases, increasing the likelihood of undetected vulnerabilities and bugs [21]. This limitation is exacerbated by the fact that smart contracts often integrate with other protocols to form highly interconnected ecosystems. Such complexity can obscure edge cases and introduce subtle bugs that lead to failures, exploits, loss of funds, or application malfunctions. Traditional testing methods often fail to capture these complex interactions [21].

Property-based testing (see Appendix B) has emerged as a valuable methodology in software testing, particularly for its ability to identify edge cases that traditional example-based testing often overlooks. Unlike unit tests, which demonstrate the existence of errors only for specific cases, PBT validates overarching properties across a vast input space [33]. By exploring these broad ranges, PBT uncovers the edge cases that traditional example-based testing overlooks, providing significantly higher guarantees of correctness [18]. Consequently, the provision of property-based testing capabilities is now considered a fundamental requirement for ensuring the reliability of smart contract languages [7].

Despite the many property-based testing tools and libraries available for Solidity [25, 16, 11], the most popular language for smart contract development [5, 27], similar tools and libraries for languages with stronger guarantees, such as Daml (see Appendix A), are notably absent. This disparity in testing tools and libraries across smart contract languages presents a significant challenge for the latter type of languages, restricting their potential adoption.

Daml draws inspiration from Haskell, adopting a purely functional programming model that facilitates the development of smart contracts for business workflows by emphasizing immutability, composability, and clear separation of concerns [31]. Unlike mainstream smart contract languages that expose low-level blockchain mechanics to developers, Daml focuses on abstracting these details away in favor of business logic modelling. Contracts in Daml are expressed as templates that define the data associated with an agreement, the parties involved, and the choices that govern how the contract may evolve over time. This abstraction allows developers to reason about workflows and permissions directly, without managing concerns such as gas costs, transaction ordering, or explicit state mutation. As a result, Daml supports platform-agnostic deployment, enabling the same contract logic to be executed over different ledger backends without modification [12].

Daml was selected as target for our contribution because it represents a class of smart contract languages designed around higher-level abstractions, strong safety guarantees, and explicit modelling of multi-party workflows [13]. Despite these design goals, the Daml

ecosystem currently lacks validation tool like practical property-based testing tools, making it an ideal target. Therefore, to contribute to supporting development of correct and secure smart contracts in the Daml ecosystem, we developed a library that enables property-based testing for real-world Daml smart contracts. The library simplifies the testing process by allowing users to evaluate relevant properties of a smart contract effectively and systematically. Concretely, the library we developed – Hypothesis2Daml – advances property-based testing for Daml smart contracts. It connects the Python testing framework Hypothesis [28] (see Appendix B.1). with the Daml JSON API [8], enabling contracts to be exercised automatically under a wide range of generated inputs.

In addition to the library, we compiled a benchmark suite to support the evaluation of Hypothesis2Daml. The suite comprises two parts: a collection of representative Daml contracts drawn from existing open-source projects, and a set of property-based tests developed specifically for these contracts. The benchmark suite consists of eight contracts, three templates, and twenty-eight property-based tests that cover happy paths, negative cases, and alternative interaction orders across representative workflows. Both the library and the benchmark suite are available in the first author's public GitHub repository under the MIT license [35].

2 Hypothesis2Daml

Hypothesis2Daml is a property-based testing framework designed to support the testing of Daml smart contracts through executable properties. The framework integrates the Hypothesis testing library with the Daml JSON API [8], enabling automated interaction with a running ledger from Python test suites. This design allows contracts to be exercised under a wide range of automatically generated inputs, while preserving the semantics and authorization rules enforced by the Daml runtime. Additional background and implementation details are provided in Appendix C.

Hypothesis2Daml provides helper functions for common ledger operations such as party allocation, contract creation, choice exercising, and querying. These helpers encapsulate authentication, request construction, and error handling, allowing tests to focus on property definitions rather than low-level protocol details. For every generated test case, the framework initializes an isolated ledger state by allocating fresh parties and deploying new contract instances, ensuring independence between executions. Ledger interactions are performed through API requests, so that authorization constraints defined in the Daml templates are respected. Failures raised by the ledger, such as violated preconditions or authorization errors, are surfaced as test failures.

This execution model supports both stateless and stateful testing. Individual properties can assert relationships between inputs and outputs of single choices, while more complex properties validate invariants across sequences of ledger interactions. Because tests execute against a real ledger, the observed behaviour corresponds exactly to deployed contract semantics, rather than to a simulated or reimplemented model.

2.1 An illustrative example: ZeroTokenBank

To illustrate the use of Hypothesis2Daml, we consider the `ZeroTokenBank` contract, a simplified banking example that manages user balances. The contract allows an operator to open accounts for users and supports deposit and withdraw operations subject to explicit preconditions. Despite its simplicity, the contract exhibits stateful behaviour and authorization constraints that are representative of common financial workflows.

Listing 1 below shows a simplified excerpt of the deposit logic implemented in Daml.

Listing 1 Excerpt of the ZeroTokenBank deposit logic.

```
choice Deposit : ContractId UserBalance
      with
        amount : Decimal
      controller user
      do
        assertMsg "Deposit must be less than 200" (amount < 200.0)
        create this with balance = balance + amount
```

The `Deposit` choice is controlled by the account owner and takes a single parameter, `amount`, representing the value to be deposited. Before updating the ledger state, the contract enforces a precondition using `assertMsg`, which requires the deposited amount to be strictly less than 200. If this condition holds, a new contract instance is created with the balance increased by the deposited amount. Otherwise, the choice execution is rejected by the ledger.

Using Hypothesis2Daml, properties for this contract are written as Python tests annotated with Hypothesis strategies. These tests interact with the ledger by creating parties, instantiating contracts, and exercising choices through the JSON API. For example, a property can specify that a valid deposit always increases a user's balance by exactly the deposited amount. Hypothesis automatically generates deposit values within the specified range and executes the corresponding ledger interactions, checking that the property holds for all cases.

Hypothesis2Daml also supports negative testing, where properties intentionally exceed contract constraints. In the `ZeroTokenBank` example, an additional property specifies that users should be able to make deposits in the range $[0.01, 500]$. A simplified version of this property is shown in Listing 2. The test generates deposit amounts using a Hypothesis strategy and then executes a complete workflow: allocating parties, creating a bank instance, opening an account, and finally exercising the `Deposit` choice with the generated amount.

Listing 2 Negative property for ZeroTokenBank deposits.

```
@given(amount=st.decimals(min_value="0.01",max_value="500",places=2))
@settings(max_examples=50, deadline=None)
def test_invalid_deposit(amount:Decimal):
    operator = allocate_unique_party("Operator")
    bank = create_bank(operator)
    contract = open_account(bank, operator, operator)
    deposit(contract, operator, amount)
```

However, the contract enforces a precondition that restricts deposits to values below 200. As a result, executions that violate this constraint cause the property to fail. When failures are observed, Hypothesis2Daml leverages Hypothesis' shrinking mechanism to produce minimal counterexamples [28]. In this case, the failing execution is reduced to a deposit of 200, clearly exposing the boundary at which the contract's precondition is violated.

Figure 1 shows an excerpt of the terminal output produced when executing this failing property.[1] After shrinking, Hypothesis reports a single falsifying example corresponding to the minimal deposit value that violates the contract precondition. This feedback allows developers to immediately identify the source of the failure without inspecting long execution traces or large generated inputs.

[1] Note that `int_to_decimal`(20000) = 200.00

Figure 1 Terminal output produced by Hypothesis2Daml for a failing `ZeroTokenBank` property, showing automatic shrinking to a minimal counterexample.

2.2 Reusable templates

The benchmark makes use of reusable Python test templates, such as `test_bank_template`, to reduce boilerplate and keep properties focused on contract behaviour. These templates abstract repetitive setup and interaction logic, including creating contract instances and exercising choices via the Daml JSON API. As a result, property definitions remain high-level and readable.

In Listing 3, the test template isolates setup actions such as creating the bank and exercising deposits from the properties being tested, allowing the properties themselves to focus solely on behavioural assertions.

Listing 3 Excerpt of the reusable ZeroTokenBank test template.

```
def create_bank(operator: str) -> str:
    res = make_request(
        "create",
        act_as=operator,
        template_id=BANK_TID,
        payload={"operator": operator},
    )
    return res["contractId"]

def deposit(ub_cid: str, user: str, amount: Decimal) -> str:
    res = make_request(
        "exercise",
        act_as=user,
        template_id=UB_TID,
        contract_id=ub_cid,
        choice="Deposit",
        argument={"amount": str(amount)},
    )
    return res["exerciseResult"]
```

2.3 Benchmark Suite

To assess the expressiveness an d practical feasibility of the approach, we constructed a benchmark suite comprising eight contracts and twenty-eight properties [35]. The benchmark includes representative contract patterns and properties adapted from existing suites and examples, and is complemented with contracts and properties developed specifically for this

work [11, 4, 10, 35]. The selected contracts cover common smart contract scenarios, including simple state-holding templates, asset transfer workflows involving multiple parties and roles, and contracts with explicit authorization and precondition checks. Together, these examples exercise both single-choice interactions and multi-step, stateful workflows. Each benchmark test executes against a running ledger via the Daml JSON API, exercising both expected behaviours and contract constraints under automatically generated inputs.

The associated properties target different classes of expected behaviour. Some properties validate functional invariants, such as balance preservation or state updates after valid operations. Others focus on safety constraints, asserting that invalid actions are rejected when preconditions or authorization requirements are violated. The benchmark also includes a property that intentionally exceeds contract limits, allowing the framework to demonstrate its ability to detect failing executions and produce minimal counterexamples through shrinking.

In addition to fixed interaction sequences, the benchmark includes tests that explore multiple valid workflows for the same contract. In such tests, parameters generated by Hypothesis determine the order in which certain choices are exercised, resulting in different but semantically equivalent execution paths. For example, properties for the `AssetTransfer` contract, a stateful workflow, validate convergence to the same ledger state regardless of whether inspection or appraisal occur first, or whether buyer or seller acceptance is performed first. These tests demonstrate that Hypothesis2Daml can validate properties across alternative interaction orders within the same contract.

From a performance perspective, the benchmark executes within a practical time frame. Figure 2 illustrates this for the `test_deposit_increases_balance` property of the `ZeroTokenBank` contract. The results exhibit an approximately linear relationship between the number of generated examples and the total execution time, indicating that each additional test case incurs a roughly constant cost. This suggests that the overhead of executing ledger interactions remains predictable as the number of property-based test cases increases. This linear scaling indicates each additional test case has a roughly constant cost, suggesting property based testing for Daml contracts can be integrated into standard testing workflows without significant performance overhead.

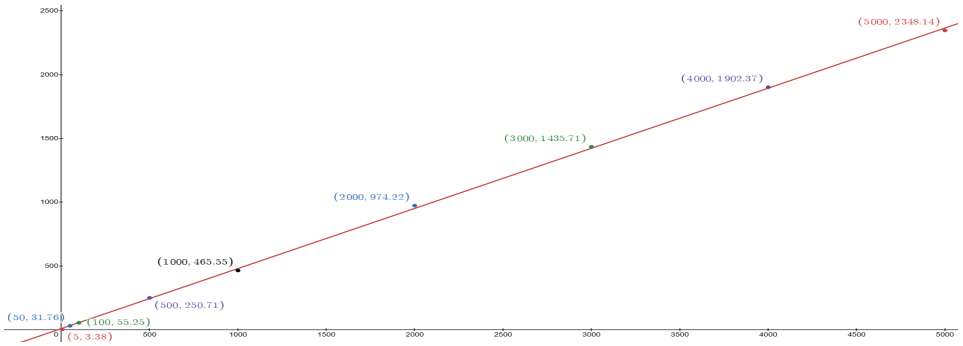

Figure 2 Execution time for the `test_deposit_increases_balance` property of the `ZeroTokenBank` contract.

3 Related Work

Prior work on improving smart contract reliability spans multiple directions, including fuzzing, property-based testing, and frameworks that combine testing with formal methods. ContractFuzzer is a representative fuzzing approach for Ethereum smart contracts, generating

inputs from ABI specifications and using vulnerability oracles to detect issues such as reentrancy [22]. Although effective for discovering known vulnerability classes at runtime, fuzzing approaches generally do not aim to validate developer-defined invariants or behavioural specifications beyond such predefined oracles [22].

Property-based testing tools for smart contracts are predominantly centered around Solidity. Solidity QuickCheck enables developers to encode properties directly for Solidity contracts and validate them under automatically generated test cases [24]. PropertyGPT takes a different angle by using LLMs to automate the generation of candidate invariants, preconditions, and postconditions for Solidity contracts, refining them through compilation feedback and evaluating them through property-based testing [26]. While PropertyGPT reduces the manual effort of writing properties, it remains tied to the Solidity ecosystem and still requires human review to assess property relevance and correctness [26].

ConCert integrates property-based testing with formal verification within Rocq, supporting the testing of execution traces and contract interactions through the QuickChick library [30]. This combination enables high-assurance validation, but requires contracts to be written or ported into Rocq, raising the adoption barrier and limiting scalability in practice [9, 37].

4 Contributions, Limitations, and Future Work

This work introduces Hypothesis2Daml, a property-based testing framework for Daml that connects the Hypothesis testing library to the Daml JSON API. The framework enables developers to specify invariants and behavioural constraints as executable tests, and to exercise these properties over realistic ledger interactions without porting contracts into a verification framework or requiring theorem proving.

The approach was evaluated using a benchmark comprising eight contracts and twenty-eight properties adapted from multiple sources and complemented with contracts developed in this work [35, 11, 4, 10]. The validation demonstrates that meaningful properties can be expressed, and that automated input generation is effective for exercising simple invariants and multi-step workflows. The evaluation also shows that execution time scales linearly with the number of generated examples, supporting the practical use of property-based testing in an interactive development workflow [35]. When failures were observed, Hypothesis2Daml leverages Hypothesis' shrinking mechanism to minimize failing cases.

A key limitation of Hypothesis2Daml is its reliance on the JSON API execution model, which introduces transport overhead and can reduce throughput for complex workflows. In addition, property definition remains developer-driven: the effectiveness of testing depends on the relevance of the chosen properties and the quality of the associated generators.

Future work includes improving throughput through concurrent execution (e.g., using `pytest-xdist`) [32], extending the framework with reusable property templates for common Daml patterns, and reducing the effort of writing and maintaining properties through automation and domain-aware generators. Finally, broader applicability could be studied by evaluating the approach on additional real-world Daml applications and workflows.

References

1 Property-based and stateful testing via hypothesis (brownie documentation). accessed 2025-08-29. URL: `https://eth-brownie.readthedocs.io/en/stable/tests-hypothesis-property.html`.

2 Quickcheck. Accessed: 2025-01-15. URL: `https://en.wikipedia.org/wiki/QuickCheck`.

3 Software testing: Property testing. Accessed: 2025-01-15. URL: https://en.wikipedia.org/wiki/Software_testing#Property_testing.

4 Solidity check, 2019. URL: https://github.com/xf97/SolidityCheck.

5 Solidity and smart contracts: A quick introduction, 2022. Accessed: 2025-01-22. URL: https://www.pluralsight.com/blog/software-development/what-is-solidity-smart-contracts.

6 An introduction to daml, 2024. Accessed: 2025-01-22. URL: https://docs.daml.com/daml/intro/0_Intro.html#:~:text=Daml%20is%20a%20smart%20contract,abstract%20Daml%20Ledger%20Model%20Concepts.

7 Testing smart contracts, 2024. Accessed: 2025-01-22. URL: https://ethereum.org/en/developers/docs/smart-contracts/testing.

8 Daml json api documentation, 2025. Accessed: 2025-09-03. URL: https://docs.daml.com/json-api/.

9 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: a smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 215–228. ACM, 2020. doi:10.1145/3372885.3373829.

10 Microsoft Azure. Blockchain workbench: Application and smart contract samples, 2018. Accessed: 2025-01-24. URL: https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples.

11 Massimo Bartoletti, Fabio Fioravanti, Giulia Matricardi, Roberto Pettinau, and Franco Sainas. Towards Benchmarking of Solidity Verification Tools. In Bruno Bernardo and Diego Marmsoler, editors, *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*, volume 118 of *Open Access Series in Informatics (OASIcs)*, pages 6:1–6:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/OASIcs.FMBC.2024.6.

12 Rob Behnke. Learning daml: Advantages and challenges, 2023. Accessed: 2025-01-22. URL: https://www.halborn.com/blog/post/learning-daml-advantages-and-challenges.

13 Alexander Bernauer, Sofia Faro, Rémy Haemmerle, Martin Huschenbett, Moritz Kiefer, Andreas Lochbihler, Jussi Mäki, Francesco Mazzoli, Simon Meier, Neil Mitchell, and Ratko G. Veprek. Daml: A smart contract language for securely automating real-world multi-party business workflows. *CoRR*, abs/2303.03749, 2023. doi:10.48550/arXiv.2303.03749.

14 Maryna Cherednychenko. Daml or solidity: What should you choose for smart contracts?, 2025. Accessed: 2025-01-22. URL: https://erbis.com/blog/daml-vs-solidity/#:~:text=Daml%20and%20Solidity%20are%20some,many%20blockchain%20platforms%20behind%20it.

15 Peer D. Understanding state management: Functional programming vs. imperative approaches, 2024. Accessed: 2025-01-22. URL: https://peerdh.com/blogs/programming-insights/understanding-state-management-functional-programming-vs-imperative-approaches.

16 Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. A Survey on Formal Verification for Solidity Smart Contracts. In *2021 Australasian Computer Science Week Multiconference*, pages 1–10. ACM, 2021. doi:10.1145/3437378.3437879.

17 Kevin George. The largest cryptocurrency hacks so far, 2024. Accessed: 2025-01-23. URL: https://www.investopedia.com/news/largest-cryptocurrency-hacks-so-far-year/.

18 Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, 2024. doi:10.1145/3597503.3639581.

19 Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. Tyche: Making Sense of PBT Effectiveness. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2024. doi:10.1145/3654777.3676407.

20 Hypothesis Works. Hypothesis quickstart guide. Version 6.138.3. URL: https://hypothesis.readthedocs.io/en/latest/quickstart.html.

21 Abdul Sami Jay. Property-based testing & fuzzing smart contracts, 2022. Accessed: 2025-01-23. URL: https://abdulsamijay.github.io/posts/fuzzing-property-testing.

22 Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018. doi:10.1145/3238147.3238177.

23 Cem Kaner. Measuring the effectiveness of software testers, 2003. Supported by NSF Grant EIA-0113539, Improving the Education of Software Testers. URL: https://web.archive.org/web/20100326042728/http://www.testingeducation.org/a/mest.pdf.

24 Kaan Kaçar. Solidity security practices part viii: Property-based testing, 2023. Accessed: 2025-01-30. URL: https://medium.com/coinmonks/solidity-security-practices-part-viii-property-based-testing-430daf391231.

25 Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Ethereum Smart Contract Analysis Tools: A Systematic Review. *IEEE Access*, 10:57037–57062, 2022. doi:10.1109/ACCESS.2022.3169902.

26 Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation, 2024. doi:10.14722/ndss.2025.241357.

27 Haoyang Ma, Wuqi Zhang, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. Towards Understanding the Bugs in Solidity Compiler. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1312–1324. ACM, 2024. doi:10.1145/3650212.3680362.

28 David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43), 2019. doi:10.21105/joss.01891.

29 Bertrand Meyer. Seven Principles of Software Testing. *Computer*, pages 99–101, 2008. URL: https://se.inf.ethz.ch/~meyer/publications/testing/principles.pdf.

30 Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Finding Smart Contract Vulnerabilities with ConCert's Property-Based Testing Framework. *OASIcs, Volume 105, FMBC 2022*, 105:2:1–2:13, 2022. Artwork Size: 13 pages, 863331 bytes ISBN: 9783959772501 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.FMBC.2022.2.

31 Bernard Njenga. Comparison between daml and solidity, 2022. Accessed: 2025-01-22. URL: https://bernardnjenga.hashnode.dev/comparison-between-daml-and-solidity.

32 pytest-dev team. pytest-xdist. Accessed 2025-09-13. URL: https://pytest-xdist.readthedocs.io/en/stable/.

33 Emanuel Trandafir. A guide to property-based testing in kotlin, 2024. Reviewed by Vinicius Peixoto, Accessed: 2025-02-05. URL: https://www.baeldung.com/kotlin/property-based-testing.

34 Miguel Valido. Hypothesis2Daml. Software (visited on 2026-03-30). URL: https://github.com/miguelvalido/Hypothesis2Daml, doi:10.4230/artifacts.25682.

35 Miguel Valido. Hypothesis2daml, 2025. URL: https://github.com/miguelvalido02/Hypothesis2Daml.

36 Cindy Wauters and Coen De Roover. Property-based testing within ml projects: an empirical study. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 648–653, 2024. doi:10.1109/ICSME58944.2024.00067.

37 Krl Ziborov. Formal verification of smart contracts in the concert framework, 2025. Accessed: 2025-01-30. URL: https://securityboulevard.com/2025/01/formal-verification-of-smart-contracts-in-the-concert-framework.

A Daml

Digital Asset Modelling Language (Daml) is an open-source, high-level, functional, and domain-specific language designed for building composable applications on an abstract ledger, referred to as the Daml Ledger [6]. The functional nature of Daml plays a central role

in ensuring predictable and deterministic contract behavior. Because functions are pure and state transitions are explicit, contract execution does not rely on hidden side effects. Instead, every state change corresponds to the consumption and creation of contracts on the ledger. This approach aligns with functional programming principles, where immutability and referential transparency lead to more maintainable and verifiable code. In contrast to imperative programming models, where mutable state can introduce subtle bugs, Daml's execution model ensures that contract behavior is reproducible and deterministic across executions [15].

A defining feature of Daml is its built-in privacy and authorization model. Rather than relying on global visibility of contract state, Daml allows developers to explicitly specify which parties are authorized to observe or act upon a given contract. Each contract template declares its signatories and observers, and choices are guarded by controllers that determine which parties may exercise them. These access rights are communicated to the underlying ledger, which enforces confidentiality by ensuring that each participant only receives data for contracts they are entitled to view. This design provides privacy without requiring developers to manage cryptographic primitives or custom access-control logic [14].

Role-based access control is a common requirement in smart contract applications, particularly those modeling real-world business processes. Many contracts define multiple roles with distinct permissions, such as owners, buyers, sellers, or inspectors. For example, the Azure Blockchain Workbench includes several smart contract examples that explicitly model roles to govern interactions, including marketplace scenarios with clearly separated responsibilities [10]. Daml's explicit permission model simplifies the implementation of such role-based workflows by making authorization constraints part of the contract specification itself. This leads to shorter, clearer, and less error-prone code, and allows developers to express complex multi-party interactions in a direct and declarative manner [14].

Despite its advantages, Daml also presents several limitations that impact its adoption. The language was primarily designed for private permissioned ledger environments, which typically involve fewer network participants than public blockchains. As a consequence, Daml-based systems offer limited decentralization when compared to fully public platforms, and system operators may retain greater control over consensus rules or governance mechanisms. This can increase the risk of collusion or centralization, depending on the deployment context. In addition, Daml was introduced relatively recently, in 2019, and therefore has a smaller ecosystem and community when compared to more established smart contract languages [14].

The relative youth of the Daml ecosystem also affects the availability of educational resources and experienced developers. While the language provides strong abstractions and safety guarantees, new developers may face a steeper learning curve due to limited documentation, tutorials, and third-party tooling. Furthermore, the smaller pool of experienced practitioners can make it more challenging for organizations to recruit specialists, increasing the cost and effort required to adopt Daml in production settings. These factors can pose practical barriers to entry, particularly for teams accustomed to more widely adopted smart contract platforms [14].

In summary, Daml represents a modern approach to smart contract development that prioritizes high-level abstractions, functional programming principles, and explicit modeling of permissions and workflows. By emphasizing immutability, deterministic state transitions, and built-in privacy, Daml provides a strong foundation for building secure and expressive business applications. Its design enables developers to focus on domain logic rather than low-level ledger mechanics, improving productivity and reducing implementation complexity. At the same time, its relatively young ecosystem and limited decentralization require careful consideration

when selecting Daml for a given application. As the language and its surrounding tooling mature, Daml has the potential to further establish itself as a valuable platform for modeling complex, real-world contractual workflows.

B Property-Based Testing

Property-based testing (PBT) is a software testing methodology that focuses on validating general properties of a system rather than individual input-output examples. The approach originated with the Haskell library QuickCheck, which introduced automated generation of test cases and systematic validation of properties against them [2, 3]. QuickCheck's combinator-based design established the foundations of PBT and inspired a wide range of libraries in other programming languages.

In contrast to traditional example-based testing, where developers manually specify concrete test cases, PBT requires developers to express formal specifications as executable properties. These properties describe high-level expectations about system behavior and are then tested against a large number of automatically generated inputs. For each generated input, the testing framework evaluates whether the property holds. If an input violates the property, it constitutes a counterexample that exposes a potential defect in the system [3, 18]. This process allows developers to reason about correctness over broad input spaces rather than isolated scenarios.

A central principle of PBT is that testing aims to uncover failures rather than prove correctness. This principle aligns with established software testing theory, which emphasizes that no finite testing process can demonstrate the absence of defects [29]. By systematically generating diverse inputs, PBT increases the likelihood of discovering edge cases and unexpected behaviors that would be difficult to anticipate through manual test design [18, 19]. As a result, PBT complements traditional testing approaches by broadening coverage and reducing the risk of untested scenarios.

An important feature of modern PBT frameworks is shrinking. When a counterexample is found, shrinking attempts to minimize the failing input while preserving the failure. The result is a simpler, more interpretable counterexample that helps developers quickly understand the root cause of the defect. This mechanism significantly improves debugging efficiency and is one of the key advantages of PBT over random testing [23, 19].

Automated oracles play a critical role in property-based testing. An oracle determines whether a test execution satisfies the specified property. In PBT, the property itself typically acts as the oracle, enabling consistent and repeatable validation without manual inspection. This is particularly important for large-scale or complex systems, where human verification of each test outcome would be infeasible [23, 19]. By relying on executable specifications, PBT ensures that test outcomes are evaluated uniformly across all generated inputs.

Property-based testing is closely related to fuzz testing, or fuzzing. Fuzzing involves executing programs with large volumes of random, unexpected, or malformed inputs to uncover crashes, vulnerabilities, or undefined behavior. While fuzzing often relies on generic oracles such as crashes or assertion failures, PBT integrates randomized input generation with developer-defined semantic properties. This combination enables more targeted exploration of system behavior, allowing developers to validate correctness conditions while simultaneously stress-testing the system under extreme or anomalous inputs [18].

The effectiveness of PBT depends heavily on the quality of the properties being defined. Properties serve as generalized specifications that describe expected behavior across many executions. In domains such as financial systems, properties may encode invariants like balance

preservation, non-negativity constraints, or consistency across transaction sequences [18, 19]. Defining such properties requires a deep understanding of the system and its domain, making property design one of the most challenging aspects of PBT. Poorly chosen properties may fail to capture relevant behaviors or may be too weak to detect meaningful defects.

Another important consideration is the design of input generators. Generators must produce inputs that are both diverse and relevant to the system under test. If generators are poorly constructed, they may focus excessively on trivial cases and fail to explore meaningful edge conditions. At the same time, overly complex generators can introduce unnecessary computational overhead, reducing test efficiency. Balancing coverage and performance is therefore essential for effective PBT [18].

Property-based testing is particularly well suited to decentralized and distributed systems. Such systems often involve complex state transitions, multiple interacting components, and large input spaces. Smart contracts, in particular, frequently manage valuable assets and enforce critical business rules. Any defect in these systems can lead to severe financial or security consequences. PBT enables systematic exploration of contract behavior under a wide range of inputs and interaction patterns, helping ensure adherence to intended rules under all tested conditions [21].

In the context of smart contracts, PBT can be used to validate safety properties such as enforcing access control or ensuring correct state evolution across transaction sequences. It also supports testing adversarial scenarios by generating unexpected or malicious inputs that may trigger vulnerabilities. By uncovering edge cases and rare execution paths, PBT strengthens confidence in contract correctness prior to deployment [19].

Despite its challenges, PBT offers a significant advantage over traditional testing approaches by systematically exploring large input spaces. Its ability to discover subtle defects, combined with automated shrinking and executable specifications, makes it especially valuable in high-stakes domains such as financial systems and safety-critical applications. When applied effectively, PBT serves as a powerful complement to other verification techniques, helping improve reliability, security, and developer confidence [21].

B.1 Hypothesis

Hypothesis is a popular property-based testing library for Python that automates input generation and systematically validates executable properties [28]. Inspired by QuickCheck, Hypothesis adapts the core ideas of property-based testing to Python's dynamic environment and integrates with existing testing workflows.

In contrast to unit testing, where developers manually enumerate concrete test cases, Hypothesis requires properties to be expressed as general specifications that must hold for a wide range of inputs. For each property, Hypothesis automatically generates diverse test inputs, evaluates whether the property holds, and reports violations as counterexamples [3, 18]. This enables developers to reason about program behavior over broad input spaces rather than isolated scenarios.

One of the defining features of Hypothesis is its shrinking mechanism. When a property violation is detected, Hypothesis attempts to minimize the failing input while preserving the failure. The resulting minimal counterexample is typically simpler and more interpretable than the original failing case, significantly improving debugging efficiency [23, 19]. This mechanism distinguishes Hypothesis from random testing approaches and is a key contributor to its practical usefulness.

Hypothesis provides a rich set of declarative data generation strategies. These include generators for primitive data types such as integers, floating-point numbers, and strings, as well as compositional strategies for complex structures such as lists, dictionaries, and

user-defined data types. By combining these strategies, developers can precisely control the shape and constraints of generated inputs, improving coverage while avoiding irrelevant or trivial cases [28, 20].

Although Hypothesis was not originally designed for blockchain or smart contract applications, its flexibility has enabled its adoption in this domain. For example, development frameworks such as Brownie support Hypothesis-based testing of Ethereum smart contracts, allowing developers to combine randomized input generation with contract-level assertions [1]. This demonstrates that Hypothesis can serve as a general-purpose foundation for property-based testing in high-stakes domains. Empirical studies highlight the role of Hypothesis in large-scale software development, with notable implementations in projects such as NumPy and PyTorch [36].

Overall, Hypothesis provides a mature and expressive implementation of property-based testing for Python. Its automated input generation, shrinking capabilities, and integration with existing testing frameworks make it a strong foundation for extending property-based testing to new domains, including smart contract systems.

C Hypothesis2Daml

The implementation of Hypothesis2Daml defines a set of core functions that connect the Hypothesis property-based testing library [28] with the Daml JSON API [8]. These functions cover authentication, request dispatching, error handling, and party allocation, and together provide the foundational building blocks required to construct property-based tests for Daml smart contracts. The framework is intentionally lightweight, delegating contract semantics to the Daml runtime while orchestrating test execution from Python.

C.1 Authentication helpers

Interaction with the Daml JSON API requires authorization headers encoded as JSON Web Tokens (JWTs). Hypothesis2Daml generates these headers directly, without relying on external token-handling libraries. Instead, it constructs the required tokens using Python's `json` and `base64` libraries, following the structure defined in the API specification [8].

A small helper function performs Base64URL encoding of token components (see `_b64url`[1]), and a higher-level function assembles the authorization payload for a specific acting party (see `make_auth`[2]). This design makes authentication transparent to the user: once the acting party is specified, the appropriate authorization header is constructed automatically. A similar helper is provided for administrative access, enabling operations such as party allocation by setting the required administrative flag (see `make_admin_auth`[3]).

It should be noted that this design relies on unsigned JWTs using `"alg":"none"`, which are accepted by the Daml Sandbox and Daml JSON API in development mode. In production deployments, the API requires properly signed tokens, for example using shared secrets or OAuth-based mechanisms, as described in the official documentation [8].

C.2 Request dispatching

All interactions with the ledger are centralized through a unified request dispatcher. This function abstracts over the three main operations supported by the Daml JSON API: contract creation, choice execution, and queries over active contracts. The user specifies the operation type together with the relevant parameters, and the dispatcher forwards the request to the appropriate endpoint (see `make_request`[4]).

Contract creation requests require a template identifier and payload, while choice execution additionally requires a contract identifier, choice name, and argument(s). Query operations retrieve active contracts based on template identifiers and optional filtering conditions. Centralizing these cases within Hypothesis2Daml eliminates redundant HTTP logic while ensuring uniform authentication and response handling across ledger interactions.

C.3 Error handling

Responses from the API are validated using a dedicated helper that checks both the HTTP status code and the structure of the returned JSON body. If a response does not conform to expectations, an `AssertionError` is raised with contextual information about the failing operation (see `ensure_ok`[5]).

This design choice ensures that errors are surfaced immediately during property-based testing. Failures such as violated assertions, authorization errors, or malformed responses are propagated directly to the test framework, making it easier to diagnose issues.

C.4 Party allocation and isolation

Property-based testing requires that individual test cases execute in isolation to prevent interference between generated examples. Hypothesis2Daml provides helper functions to allocate fresh parties for this purpose. One function performs direct party allocation by invoking the `/parties/allocate` endpoint and extracting the resulting party identifier (see `allocate_party`[6]).

To guarantee uniqueness across test cases, a second helper builds on this mechanism by generating party identifiers that incorporate a UUID fragment. This ensures that no party names collide between different Hypothesis-generated examples (see `allocate_unique_party`[7]).

Together, these helpers ensure that each test execution runs in a clean environment with freshly allocated parties and contracts. This isolation is essential for reliable property-based testing, as it guarantees that observed failures are attributable solely to the generated inputs and tested properties, rather than to residual ledger state from previous executions.