PROGRAM SYNTHESIS FOR DECLARATIVE SYSTEMS

Haoxian Chen

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2023

Supervisor of Dissertation

_____

Boon Thau Loo, Professor, Computer and Information Science

Graduate Group Chairperson

_____

Mayur Naik, Professor, Computer and Information Science

Dissertation Committee:

Mayur Naik, Chair, Professor of Computer and Information Science

Andre Scedrov, Professor of Mathematics, Professor of Computer and Information Science

Vincent Liu, Assistant Professor of Computer and Information Science

Yuepeng Wang, Assistant Professor in School of Computing Science, Simon Fraser University

PROGRAM SYNTHESIS FOR DECLARATIVE SYSTEMS

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Boon Thau Loo, for his guidance and support over my PhD study at Penn. I could not have undertaken this journey without his mentoring.

My committee consists of Mayur Naik, Andre Scedrov, Vincent Liu, and Yuepeng Wang. I am grateful for their feedback and suggestion on this dissertation, as well as their valuable guidance on my general research, presentation, and career planning.

I would also like to thank all my collaborators during my PhD study, in particular Mukund Raghothaman, who taught me a lot of important skills in research and technical writing in my first project. Other projects are very enjoyable and educational for me too. And I was fortunate to have such opportunities to learn from a group of amazing researchers, including Anduo Wang from Temple University, Behnaz Arzani, Kevin Hsieh from Microsoft Research, Chenyuan Wu, Andrew Zhao, Gerald Whitters, Mohammad Javad Amiri, Lan Lu and Brendan Massey from UPenn.

I have spent a wonderful time at Penn. I enjoy the conversations and discussions with my lab mates at the NetDB group, DSL seminar, and the department: Qizhen Zhang, Hui Lyu, Lei Shi, Max Demoulin, Nik Sultana, Chenyuan Wu, Mohammad Javad Amiri, Gerald Whitters, Lan Lu, Tao Luo, Heena Nagda, Bhavana Mehta, Leshang Chen, Xinyi Chen, Elizabeth Dinella, Hangfeng He, Yishuai Li, Yao Li, Danni Ma, Kelvin K.W. Ng, Pardis Pashakhanloo, Xuejie Si, Yinjun Wu, Jiali Xing, Yi Zhang, Ke Zhong, and many others.

I want to thank my buddies Xing Yue and Dong Shi. We went to graduate school at the same time in the US. Hanging out and chatting with them helps me release pressure from work and stay positive in life.

Finally, I want to thank my family, especially my parents, my partner Deng Miao,

and my cat Ruan-ruan. Their love and companion have kept my spirits and motivation high during graduate school. This dissertation would not have been possible without their emotional support.

This dissertation is supported by NSF under Grant CCF-2107429, CCF-2107261, CCF-2124431, CNS-2104882, CNS-2104882, CNS-2107147, NSERC Discovery Grant, and U.S. Office of Naval Research under award numbers N00014-20-1-2635 and N00014-18-1-2618.

ABSTRACT

PROGRAM SYNTHESIS FOR DECLARATIVE SYSTEMS

Haoxian Chen

Boon Thau Loo

Formal methods are essential in assuring system correctness. However, formal specification languages have steep learning curves, thus hindering broader application to system development in practice. To address this problem, we propose NetSpec, a tool that generates system specification via intuitive example-based interface, DeCon, a high-level language for Ethereum smart contracts that provides unified interfaces for contract implementation and specification, and DCV, a safety verification tool for DeCon.

NetSpec aims to be i) highly expressive, capable of synthesizing network specifications with complex semantics; ii) scalable, by virtue of using a novel stochastic search algorithm to efficiently explore an unbounded solution space, and iii) robust, avoiding the need for exhaustive input-output examples by actively generating new examples. Our experiments demonstrate that NetSpec can synthesize a wide range of specifications used in network verification, analysis, and implementations. Furthermore, NetSpec improves upon existing approaches in terms of expressiveness, efficiency, and robustness to examples.

DeCon, a specification language for Ethereum smart contracts, models a contract as a set of relational tables that store transaction records, driven by the observation that smart contract operations and contract-level properties can be naturally expressed as relational constraints. This relational representation enables convenient specification of contract properties, facilitates run-time monitoring of potential property violations, and brings clarity to debugging via data provenance. DeCon programs are compiled into executable Solidity programs, with instrumentation for run-time property monitoring. Our case studies demonstrate that DeCon

can implement realistic smart contracts such as ERC20 and ERC721 digital tokens. The evaluation shows that DeCon has comparable efficiency with the open-source reference implementation, incurring 14% median gas overhead for execution, and another 16% median gas overhead for run-time verification.

DCV is a sound and fully automatic verification tool for DeCon contracts. It proves safety properties by mathematical induction and can automatically infer inductive invariants without annotations from the developer. Our evaluation shows that DCV is effective in verifying smart contracts adapted from public repositories, and can verify contracts not supported by other tools. Furthermore, DCV significantly outperforms baseline tools in verification time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

Formal specifications are vital for assuring system correctness. In networking systems, formal specification of the network topology, communication protocols, and execution environment are used to perform a wide range of quality assurance tasks, including verification [59, 30, 33, 116, 117], analysis [29, 82, 37], and debugging [43, 122]. In cloud management, cluster administrators who wish to ascertain reachability of nodes must specify the desired behavior using declarative queries [29, 82]. In distributed systems, programmers who wish to verify certain system properties rely on formal specifications of a wide range of protocols, including inter-domain routing [117, 59, 33], consensus protocols [24, 106], and security protocols [44]. Furthermore, various domain specific languages [92, 81, 54, 32] rely on formal specifications expressed in logic as a basis for generating actual implementations, thereby bridging the specifications-implementation divide.

Despite their promising benefits, formal specifications have not yet gained mainstream adoption in practice. Today, it remains challenging for a network practitioner to write these formal specifications in the first place. It is even harder to ensure that the specifications capture all aspects of the network. Formal languages have steep learning curves, and it is difficult to find engineers who are simultaneously well-versed in network operations and formal methods. Consequently, despite the progress in tools for network verification and analysis, undertaking these tasks still necessitates a formal methods expert who can at least write the desired properties or model the network in formal specification languages.

On the other hand, even with written specifications, keeping them consistent with evolving systems is also challenging. System designers and the engineering

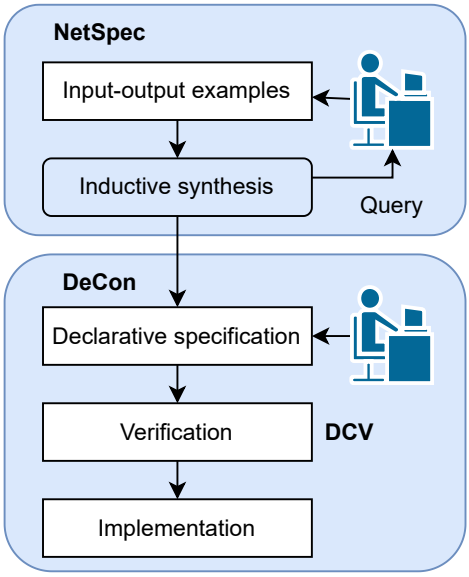


Figure 1.1: System development flow: from design, verification, to implementation. This dissertation proposes two systems, NetSpec and DeCon, each tackling particular stages for different application domains. NetSpec synthesizes declarative specifications from input-output examples for system protocols. DeCon takes declarative specification for Ethereum smart contracts, performs verification (DCV), and generates implementation in Solidity codes.

team need to communicate carefully to make sure the implementation is faithful to the specification.

This dissertation proposes to improve the efficiency of the system specification process from two angles: learning a system specification from input-output examples, and generate system implementation automatically from specification.

Figure 1.1 shows the proposed system development process. A system designer may first propose a system design via input-output examples, e.g., message exchanges between system nodes. Then an inductive program synthesizer generates declarative system specification consistent with these input-output examples. Given the system design specification, together with property specification, formal verification tools can be applied to look for bugs or prove properties for the system.

NetSpec focuses on the specification generation phase. It is an inductive program synthesis tool that generates declarative specifications for general network protocols from protocol input-output examples. To mitigate the under-specified situation, that is, some critical examples are missing, NetSpec augments the example set by generating additional input example, and querying the user to annotates the correct output. If the verification is successful, the specification is then automatically compiled into executable codes as system implementation. Otherwise, the verifier returns a counter example to the user to revise the system design.

For emerging areas where a lot of new applications are being developed, the efficiency of the design and implementation cycle can be improved by introducing a high level language that is both formal and executable.

We apply this idea to the domain of Ethereum smart contracts, and propose DeCon, a declarative language for smart contract implementation and specification. It generates efficient executable programs from the high-level declarative specification. DeCon also supports both run-time and static verification of safety properties.

The rest of the dissertation is organized as follows. Chapter 2 gives a brief introduction of declarative programming. Chapter 3 introduces NetSpec, an example-based specification synthesis tool for network protocols. Chapter 4 introduces DeCon, a declarative language for Ethereum smart contracts. Chapter 5 introduces DCV, an automatic verification tool for DeCon contracts, targeting safety invariants. Chapter 6 concludes this dissertation and discusses future work.

CHAPTER 2

DECLARATIVE SPECIFICATION LANGUAGE

This chapter provides a formal definition of the declarative specification language used in this dissertation. The design of the language is motivated by two key goals: the ability to express a wide range of system specifications, and the ability to leverage a variety of network verifiers, analyzers, and implementations.

(input relation)	`I`		
(output relation)	`O`		
(function)	`F`		
(aggregation)	`A`	$\in$	{ `min`, `max`, `count` }
(variable)	x		
(body literal)	b	::=	$\texttt{I}(\bar{x})$ | $\texttt{!I}(\bar{x})$ | $\texttt{O}(\bar{x})$ | $\texttt{F}(\bar{x})$
(head argument)	a	::=	x | $\texttt{A}(x)$
(head literal)	h	::=	$\texttt{O}(\bar{a})$
(rule)	r	::=	h `:-` b_1, ..., b_n
(specification)	p	::=	{ r_1, ..., r_n }

Figure 2.1: Abstract syntax of specifications in NetSpec. Input relation (I), output relation (O), user-defined functions (F), and aggregation (A) are application-specific.

Figure 2.1 presents the abstract syntax of specifications. We elucidate it using our running example of the shortest-path routing specification shown in Figure 2.2. A specification is a program whose inputs and outputs are a set of relations. In our routing example, the input relations include `link`, which represents the network topology, as well as common predicates such as `in` (list membership). The output

```
// compute available paths
r1: path(x, y, p, c) :- link(x, y, c), p=[x, y].
r2: path(x, y, x::p1 ,c1+c2) :- link(x, z, c1),
   path(z, y, p1, c2), !(x in p1).
// select the minimum cost path
r3: minCost(x, y, min<c>) :- path(x, y, _, c).
r4: bestPath(x, y, p, mc) :- path(x, y, p, mc),
   minCost(x, y, mc).
```

Figure 2.2: Declarative specification for shortest-path routing protocol.

relations include `bestPath`, which represents the shortest path between every pair of nodes in the input network, as well as relations such as `path` and `minCost` which hold intermediate results needed to compute `bestPath`.

A specification comprises a set of rules that specify how to compute the output relations from the input relations. Our routing example comprises four rules denoted `r1` through `r4`. Each rule is a Horn clause of the form:

$$R_h(\bar{x_h}) \text{ :- } R_1(\bar{x_1}), ..., R_n(\bar{x_n})$$

where the $\bar{x_i}$'s are vectors of variables of appropriate arity. Each rule is read from right-to-left as a universally quantified implication: for all variable valuations $\bar{x}$, if each of tuples $R_1(\bar{x_1})$, ..., $R_n(\bar{x_n})$ are derivable, then so is $R_h(\bar{x_h})$.

For instance, rule `r4` in our routing example states that if `path`(x, y, p, mc) and `minCost`(x, y, mc) are derivable, then so is `bestPath`(x, y, p, mc). This rule also depicts a basic logic operation: conjunction (i.e., join). On the other hand, disjunction (i.e., union) is expressed by means of different rules with the same head relation, as illustrated by rules `r1` and `r2` which denote the base case and inductive step, respectively, for computing the `path` relation. These two rules also illustrate recursion—an operation commonly needed in network specifications to specify reachability properties.

The features described thus far constitute the declarative logic programming language Datalog [20]. However, Datalog is inadequate to express real-world network specifications with rich functionality. The specification language of NetSpec therefore extends Datalog with three additional kinds of operations: negation (denoted `!`), aggregation (e.g., `min` and `count`), and user-defined functions, which include common utility functions such as `::` (list prepend) and + (integer addition).

Specifications are executable programs: execution begins with all output relations initialized to empty, and proceeds by repeatedly evaluating the rules until the output relations stop changing. The syntactic restrictions described above ensure a deterministic result regardless of the order in which rules are evaluated. However, note that the presence of recursion together with user-defined functions can lead to non-termination (e.g., by recursively applying integer addition).

An important benefit of this declarative specification is its suitability for a variety of networking tasks. They can be verified using SDN verifiers such as Vericon [30] and FlowLog [92], and routing verifiers such as Batfish [59] and FSR [117]; They can be analyzed using network analysis tools such as NOD [82],Tiros [29] and ExSPAN [125]. Lastly, they can be compiled to distributed implementations in Network Datalog [81] and FlowLog [92].

In Chapter 3, we will show how to synthesize these declarative specifications from input-output examples. In Chapter 4, we adopt this declarative specification language for smart contracts, and show how to perform run-time verification and code generation based on this declarative specification. Chapter 5 shows the how to perform automatic safety verification for such declarative smart contracts.

CHAPTER 3

SYNTHESIZING SPECIFICATIONS FROM INPUT-OUTPUT EXAMPLES

3.1 Introduction

This chapter presents NetSpec, a *specification-by-example* (SBE) toolkit that aims to automatically synthesize formal specifications of network protocols in logic. NetSpec aims to make formal network analysis more accessible to network programmers, who do not necessarily have expertise in formal methods. In the SBE paradigm, programmers provide input-output examples of their protocol designs. These designs can be handwritten or derived from actual runtime communication traces. NetSpec then applies program synthesis techniques to automatically yield the logical specifications which are amenable to verification [116] or generation of distributed implementations [81].

Our choice of logic as a basis for NetSpec is motivated by the fact that many formal network models trace their roots to logical specifications. In particular, we target an extension of the declarative logic programming language Datalog [20], which is popular in the literature on network verification [59, 54, 30, 116, 117], analysis [92, 29, 82], debugging [122, 43], and implementation [92, 81, 24, 106]. Thus, our logical specifications can be seen as declarative programs in themselves: the input comprises facts about a network (e.g., topology, VM configurations, etc) or incoming messages (e.g., route requests), while the output comprises actual network state (e.g., the shortest path, the reachable VM pairs, etc) or outgoing messages (e.g., route updates).

We envision NetSpec being used in a variety of settings:

1. rapid prototyping of a protocol design by compiling the synthesized logical specifications into distributed implementations.
2. verifying network protocols at design time by providing input-output examples that can be proof-checked based on its synthesized logical specifications. When a design bug is revealed by a verifier, the user can correct the design by adding new examples.
3. taking a legacy program and deriving its logical specifications from runtime executions for subsequent verification or software analysis. When a verifier finds a counter-example, it can be used to test against the legacy program. If the legacy program exhibits undesired behavior, a real bug is caught. Otherwise, the logical specifications is inaccurate, and can be refined by adding the counter-example to NetSpec.

To this end, NetSpec provides key features that advance upon state-of-the-art programming-by-example approaches and enable it to effectively address the above use-cases. We next elucidate each of these features:

- **Expressivity.** NetSpec supports expressive features necessitated by complex semantics involved in network specifications. These features include recursion, aggregation, and user-defined functions (UDFs). None of the existing techniques for synthesizing declarative programs support this combination of features, which precludes them from targeting many common network specifications, e.g., routing protocols and consensus protocols.

- **Scalability.** NetSpec uses a novel best-first search algorithm that incrementally proceeds from simple to complex programs, with the ability to rapidly backtrack, which enables to efficiently explore an unbounded search space and produce succinct specifications. In contrast, existing techniques either require the user to bound the search space [78, 98] (e.g., by providing the maximum number of operators), or suffer in terms of efficiency by exploring a large number of

incorrect programs [87].

- **Robustness.** NetSpec is robust to the quality of input-output examples. Approaches based on programming-by-example rely on the user to craft a complete set of examples in order to learn the correct program. However, it is easy to miss corner cases when providing these examples manually. NetSpec proactively detects the incompleteness in the specified examples, and generates new input queries to the example provider—a network operator or a legacy implementation. These new inputs, together with the provider's answers as the outputs, improve the example quality and enable NetSpec to unambiguously learn a correct program.

We have developed a prototype of NetSpec and evaluate it on a suite of 26 benchmarks that encompass a wide range of network protocols in different subdomains, including network analysis, software-defined networking (SDN), sensor networks, routing protocols, and consensus protocols. Our experiments demonstrate that NetSpec can faithfully synthesize most logical specifications in under a few seconds, with the most complex one in slightly more than 1 minute. In contrast, state-of-the-art tools GenSynth [87] and Scythe [118] cannot synthesize benchmarks requiring either aggregation or user-defined functions (10 out of 26), and benchmarks requiring recursion or user-defined functions (11 out of 26), respectively. Moreover, the specifications synthesized by NetSpec can be directly compiled into declarative networking [92, 81] for distributed implementations.

To validate NetSpec on actual implementations, we further demonstrate that NetSpec is able to synthesize logical specifications from actual program execution traces derived from popular open-source SDN controller implementations written in Floodlight [58] and POX [97], highlighting its ability to synthesize specifications for large-scale programs.

Contributions. To summarize, the key technical contributions of this chapter are as follows:

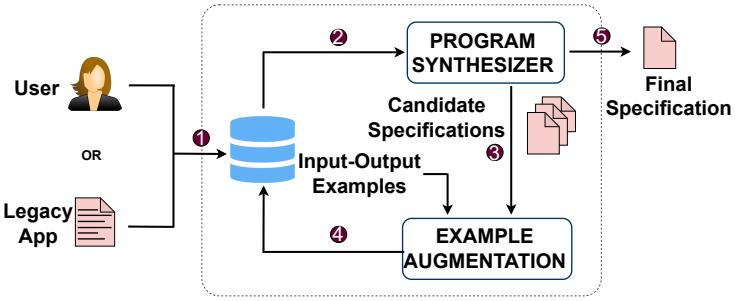
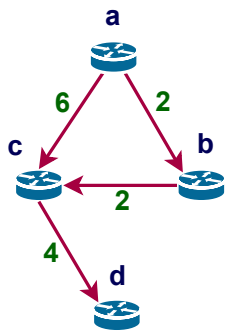


Figure 3.1: Architecture of NetSpec.

- We propose a novel synthesis algorithm to efficiently synthesize highly expressive network specifications from input-output examples. The specifications, expressed in first-order relational logic, have a variety of uses including verifying, analysing, and generating implementations.
- Since programming-by-example approaches are susceptible to missing examples, we develop a novel example generation algorithm to supplement synthesis. It queries the example provider for new examples that guide the synthesis algorithm to an unambiguous specification.
- We realize our approach in a tool NetSpec and evaluate it on diverse benchmarks and use-cases. NetSpec is able to correctly synthesize a wide-range of network protocols within seconds and is robust to missing examples. Moreover, we demonstrate that NetSpec outperforms state-of-the-art synthesis approaches in terms of its expressiveness, and in the quality of its synthesized programs.

3.2 Illustrative Example

In this section, we illustrate the end-to-end operation of NetSpec using the shortest path routing protocol as an example. The overall architecture of NetSpec is depicted in Figure 3.1. In Sections 3.2.1, 3.2.2, and 3.2.3, we describe the input-output examples, the synthesis algorithm, and the example augmentation process respectively.



link

src	dst	cost
a	b	2
b	c	2
a	c	6
c	d	4

bestPath

src	dst	path	cost
a	b	[a,b]	2
a	c	[a,b,c]	4
a	d	[a,b,c,d]	8
b	c	[b,c]	2
b	d	[b,c,d]	7
c	d	[c,d]	4

Figure 3.2: A small network topology as a weighted directed graph (left), its relational representation (middle), and the expected output (right).

3.2.1 Problem Specification

NetSpec takes two kinds of input: (1) A set of input-output example pairs, where each pair is consist of a set of input tables, and a set of output tables. These tables are relational, where each row is interpreted as relational tuples in Datalog. (2) Optionally, a list of user-defined functions and aggregators that could appear in the output specification. And NetSpec returns a logical specification, in the syntax of Datalog, that is consistent with the input-output examples. In the remainder of this paper, we will use "specification" and "program" to refer to NetSpec's output interchangeably.

Figure 3.2 depicts such an example for our shortest path routing protocol. In this example, one input-output pair is provided. The input table is named `link`, describing the network topology as a weighted graph. And the output table is named `bestPath`, specifying an optimal path for each pair of source and destination nodes. Functions including list initialization ($l = [x, y]$), concatenation ($x :: l$), and membership checking ($x \text{ in } l$), and aggregators (min and max) are also provided.

From this data, NetSpec automatically synthesizes the declarative logical specification shown in Figure 2.2. We have expressed the specification using the syntax

for Datalog, which we briefly review in Chapter 2. The first two rules specify paths between pairs of nodes and their associated costs: rule `r1` specifies a network link as a one-hop path, and rule `r2` specifies the transitive case. In particular, `x::p1` prepends node `x` to the head of path `p1`, and `!(x in p1)` checks that `x` is not in path `p1`, to avoid generating loops and to enforce termination. Rules `r3` and `r4` select the path with the minimum cost as the output best path.

This specification provides a high-level abstraction for verifying route convergence properties [117] and explaining route derivations [125]. Similarly, logical specification of other routing protocols can also be used to reason about network connectivity under different network dynamics [82, 59].

Despite the simplicity of the final specification, several aspects of the synthesis problem make it challenging in practice. First, the search space is enormous. For example, the rule `r2` contains 13 variable occurrences, so that there are $13! \approx 10^9$ ways of filling in its variables even after the rest of the rule structure is fixed. Furthermore, interaction between the rules makes the problem non-compositional, and techniques which synthesize one rule at a time become inapplicable [90, 52]. Finally, because input-output examples often under-specify the target concept and because of the undecidability of program equivalence [20], it is difficult to determine whether the synthesized specification correctly captures the user's intent.

3.2.2 Synthesis by Optimization

We organize the synthesis algorithm as an optimization problem and illustrate the process in Figure 3.3. Each node in the figure represents a candidate program, and its outgoing edges indicate each of its possible offspring. We highlight critical steps that lead to the final program and defer details of the algorithm to the next two sections.

Conceptually, we consider three possible modifications to each candidate program: introducing rules, introducing literals within a rule, and introducing aggrega-

tion operators. In the rest of this section, we first describe the overall search strategy for applying the modifications, and then outline each one of the three modification steps.

Search strategies. The objective function of the optimization problem is based on two measures of success on a candidate program s:

$$\text{score}(s) = \text{precision}(s) \times \text{recall}(s) \tag{3.1}$$

In particular, given the set of expected output tuples O_{exp}, and a candidate specification s that produces set of output tuples O_{ret}, we calculate $\text{precision}(s) = |O_{exp} \cap O_{ret}|/|O_{ret}|$, which is the fraction of tuples produced which are expected, and $\text{recall}(s) = |O_{exp} \cap O_{ret}|/|O_{exp}|$, which is the fraction of expected tuples which are produced by the candidate specification. The $\text{score}(s)$ is discounted by a $\gamma(s)$ metric that is a fraction of columns whose column values are all known given s.

Starting with an empty program, with score 0, the synthesis algorithm repeatedly generates offspring programs by applying all mutation strategies, and adds these offspring into a set of candidate programs. The next program to mutate is sampled from offspring that have higher scores, or the whole set of candidate programs when no offspring has a higher score.

In Figure 3.3, the input relation `link` generates 3 of the 6 expected `bestPath` tuples in Programs 1 and 2. For instance, the rule `bestPath(x,y,p,c):- link(x,y,c), p=[x,y]` has a recall of 0.5 and a precision of 0.75, and has the highest score among all candidate programs. A red color `bestPath` tuple denoting the shortest path from `a` to `c` is incorrect and needs to be fixed. Moreover, some `bestPath` tuples are missing. In subsequent steps, the candidate program with the highest score is successively modified to include the transitive rule for paths, and the aggregation operation to select the optimum weight path. Eventually, the red tuple is corrected, the missing tuples generated, and we converge on the best paths that matches the given input-output examples.

We next describe the three modification steps that can be applied to the current

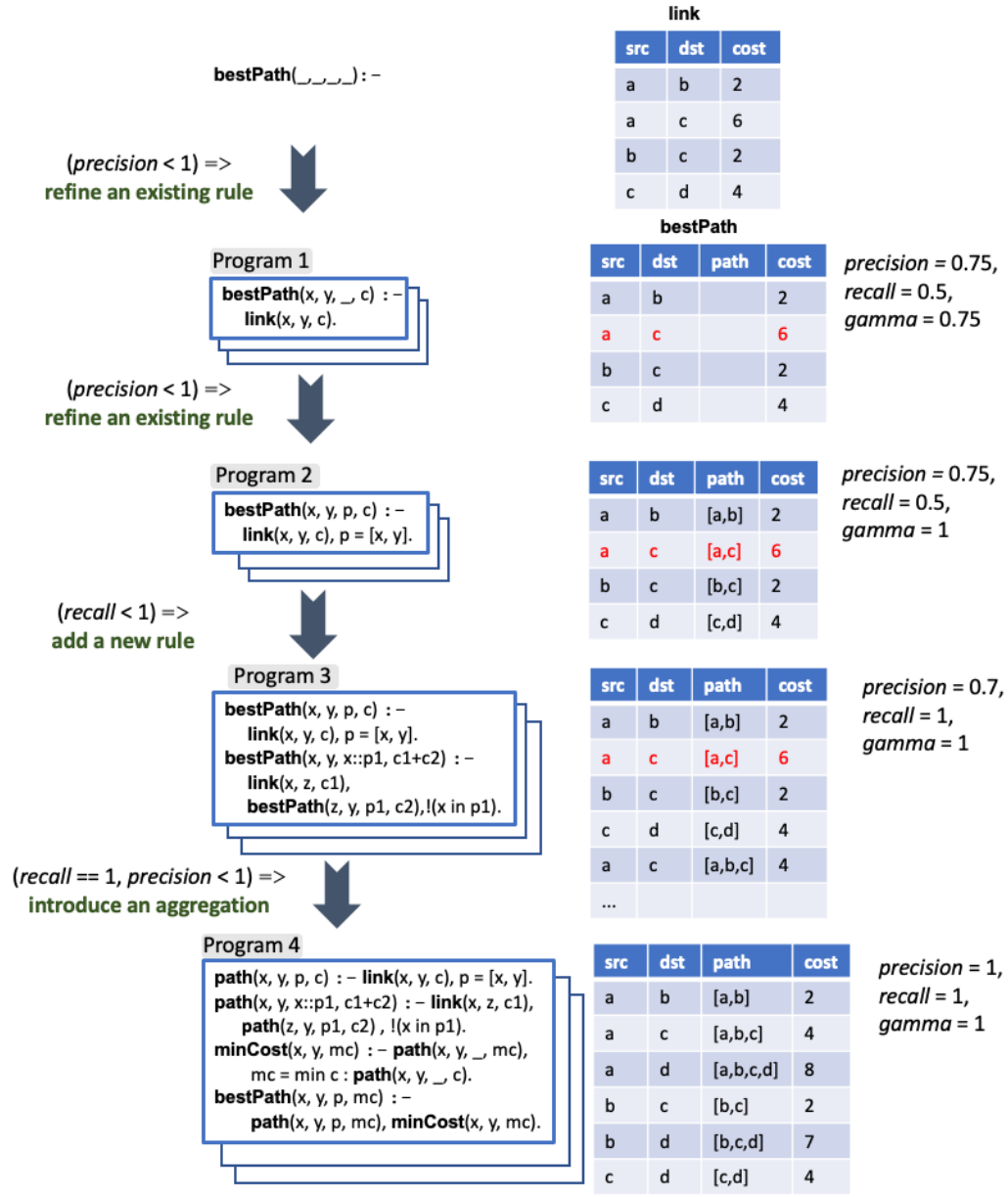


Figure 3.3: NetSpec synthesis procedure. On the left is the input table `link`. The highlighted blue boxes show the three intermediate programs leading up to the final solution. The tables underneath describe their respective outputs.

best candidate program. Each modification is described by referencing the generated candidate programs (1–4) in Figure 3.3.

Modification 1: Introducing new rules. The algorithm begins by enumerating single rule programs which produce at least one expected output tuple. The same rule generation algorithm is also invoked when the intermediate program fails to produce a desired output tuple, i.e., it has imperfect recall (less than 1). Each synthesized rule is chosen so that it produces at least one desirable tuple which is currently missing. These rules are synthesized by repeatedly introducing literals (modification 2) to the set of minimal rules, which contain only one head literal and one body literal, until it produces at least one expected output tuple.

We illustrate this process for the running example in Figure 3.3. Midway through running the best-first search algorithm after two refinement steps, the precision and recall of the best candidate program are 0.75 and 0.5 respectively. At this point, the best candidate program is only able to generate one-hop best paths by virtue of the rule `bestPath(x,y,p,c):- link(x,y,c),p=[x,y]` (Program 2). New rules need to be added so that one can generate outputs for paths that are two-hops and beyond, and this is done by adding the recursive rule that contains $bestPath$ in the rule body. Upon adding this new rule, the recall of the resulting output (Program 3) is increased to 1 although the precision is still not yet 1 (pending one additional modification to introduce aggregates).

Modification 2: Rule refinement by introducing literals. If the precision of a candidate program is less than 1, the algorithm adds new constraints to its rules by introducing literals, or by augmenting them with aggregation operations. By adding new literals to its rules, the algorithm produces an offspring program s' which produces a subset of the output tuples produced by the original program s.

To provide some intuition on rule refinement, we consider the scenario shown in Figure 3.3 where the current best candidate is the partial rule `bestPath(x,y,_,c):- link(x,y,c)` (Program 1). Since the third column of the output relation has not yet

been specified, the algorithm scores this partial rule by only comparing the remaining columns to the reference output. Intuitively, the partial rule mispredicts the cost of the `(a,c)` path, so that the program has a precision of 0.5 and a recall of 0.75. This score is additionally discounted by a factor of $\gamma = 0.75$ to account for incompleteness in output and bias the rule search towards faster rule completion. At this point, the rule refinement adds a literal $p = [x, y]$ where $[\ ]$ is a path concatenation function which is one of the candidate user-defined functions provided to the synthesis algorithm. Interestingly, with this refinement, while $precision$ is unchanged in the resulting output (Program 2), γ increases to 1 and all column values are known.

Observe that this process of adding literals provides flexibility in supporting arbitrary functions because it makes no assumptions about the underlying semantics. It is also highly efficient because it only considers one literal at a time, instead of arbitrary combinations of literals.

Modification 3: Aggregation operators. The final way to modify a program output is to apply an aggregation operation to produce one of its output columns. Consider the rule `r3` which finds the length of the shortest path between `x` and `y`. Informally, the aggregation operator `min` first groups the output tuples by their source and destination nodes, `(x,y)`, and then aggregates over all possible values of `c` for which a path exists: `path(x,y,_,c)`. In Figure 3.3, after adding the `min` aggregate to a candidate (Program 3), the algorithm converges upon the final solution (Program 4).

3.2.3 The Example Augmentation Process

The synthesis algorithm discovers all programs which are consistent with the input-output examples up to a maximum depth. When the provided input-output examples only partially constrain the possible solutions, the algorithm may discover multiple solutions, all of which are consistent with the data. We show two pos-

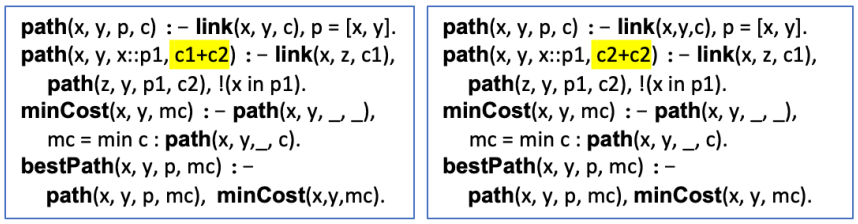


Figure 3.4: Two solutions of the routing protocol specified with incomplete examples. The difference is highlighted.

sible solutions to the shortest path routing program in Figure 3.4, and highlight their difference in yellow. In general, dealing with under-constrained specifications is a major outstanding challenge in programming-by-example (PBE) systems, and solution disambiguation is an important contribution of this paper.

One reason for the difficulty of disambiguation is that the equivalence checking problem for Datalog programs is undecidable [20]. To address this, NetSpec employs the idea of differential testing from program analysis [86] to repeatedly run the two programs with randomly perturbed inputs. In our example, by modifying the link costs, one obtains an input which reveals the difference between the two programs. We can then request the user to provide the ground truth for this new example, which will in turn eliminate at least one of the candidate solutions. The process repeats until only one program remains, or NetSpec fails to generate a distinguishing input among the programs. In this latter case, NetSpec produces the simplest program as the final solution. Because the enumeration process is biased towards smaller programs, and because the tie-breaking routine favors the syntactically smallest solution, in practice NetSpec produces small programs that are also readily interpretable and resistant to over-fitting.

3.3 Problem formulation

The specification synthesis problem is defined as follows. Given a set of input tuples I, a set of output tuples O, and a set of library functions F, return a Datalog (Chapter 2) program p such that $p(I) = O$, and p is constructed using the relations in I and O, and functions in F.

Note that to ensure well-founded semantics (Datalog programs with negations should be stratified [20, Chapter 15]), NetSpec only applies negations to input relations or functions, e.g., to function `in` in rule `r2`. In addition, to keep the synthesis task tractable, NetSpec applies the following syntactic restrictions to each rules:

1. Each rule can have at most 2 literals of the same relation. For example, a rule $h(x, w) :- p(x, y), p(y, z), p(z, w)$ would not be generated by NetSpec because it has 3 literals of relation "p".

2. A negation literal can have at most 2 bound variables. For example, literal $!p(a, b, c)$ would not be added to rules, because it has 3 bound variables (a, b, c). But literal $!p(a, b, \_)$ could be added.

3. At most one aggregation is used in each program.

4. Aggregation can only be applied in the head of a rule. For instance, applying `min` in rule `r3` yields the minimum cost c over all paths between each pair of nodes x and y in the input network.

5. A user-defined function's result can only be used in the head of a rule, e.g., the result of + in rule `r2`.

Algorithm 1 $\mathrm{Synth}(I, O, F)$. Given a set of input tuples I, expected output tuples O, and a library of functions F, produces all consistent programs.

1. Initialize the set of solutions, $S \coloneqq \emptyset$, the set of candidate programs, $Q \coloneqq \{P_0\}$, and the current program $P \coloneqq P_0$, where P_0 is the empty program.

2. While $Q \neq \emptyset$, do:

   (a) Let $\mathrm{Offspring}(P) = \{P'_1, P'_2, \dots\}$ be the offspring of P, computed according to Equation 3.2.

   (b) Update the set of solutions, and add all remaining programs for further enumeration:

$$S \coloneqq S \cup \{P' \in \mathrm{Offspring}(P) \mid \mathrm{score}(P') = 1\}, \text{ and}$$
$$Q \coloneqq (Q \setminus P) \cup \{P' \in \mathrm{Offspring}(P) \mid \mathrm{score}(P') > 0\}.$$

   (c) Sample the next program to explore:

$$\mathrm{HS} \coloneqq \{P' \in \mathrm{Offspring}(P) \mid \mathrm{score}(P') > \mathrm{score}(P)\}$$
$$\mathrm{HR} \coloneqq \{P' \in \mathrm{Offspring}(P) \mid \mathrm{recall}(P') > \mathrm{recall}(P)\}$$
$$P \coloneqq \begin{cases} \mathrm{Sample}(\mathrm{HS}, P) & \text{if } \mathrm{HS} \neq \emptyset \\ \mathrm{Sample}(\mathrm{HR}, P) & \text{else if } \mathrm{HR} \neq \emptyset \\ \mathrm{Sample}(Q, P) & \text{otherwise.} \end{cases}$$

3. Return S.

3.4 Synthesis Algorithm

$$\text{Offspring}(P) = O_D(P) \cup O_C(P) \cup O_A(P), \text{ where} \tag{3.2}$$

$$O_D(P) = \begin{cases} \text{AddRule}(P) & \text{if } \text{recall}(P) < 1, \text{ and} \\ \emptyset & \text{otherwise,} \end{cases}$$

$$O_C(P) = \begin{cases} \text{ExtRule}(P) & \text{if } \text{precision}(P) \leq 1, \text{ and} \\ \emptyset & \text{otherwise, and} \end{cases}$$

$$O_A(P) = \begin{cases} \text{MkAgg}(P) & \text{if } \text{precision}(P) \leq 1 \text{ and} \\ & \quad \text{recall}(P) = 1, \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$$

We present the top-level synthesis procedure in Algorithm 1. It takes only a set of input tuples (I), and a set of output tuples (O), and we will explain how to support multiple instances of input-output example pairs in section 3.4.4. As described in Section 3.2, it models an optimization problem in the Datalog program space, where each state is a program. And the objective function $\text{score}(p)$ is defined as the product of $\text{precision}(p)$ and $\text{recall}(p)$.

At each iteration, the algorithm explores the program space by mutating the current program P, which gives rise to several offspring (step 2a). $\text{Offspring}(P)$ is defined in equation 3.2, where the D, C, and A subscripts indicate the generation of offspring by adding new rules (disjunctions), extending existing rules (conjunctions), and by applying aggregation operators, respectively. The conditions to apply each of these mutation strategy are based on the semantics of Datalog. Both adding clause in a conjunction rule, and aggregate the output of current program, monotonically decrease the size of program output (number of tuples), thus may improve precision, but may also lower recall at the same time. Thus they are only applied when the program has imperfect precision. Adding a rule, on the contrary, mono-

tonically increase the size of program output, and could potentially improve recall, but lower precision at the same time. Therefore it is only applied when the program has imperfect recall. In addition, we assume the program space where only one aggregator is used, thus we wait until all necessary rules are added to reach perfect recall before applying aggregation.

In step 2b, offspring with score 1 are added to the solution set S. Offspring with score 0 implies that it produces no desired output ($P(I) \cap O = \emptyset$). Such offspring are discarded, based on the previous observation that, applying $\mathrm{ExtRule}$ or MkAgg to a program monotonically decreases the output size of the program. This means that further extending any rule of this program would not produce any desired output, except adding new rules. In addition, we assume that in all solution programs, every non-aggregate rule directly contributes to some output in O. Therefore, only rules with non-zero score ($P(I) \cap O \neq \emptyset$) are added into the set of candidate programs (Q) for further mutations.

In step 2c, the next program to explore is sampled probabilistically. When there are offspring with higher score or higher recall, these offspring will always be chosen as the next program to explore. Otherwise, it samples from the whole set of candidate programs Q. The sub-routine $\mathrm{Sample}(Q, P)$ is described in algorithm 2. Borrowing the idea in simulated annealing, it iteratively samples a candidate $P' \in Q$ uniformly, and accept it with probability computed by equation 3.3. Intuitively, when a candidate program has higher score than current program, it is accepted with probability 1. Otherwise, it is accepted with probability between 0 to 1, depending on how worse its score compared to the current program.

The rest of this section describes each of these mutation strategies, and formal properties of the synthesis algorithm.

3.4.1 Adding and extending Rules

The $\mathrm{AddRule}(P)$ procedure enumerates all minimal rules and generate offspring by

Algorithm 2 $\mathrm{Sample}(Q, P)$. Given a set of candidate programs Q, the current program P, return a program $P' \in Q$.

For $k \in \{1, 2, ..., K_{max}\}$, do:

1. Uniformly sample a program P' from Q.

2. Compute acceptance probability of P':

$$
\begin{aligned}
s_0 &\coloneqq \mathrm{score}(P), s_1 \coloneqq \mathrm{score}(P') \\
T &\coloneqq 1 - \frac{k}{K_{max}} \\
Pr[\text{accept } P'] &\coloneqq \begin{cases} 1 & \text{if } s_1 > s_0 \\ \exp(-\frac{s0 - s1}{T}) & \text{otherwise} \end{cases}
\end{aligned} \tag{3.3}
$$

3. If $Pr[\text{accept } P'] \geq \mathrm{random}(0, 1)$:

   - return P'

adding one minimal rule to P. A minimal rule is a rule that has only one literal in the body, and the head only have one field bound to the body, with all remaining fields being empty place holders ("_"). In the short-path routing example, one of the minimal rules is:

```
r_0: bestPath(x,_,_,_) :- link(x,_,_).
```

Let $\mathrm{MinimalRules}$ be the set of all minimal rules obtained from the given input and output relations, $\mathrm{AddRule}(P)$ is defined as:

$$\mathrm{AddRule}(P) \coloneqq \{(P \cup r) | r \in \mathrm{MinimalRules}\} \tag{3.4}$$

Next we introduce "ExtRule(P)" procedure, which further contains two atomic operations on a rule, namely "AddLiteral(r)" and "AddBinding(r)". They are defined as:

$$\mathrm{AddLiteral}(r) = \{r \wedge l | r \in P, l \in L\} \tag{3.5}$$

where L is the set of all literals whose relation is from the set of all input relations, output relations, and user-defined functions, and contains only empty place holders "_". $r \wedge l$ represents a new rule by adding literal l in conjunction with r's body. Continuing on the example on shortest-path routing, one of the new rules generated by "AddLiteral(r_0)" is:

```
r_1: bestPath(x,_,_,_) :- link(x,_,_), bestPath(_,_,_,_).
```

where $bestPath(\_, \_, \_, \_)$ is a literal instantiated from the output relation $bestPath$.

Next, "AddBinding(r)" is defined as follow:

$$\begin{aligned}\mathrm{AddBinding}(r) =& \{r \wedge (v_1 = v_2) | v_1, v_2 \in r \\ & \wedge dom(v_1) = dom(v_2)\}\end{aligned} \tag{3.6}$$

where $v_1, v_2 \in r$ means that variable v_1 and v_2 appear in the rule r, and $dom(v)$ is the domain of variable v, as specified in the schema of the literal where v appears. As an example, we show one of the rules generated by "AddBinding(r_1)":

```
r_2: bestPath(x,_,_,_) :- link(x,z,_), bestPath(z,_,_,_).
```

where the second variable in literal $link$ is bound with the first variable in literal $bestPath$ in the body. Note that instead of explicitly add a predicate that match two variables as: $bestPath(x,\_,\_,\_) : -link(x, v1, \_), bestPath(v2,\_,\_,\_), v1 = v2.$, we rename $v1$ and $v2$ to z for brevity.

Putting them together, "ExtRule(P)" generates all programs resulted from applying either "AddLiteral" or "AddBinding" to any one of the rules in program P. "ExtRule(P)" is defined as:

$$\begin{aligned}\mathrm{ExtRule}(P) =& \{(P \setminus r) \cup r' | r \in P, \\ & r' \in (AddLiteral(r) \cup AddBinding(r))\end{aligned} \tag{3.7}$$

3.4.2 Evaluating partial programs

When applying $\mathrm{AddRule}(P)$ and $\mathrm{ExtRule}(P)$, we will have partial rules in the program queue. By partial rule we mean rules that have empty place holders in the head. We further define partial programs as programs that contains at least one partial rule.

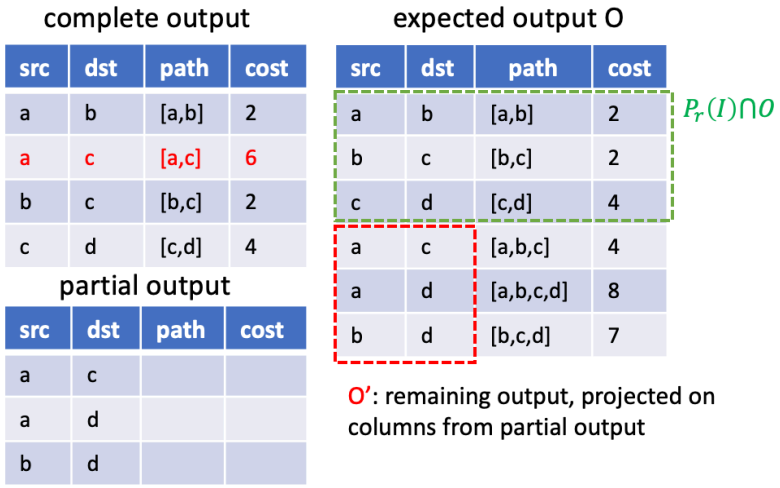
As an example, consider the four-place `bestPath(x, y, p, c)` relation, and a partial rule as the following:

```
r_p1: bestPath(x, y, _, _) :- link(x, z, _), bestPath(z, y, _, _).
```

Notice that this rule only produces the first two columns of the output relation, the source node and and the destination node, but does not produce the remaining two columns, the optimum path, and its length.

As a consequence, programs with these partial rules, such as $P \cup \{r_{p1}\}$, cannot be directly compared to the entire reference output O, and we are instead only able to compare its first two columns to $\pi_{\text{src,dest}}(O)$, borrowing the notation for projections from relational algebra.

Table 3.1: Example of scoring a partial program P_r. It contains a partial rule r, where only two fields in the head are determined, thus only two columns are generated by this rule. Precision is 0.86 because 6 out of the 7 output tuples are desired (in $O \cup \pi_c(O)$). Recall is composed of two parts, the complete tuples (3 in green box), and the partial tuples (3 in red box). The recall on the partially generated output is discounted by factor 0.5 because only 2 out of 4 columns are generated. Putting them together, the total recall is $\frac{3+3\times0.5}{6} = 0.75$.



This leads us to the following definition of precision and recall for partial programs P_r:

$$\text{precision}_d(P_r) = \frac{|P_r(I) \cap (O \cup \pi_c(O))|}{|P(I)|}, \tag{3.8}$$

$$O' = \{t \in (O \setminus P(I))|\ \pi_c(t) \in P(I)\}$$

$$\text{recall}_d(P_r) = \frac{|P(I) \cap O| + \gamma|O'|}{|O|} \tag{3.9}$$

where π_c is the projection operator that project to the columns that have been bound on the partial rule's head, The set O' is the subset O that are not in $P(I)$, but whose projection on columns c appear in $P(I)$, we visualize this set computation in Table 3.1.

3.4.3 Introducing Aggregation Operations

Algorithm 3 $\mathrm{MkAgg}(P)$. Produces offspring of P by introducing aggregation operators.

1. Let F be the family of aggregation operations, and Let C be the set of all columns in output relation R.

2. For each operator $\mathrm{op} \in F$, for each subset $C_{\text{agg}} \subseteq C$, and for each aggregation column $f \in C \setminus C_{\text{agg}}$, construct the offspring program P': First rename the output relation R of P to R_{base}, and let C_{rem} be the remaining columns $C \setminus C_{\text{agg}} \setminus \{f\}$. Then add the following two rules.

```
R_opt(C_agg, f_opt) :- R_base(C_agg, f_opt, _),
    f_opt = op f: R_base(C_agg, f, _).
R(C_agg, f_opt, C_rem) :- R_opt(C_agg, f_opt),
    R_base(C_agg, f_opt, C_rem).
```

3. Return all offspring produced in Step 2.

Algorithm 3 describes MkAgg Procedure. Recall the third program in our running example of Figure 3.3:

```
r1: bestPath(x, y, p, c) :- link(x, y, c), p = [x, y].
r2: bestPath(x, y, x::p, c1 + c2) :- link(x, z, c1),
     bestPath(x, y, p, c2), !(x in p1).
```

This program correctly predicts the reachability relation, but it produces additional incorrect paths, i.e., it has perfect recall but imperfect precision. In this case, NetSpec attempts to remedy the situation by introducing aggregation operators. It introduces two new rules, which may be informally interpreted as follows: The first rule selects a subset of columns (in this case, the source node `x` and the destination node `y`), and performs an aggregation on another column (in this case, computing

the minimum of all path weights which share the source and destination nodes). The second rule then selects the values of the remaining columns which lead to this maximization or minimization objective. Thus, after mutation, the following program is added to the queue:

```
r1: path(x, y, p, c) :- link(x, y, c), p = [x, y].
r2: path(x, y, x::p, c1 + c2) :- link(x, z, c1),
     bestPath(x, y, p, c2), !(x in p1).
r3: minPath(x, y, mc) :- path(x, y, _, mc),
     mc = min c: path(x, y, _, c).
r4: bestPath(x, y, p, mc) :- minPath(x, y, mc),
     path(x, y, p, mc).
```

3.4.4 Supporting multiple input-output example pairs

So far our algorithm description is based on one set of input tuples (I) and one set of output tuples (O). To support multiple instances of examples, NetSpec introduces an additional field 'InstanceID' to every tuple, which indicates the particular example instance that the tuple belongs to. Tuples across different instances are then combined into one set of input tuples (I), and one set of output tuples (O), respectively. During the rule search process, NetSpec only generates rules that bind all 'InstanceID' fields to one single variable. For example, a rule generated by NetSpec would look like the following:

```
h(v1,v2,i) :- p1(v1,v3,i), p2(v3,v2,i).
```

where all variables for 'InstanceId' field are bound to the same name i. Thus, the synthesis problem with multiple example instances is reduced to one with single instance, which is solved by Algorithm 1.

3.4.5 Soundness and completeness

Theorem 3.4.1 (Soundness). *Given a set of input tuples and a set of output tuples (I,O), when NetSpec terminates, its output S satisfies the following property:*

$$\forall p \in S, p(I) = O. \tag{3.10}$$

Proof sketch. In algorithm 1, a program p is added to solution set S if and only if $\text{score}(p) = 1$ (step 2b). To prove Theorem 3.4.1 suffice to show that:

$$\text{score}(p) = 1 \implies p(I) = O \tag{3.11}$$

where $\text{score}(p)$ is defined in equation 3.1. By the definition, when $\text{score}(p) = 1$, program output $p(I)$ has perfect precision and recall on the reference output O. This implies that $p(I) = O$.

Next, we state the completeness property. We first define the program space, using the following definitions:

Definition 3.4.2 (Empty program). *Program p is an empty program if and only if it consists of no rule.*

Definition 3.4.3 (Successor relation). *Let $\rightarrow$ be a binary relation on the set of Datalog programs:*

$$p \rightarrow q \iff q \in \textit{Offspring}(p) \tag{3.12}$$

Let $\rightarrow^*$ be a binary relation on the set of Datalog programs:

$$p \rightarrow^* q \iff p \rightarrow p_1 \rightarrow ... \rightarrow p_n \rightarrow q \tag{3.13}$$

where $n \geq 0$.

Definition 3.4.4 (Output-contributing rule). *Given a set of input tuples and a set of output tuples (I,O), a rule r in a program p is an output-contributing rule if r's evaluation result on input I intersects with O.*

A special case is for programs with aggregations (either $argMax$ or $argMin$). If r's output is aggregated, then r's result is compared with renamed tuples in O, whose relations are renamed as r's output relation. In the shortest-path example (Figure 2.2), $path$ relation is aggregated into $bestPath$, when determining if $r1$ is contributing to output, we rename relations of tuples in O from $bestPath$ to $path$, and then check intersection. If r is an aggregation rule, because NetSpec introduces an aggregation (min or max) rule and a selection rule simultaneously to achieve $argMin$ or $argMax$ semantics, r's output is interpreted as the derivation result of both aggregation and selection rules. If the shortest-path example, $r3$ and $r4$ are introduced simultaneously, and they are both considered output-contributing rules if the derivation results of $r4$ intersects with O.

All solutions of NetSpec contain only output-contributing rules. Because, during the rule extension phase in algorithm 1, a candidate program is discarded if the newly extended rule does not produce desired output.

Definition 3.4.5 (Program space)**.** *Given a set of input tuples and a set of output tuples (I,O), and a set of user-defined functions, let P_c be the set of Datalog programs that contain only output-contributing rules (definition 3.4.4). The program space is defined as programs in P_c that are descendants of the empty program p_0:*

$$\{p \in P_c |\ p_0 \rightarrow^* p\} \tag{3.14}$$

Theorem 3.4.6 (Weak completeness)**.** *For all input-output tables (I,O), if there exists a program p in the program space (definition 3.4.5), such that p terminates on input I within a time bound T, with output O, then NetSpec always returns a solution set S, which contains at least one such program p. Otherwise, it returns an empty solution set $S = \emptyset$.*

This is completeness theorem is "weak" because it assumes a smaller program space (programs where each rule derives some output tuples, definition 3.4.5) than

the full program space $\{p|p_0 \rightarrow^* p\}$. Appendix A.1 shows the proof sketch of Theorem 3.4.6.

3.5 Handling Incomplete Examples

We now describe the example augmentation process. Given an initial set of input-output examples, when multiple satisfying programs are found, NetSpec searches for a new input example that can differentiate these candidate programs, and asks the user to specify the expected output for this new input example. By actively querying the user for feedback, this allows the system to robustly learn programs even from a set of initially under-specified examples.

Algorithm 4 $\mathrm{Sample}()$. Samples new input tuples for disambiguation.

1. Initialize the set of input tuples $I \coloneqq \emptyset$.
2. For each input relation R:
   (a) Uniformly sample the number of tuples, $n \in \{1, 2, \ldots, n_{\max}\}$, where $n_{\max}$ is the upper bound on the size of the sampled tables.
   (b) Sample n tuples, t_1, t_2, ..., t_n, where each $t_i = (c_1, c_2, \ldots, c_k)$, all constants being uniformly sampled, and where k is the arity of the relation R.
   (c) Insert t_1, t_2, ..., t_n into I_R.
3. Return I.

We describe the core example sampling process in Algorithm 4. In particular, in step 2a), n_{max} is the maximum of the table sizes in the initial example input I. In step 2b), constants are uniformly sampled from the set of all constants, that appear in the initial example input I.

Given a set of candidate programs P_1, P_2, ..., P_n which are consistent on the initial example input I (i.e., $P_1(I), P_2(I), ..., P_n(I)$ match the initial example output), we repeatedly run the sample procedure to obtain k new example inputs, I_1, I_2, ..., I_k. We then choose an example input $I_q \in \{I_1, I_2, \ldots, I_k\}$ for the user to label the corresponding example output, as follows:

$$I_q = \underset{I_j}{\text{argmax}}(-\sum_O p_O \log(p_O)), \tag{3.15}$$

where O ranges over the set of example outputs $\{P_1(I_j), P_2(I_j), \ldots, P_n(I_j)\}$, and p_O is the fraction of the candidate programs which produce O as output. By maximizing the entropy of the new example, we eliminate as many programs as possible after user feedback. We illustrate this using $n = 4$ candidate programs and $k = 3$ sampled examples $\{I_1, I_2, I_3\}$ [1] such that:

1. The programs are consistent on I_1. Then, O ranges over a singleton set of outputs and $p_O = 1$, so the score of I_1 is $-(1 \cdot 1 \cdot log(1)) = 0$.
2. The programs are 50-50 split on I_2. Then, O ranges over a set of two distinct outputs and $p_O = 0.5$, so the score of I_2 is $-(2 \cdot 0.5 \cdot log(0.5)) \sim 0.69$.
3. Each program produces a unique output on I_3. Then, O ranges over a set of four distinct outputs and $p_O = 0.25$, so the score of I_3 is $-(4 \cdot 0.25 \cdot log(0.25)) \sim 1.38$.

Thus, I_3 would be selected as the new example, which corresponds with the intuition that user feedback would eliminate the most (i.e. 3 out of 4) candidate programs.

We repeat this procedure until the remaining programs can no longer be distinguished by the sampled inputs. This approach is similar to the query-by-committee method [103] and enables NetSpec to rapidly converge to the final solution.

Theorem 3.5.1. *Assume NetSpec is always able to disambiguate candidate programs, and that the user always gives correct answers to NetSpec's queries, if there exists a so-*

[1] Concrete examples: https://github.com/HaoxianChen/netspec/blob/master/docs/active-learning-example.md

lution p in the program space (Definition 3.4.5), then NetSpec always returns solutions that are logically equivalent to p after active-learning.

Proof sketch. By theorem 3.4.6, p is always in the solution set S after every iteration of synthesis. By the assumption that NetSpec is always able to disambiguate candidate programs, a new queries will always be generated to differentiate p from other solutions, until all programs in S are logically equivalent. Therefore, when NetSpec terminates, all solutions are logically equivalent to p.

3.6 Implementation

NetSpec is implemented in Scala and comprises $\sim$ 3.5K lines of code. [2] It uses Souffle [110] as the backend Datalog interpreter to validate the candidate specifications. In this section, we discuss implementation details regarding how NetSpec handles non-terminating candidate specifications, and how it synthesizes specifications with constants.

Handling non-terminating specifications. During the synthesis process, NetSpec could encounter non-terminating specifications in the presence of recursion and user-defined functions. For example, consider the following candidate which NetSpec encounters in the process of synthesizing the routing example in Section 3.2.1:

```
// compute available paths
r1: path(x,y,p,c) :- link(x,y,c), p=[x,y].
r2: path(x,y,x::p1,c1+c2) :- link(x,z,c1),
                             path(z,y,p1,c2).
```

This specification does not terminate when the input network topology represented by the `link` relation contains a cycle, since both `::` (list prepend) and `+` (integer addition) used in the recursive rule `r2` generate new values every time the rule is

[2] NetSpec is available at: https://github.com/HaoxianChen/netspec

evaluated. NetSpec handles such specifications by halting the specification interpreter after a timeout period, and considers their output to be empty.

Generating constants in specifications. Many network specifications in practice require constants. For example, in an SDN firewall specification, the controller application monitors and responds only to a certain port. Such a specification cannot be synthesized without the use of constants. On the other hand, naively adding constants into the specification can lead to over-fitting it to the provided input-output examples.

To distinguish specifications where constants are fundamentally needed from those which can be realized symbolically, NetSpec employs a fail-over mechanism: it embarks by only searching for symbolic specifications; when it exhausts the candidate program queue and fails to find a solution, it switches to use constants from the input-output examples. In experiments where NetSpec learns specifications from execution traces (Section 3.7.3), all firewall applications monitor certain ports on a dedicated switch. On their traces, the fail-over mechanism is triggered and NetSpec synthesizes specifications with constants on the switch and port field.

3.7 Evaluation

Our evaluation aims to answer the following four questions:

1. **Expressivity.** Is NetSpec able to synthesize a wide range of network specifications correctly, and how does its coverage compare to state-of-the-art synthesis tools?

2. **Efficiency.** Can NetSpec synthesize a network specification in reasonable time (on the order of seconds)?

3. **Robustness.** Is NetSpec robust to input-output example quality, in particular, can it handle incomplete examples?

4. **Scalability.** Can NetSpec learn specifications from examples derived from a large volume of execution traces? Note that this question goes beyond performance to also capture NetSpec's ability to debloat legacy applications.

Benchmarks. [3] We survey the use of declarative specifications from literature, and organize them into five categories: network analysis, SDN, sensor networks, consensus protocols, and routing protocols. Network analysis refers to prior work on formalizing reachability and other correctness properties in networks [29, 82, 92]. SDN specifications are from works on verifying correctness of controller programs [30, 92]. Sensor network specifications are based on a declarative sensor network system [50]. Consensus protocols [24] and distributed routing are based on declarative specifications targeted for distributed execution [81, 50], and verification [59, 54, 117]. Table 3.2 shows the list of benchmark specification and their characteristics.

Input-Output example generation. To provide examples free of bias to any synthesizer, we manually read through the documentations for each benchmark protocol, and come up with input-output examples that cover all the use scenarios described in the documentations. The example size is measured by the number of input-output example instances (i.e. groups of input-output tables), and the number of total tuples (i.e., number of rows in relational tables) in all instances, as shown in the "#Examples" column in Table 3.3.

Result validation. A synthesis result is correct if it is identical to the reference specification after two modifications: (1) variable renaming, and (2) removing redundant predicates and rules (if any). We manually validate all experiment results. Reference [14] illustrates how each benchmark is validated, and has synthesis results of all experiments. Modification (1) dominates the validation process and a few results require modification (2). In the remainder of this section, we refer to

[3] The full list of benchmarks: https://github.com/HaoxianChen/netspec/tree/master/benchmarks

Table 3.2: Benchmark specifications. The features of recursion, aggregation, and UDFs are highlighted in the "Features" columns.

Category	Specification	Features			Relation	
		recur.	agg.	UDF	input	output
Network	reachable	✓			1	1
analysis	path	✓		✓	1	1
	path-cost	✓		✓	1	1
	publicIP				4	1
	subnet				4	1
	sshTunnel	✓			1	1
	protection				3	1
	locality				3	1
SDN	learning-switch				2	3
	l2-pairs				2	3
	stateless-firewall				3	2
	stateful-firewall				5	3
	firewall-l3				6	3
	firewall-l3-stateful				5	3
Consensus	2pc		✓		5	2
protocol	paxos-acceptor				3	3
	paxos-proposer				4	1
	paxos-quorum		✓	✓	2	1
	paxos-maxballot				2	1
	paxos-decide				2	1
Routing	shortest-path	✓	✓	✓	1	1
protocol	least-congestion	✓	✓	✓	1	1
	ospf		✓	✓	2	1
	bgp		✓	✓	2	1
	tree		✓	✓	1	1
	min-admin		✓	✓	2	1
	rip		✓	✓	3	1
Sensor	evidence				2	1
network	store				2	1
	temperature-report				6	2
Wireless	dsdv			✓	4	1
routing	dsr-rreq				2	2
	dsr-rreq				2	2
	dsr-rerr				3	1

such results as *validated solutions*.

The rest of the section are structured as follows. In Section 3.7.1, we evaluate NetSpec's expressivity and efficiency by comparing with state-of-the-art program synthesis tools. In Section 3.7.2, we evaluate NetSpec's robustness to input-output example quality, on benchmarks with insufficient examples. In Section 3.7.3, we evaluate NetSpec's scalability on execution traces that consist of thousands of examples.

3.7.1 Synthesis Expressivity and Efficiency

We first evaluate expressivity and synthesis efficiency given sufficient examples. We compare NetSpec to two state-of-the-art tools, Facon [45] and GenSynth [87].

Applicable benchmarks. Like NetSpec, both Facon and GenSynth operate on relational input-output data. However, they are less expressive: neither tools support UDFs and aggregation. Therefore, we only run these tools on benchmarks that they apply to. In addition, some of the original benchmark specifications may have insufficient examples, i.e. missing corner cases and resulting in incorrect specifications. To evaluate synthesis efficiency and compare with baselines that do not augment examples, this section focuses on benchmarks with sufficient examples for this experiment (Table 3.3), where NetSpec returns validated solutions for at least 8 out of 10 repeated runs. We will revisit applications with insufficient examples in Section 3.7.2.

In addition, GenSynth does not support multiple instances of examples. We therefore combine the multiple instances into a single instance by unioning tuples that belong to the same relation into the same table. We avoid introducing spurious correlations across the original instances by renaming constants appropriately. Note that this process is challenging to automate since certain constants (e.g. port numbers) are global and must not be renamed. This highlights the benefit of supporting multiple example instances as well as constants in NetSpec.

Performance metric. We measure NetSpec's expressivity in terms of coverage of different network specifications from the areas of network analysis, SDN, sensor networks, consensus protocols, and routing protocols. To evaluate synthesis efficiency, we measure the end-to-end synthesis time, on a server with 32 2.6GHz cores and 125GB memory. Both NetSpec and Facon run in single thread. GenSynth, however, often runs indefinitely long in single thread, due to its high degree of nondeterminism. Therefore, we run GenSynth in 8-thread mode in order to obtain results within 20 minutes each.

Results. Table 3.3 summarizes our overall results. Focusing on the first two criteria of expressivity and efficiency, our main takeaways are as follows:

Expressivity. NetSpec successfully synthesizes all 23 benchmarks in Table 3.3, spanning different types of network protocols. On the other hand, due to limited language feature support, competing solutions such as Facon [45] and GenSynth [87] support only 9 benchmarks. In addition, Facon fails to synthesize the locality benchmark because it lies outside of Facon's program search space, which only contains Datalog rules where each relation appears at most once. Both Facon and GenSynth fails to synthesize learning-switch benchmark due to the lack of support for negation.

Efficiency. NetSpec is highly efficient. NetSpec finishes most benchmarks within one minute, with the exception of path-cost and the RIP protocols, which takes 92 seconds and 3 minutes respectively. On the other hand, Facon is only able to synthesize 8 out of 10 applicable benchmarks, and in fact, two benchmarks timed out after 20 minutes. Compared to GenSynth, NetSpec consistently finishes faster on all applicable benchmarks, except $reachable$ and $sshTunnel$, where NetSpec takes 6 seconds and 3 seconds longer respectively. This is impressive since NetSpec runs in single thread whereas GenSynth in 8.

Benefits of component-based synthesis. Our benchmarks also showcase the benefits of synthesis in a component-based fashion. We describe case studies based

Table 3.3: Synthesis results for benchmarks where the original examples are sufficient. The effort of specifying examples is described by the number of input-output instances and the total number of rows in all instances. Both synthesis time and output sizes are average across 10 runs. NS., Fa., and GS. stand for NetSpec, Facon and GenSynth respectively. In the "Time" column, $\times$ means the tool terminates and finds no solution, and TO means the tool times out after 20 minutes. For benchmarks where the tool is inapplicable, the time and size entries are left empty.

Category	Specification	Examples		Synthesis time (s)			Output size (#rules)		
		#Inst.	#Rows	NS.	Fa.	GS.	NS.	Fa.	GS.
Network	reachable	1	42	17	1	11	2	2	2
analysis	path	1	15	23			2		
	path-cost	3	17	92			2		
	publicIP	1	17	5	5	12	1	1	1
	sshTunnel	1	25	11	1	8	2	2	3
	locality	5	21	2	x	14	1		1
SDN	learning-switch	4	15	6	x	x	3		
	stateless-firewall	2	9	2	1	66	3	3	3
	firewall-l3	14	76	12	TO	199	5		6
Consensus	paxos-acceptor	5	14	7			4		
protocol	paxos-quorum	2	19	36			1		
	paxos-maxballot	1	8	3			3		
	paxos-decide	2	5	1			1		
Routing	shortest-path	1	10	14			4		
protocol	least-congestion	1	15	24			4		
	ospf	1	14	8			3		
	bgp	1	11	4			3		
	rip	1	18	180			4		
Sensor	evidence	1	5	1	1	4	1	1	1
network	store	1	7	1	1	6	1	1	1
Wireless	dsr-rrep	2	5	2			2		
routing	dsr-rreq	2	6	1			1		
	dsr-rerr	1	6	1			1		

on two protocols: PAXOS (paxos-* in the Table 3.3) and DSR (dsr-*) in Table 3.3. The original specification of PAXOS lies contains two layers of aggregations (first count the votes to determine which ballot has a quorum, and then decide a value by choosing the one with the maximum ballot value). In normal circumstances, this is beyond NetSpec's search capabilities since it can only synthesize programs with at most one layer of aggregations. However, by breaking PAXOS into different components, synthesis is not only possible but done efficiently.

DSR, on the other hand, can be synthesized as one monolithic protocol. Yet, by breaking its synthesis into component modules, it significantly reduce the number of examples to sufficiently specify the protocol. DSR handles three different kinds of input messages independently, thus it gives the opportunity to break down the synthesis task into independent modules. For example, when synthesizing a rule to process route request message, the synthesizer does not need to consider any input value of a route error message. On the other hand, if examples for all types of message handling are combined together, although NetSpec can still efficiently find a solution, but it will generate a lot of invalid solutions (consistent with input-output examples but not equivalent to the reference solution) due to the larger program space. We note that component-based synthesis strategy for DSR is not only complete, but also highly efficient.

3.7.2 Robustness to Insufficient Examples

We evaluate NetSpec's robustness to insufficient examples on two kinds of benchmarks: (1) benchmarks with insufficient examples (Table 3.4); (2) benchmarks with sufficient original examples, but some of the examples are randomly dropped to test NetSpec's limit (Figure 3.6).

Handling insufficient examples. For each benchmark in Table 3.4, we run NetSpec with active learning, which iteratively queries the user with extra input examples, until it finds no ambiguities in the examples.

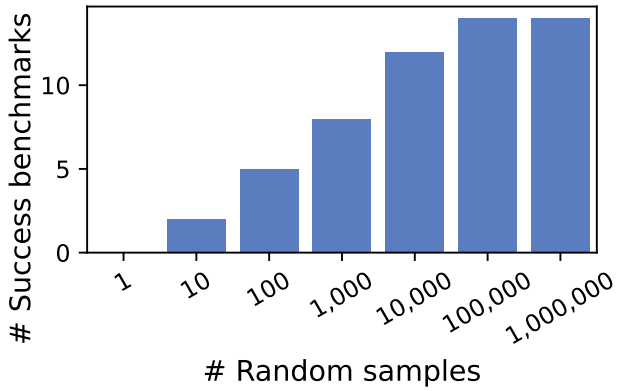


Figure 3.5: NetSpec solves more benchmarks as the number of random samples in active learning increases.

To determine the number of random samples in active learning phase (Section 3.5), we gradually increase the sample number from one to a million, and measure the number of benchmarks solved by NetSpec. Due to the randomness of the active learning algorithm, a benchmark is determined successful if NetSpec returns validated solutions in all 10 repeated experiment runs.

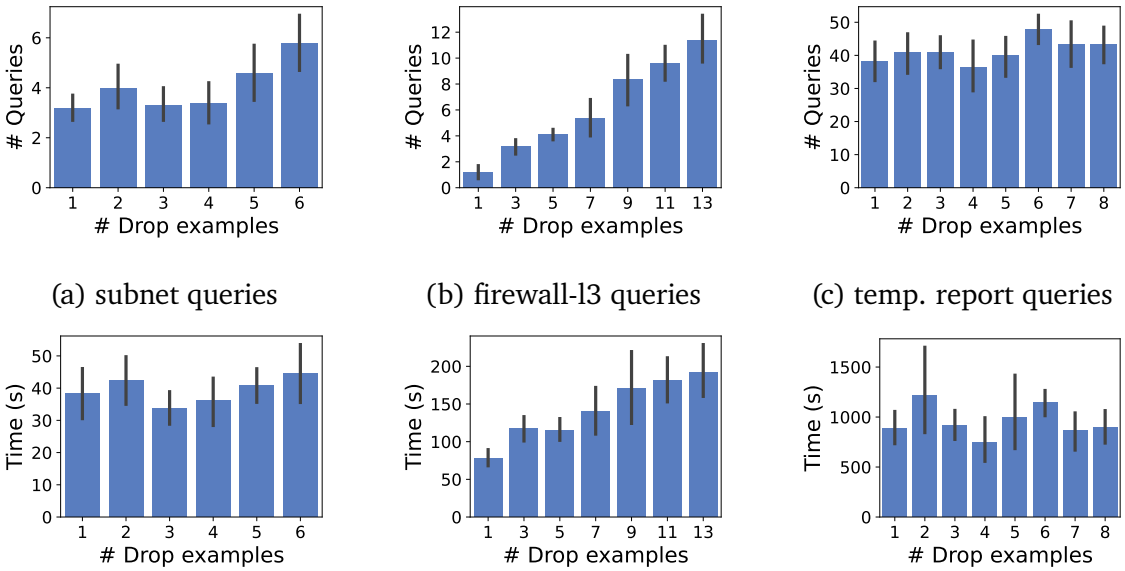
Figure 3.5 shows the results, where NetSpec's performance saturates at 100K samples, with 15 out of 18 benchmarks succeeding. The remaining three benchmarks involve the most complex specifications. They timed out and return incorrect specifications (consistent with input-output examples but different from the reference). The time bound is introduced because NetSpec is designed for interactive use. Recall that the active-learning phase involves multiple iterations of specification synthesis and new input example generation, whose output is annotated by protocol designers. Since increasing the sampling parameter beyond 100K does not reduce the end-to-end time (i.e. does not helping to solve more benchmarks within the time budget), we use 100K as the number of random samples for our remaining experiments, and the default value for this parameter. Users could also determine this parameter value for problems in their domains via the same experi-

ment procedure.

Table 3.4: Active learning results for benchmarks that needs example augmentation. NetSpec runs active learning to augment the input-output examples and finds the validated solutions. In the "#Queries" column, "med." and "max" stand for median and maximum. Column "Time" shows the average end-to-end time. "TO" means timing out after 1 hour. Column "Output size" reports the size of the synthesized specification, measured in the number of Datalog rules.

Category	Specification	# Examples		# Queries		Time (s)	Output size
		#Inst.	#Rows	med.	max		(#rules)
Network analysis	subnet	7	27	3	6	43	1
	sshTunnel	1	25	2	4	243	2
	protection	2	19	3	11	45	1
	locality	5	21	9.5	16	154	1
SDN	learning-switch	4	15	6.5	10	65	3
	l2-pairs	6	23	7.5	10	90	4
	stateful-firewall	15	78	18.5	26	367	5
	firewall-l3-stateful	13	68	12.5	15	189	4
Consensus protocol	2pc	8	129	30	48	TO	2
	acceptor	5	14	19.5	26	430	4
	proposer	7	26	31.5	35	723	2
Routing protocol	ospf	1	14	5.5	9	2,736	3
	bgp	1	11	6	10	2,945	3
	tree	1	15	2	3	1,841	4
	min-admin	1	8	6	8	1,591	3
	rip	1	18	4	8	TO	4
Wireless	dsdv	6	23	17.5	28	TO	4
Sensor	temperature-report	10	34	42.5	52	474	2

Table 3.4 shows the detailed statistics of the active learning experiments. The number of queries varies across different benchmarks, with the median ranging from 2 to 42.5. Similarly, the end-to-end time ranges from 43 seconds to 2,945 seconds across benchmarks.



(a) subnet queries (b) firewall-l3 queries (c) temp. report queries

(d) subnet time (e) firewall-l3 time (f) temp. report time

Figure 3.6: Randomly drop examples.

For the three benchmarks (2pc, rip, and dsdv) that timed out, their relations compose a much larger program space (rules with many predicates and aggregators), and thus more examples are needed to unambiguously specify a program. This results in too many iterations in active learning, which is a limitation of input-output example based interface. Synthesizing correct specifications for them require either reducing the number of queries or improving synthesis efficiency, which remains an interesting avenue of future work.

Randomly omitted examples. We further stress test NetSpec by randomly dropping examples. Three benchmarks with at least seven examples are chosen for this experiment. For each of them, examples are dropped incrementally until reaching the most extreme case, where every output relation appears in only one example instance. Otherwise, an output relation is missed from all examples, and NetSpec would skip synthesizing rules for the relation, thus returning an incomplete program.

Figure 3.6 presents the distributions of the number of queries and the end-to-end active learning times for each benchmark across ten repeated runs. The number of queries shows positive correlation with the number of dropped examples with one exception. Benchmark "temperature-report" shows weaker correlation because each active learning run takes more queries (40 ± 5) than the original example set size (9). Hence, the impact of dropping examples is weaker than benchmarks where the overall number of queries are smaller.

For relationship between end-to-end time and the number of dropped examples, "firewall-l3" shows strong positive correlation. The "temperature-report" shows no such correlation, which is expected since its query numbers is not correlated with the number of dropped examples (Figure 3.6c). On the other hand, for the "subnet" benchmark, although the number of dropped examples has strong correlation with the number of queries (Figure 3.6a), the correlation with time (Figure 3.6d) is weaker. This is because the end-to-end time is dominated by the synthesis time of the final few runs (where examples are almost sufficient). The earlier runs are fast because, with just a few examples left, NetSpec can quickly find superficial solutions and query new examples. When examples are sufficient, the solution becomes much more complex (more literals in a rule). This complexity, coupled with the randomness of the synthesis algorithm, leads to larger variation in synthesis time.

In the "subnet" and "temperate-report" benchmarks, NetSpec recovers validated specifications consistently (100%), even under extreme cases where only 1 is example is left. "Firewall-l3", on the other hand, fails when examples for particular message types are all dropped. For instance, if no example responds to ARP packets, then all candidate programs would just ignore ARP packets, because NetSpec discard rules that derive no reference output, although the reference program actually handles ARP packets. In practice, however, it is rare that a protocol designer provides no examples for a typical type of output at all.

Overall, NetSpec's active learning mechanism is effective in improving example quality and finding validated solutions. General differential testing of programs is a hard problem. However, by separating protocol logic from implementation details, declarative specifications drastically reduce the search space to differentiate alternative specifications. By exploiting the simplicity of declarative specifications, NetSpec's simple random testing mechanism is able to effectively disambiguate alternative protocol specifications.

Table 3.5: Learning specifications from program communication traces. Column "#T" measures the number of input output messages in the execution trace, column "#R" measures the number of rules of the reference specification, column "#Queries" measure the median and maximum number of queries posted by NetSpec, across 10 repeated experiments, and column "Time" shows the average end-to-end active learning time.

Program	LOC	#T	#R	#Queries		Time (s)
				med.	max	
Floodlight stateless firewall	216	895	5	8	9	188
Floodlight stateful firewall	233	121	7	23	25	468
POX l3 stateless firewall	97	185	5	10	12	79
POX l3 stateful firewall	107	4,591	6	22.5	26	505
POX learning-switch	98	334	4	6	10	2,295

3.7.3 Learning from Program Traces

In our final experiment, we explore NetSpec's ability to directly synthesize from actual execution traces as input-output examples. The benefits of this approach are two-fold. First, for code refactoring or program analysis, the generated specifications expose the essential logic of the program, and can serve as a formal model

for further analysis. Second, the logic specifications can be compiled into a more compact and less bloated program for execution.

Trace collection. We collect program communication traces from two popular SDN platforms, POX [97] and Floodlight [58], on which we run controller programs and collect its communication traces with the switches in the network. We select SDN platforms as a basis for this experiment due to readily available open-source implementations that match our benchmarks.

To generate input-output examples, we generate representative traffic loads that we inject into each SDN controller program. Based on the inputs, we capture the outputs by observing the SDN programs. For instance, for learning switches, all hosts send probe packets to establish full connectivity in the network. For firewalls, we divide the network into two zones, one protected by the firewall, and the other serving as the external network. We then randomly pick hosts from either side of the network to establish TCP sessions. We validate the functionality of the firewall by checking that only sessions initiated from internal hosts are successfully established.

Trace collection is done by running the controller programs in both POX and Floodlight on the Mininet [88] emulator. All Mininet topologies are setup on a 16-node, 8-switch, tree topology network. For each run, we collect the controller's input-output traces as it interacts with Mininet switches (via incoming/outgoing packets and flow modifications). In all our experiments, we observe that this setup suffices to collect enough examples for NetSpec to learn a validated specification, with additional queries to user.

We implemented a trace collector on both POX and Floodlight that collects the controller's input and output messages at run-time and the state changes to the program. The state monitor works as follows. Within each application's input packet handler, we inspect all the accessible global variables. We then record any changes to such global variables. We exclude the known global constructs that are irrelevant to each application's execution logic, like loggers.

Table 3.5 summarizes the results. We make the following observations. First, NetSpec is able to correctly synthesize the intended specifications for all applications. Second, even though traces contain up to 4,500 communication messages, additional queries are needed to augment the examples. This observation shows the practicability of NetSpec's example augmentation mechanism in helping user uncover corner cases. Third, even for non-trivial SDN applications with hundreds of lines of code, NetSpec is able to generate compact specifications with less than seven rules. Finally, the synthesis times are in the order of hundreds of seconds, except "learning-switch" at 2,295 seconds, despite the need to analyze actual communication traces with up to 4591 examples, and generate multiple queries, indicating the efficiency and scalability of our approach.

3.8 Related Work

Programming by example. NetEgg [124] enables programming SDN policies by example timing diagrams. NetEgg demonstrates via actual user studies that a programming-by-example paradigm can result in higher programming productivity and fewer errors. The key distinction is that NetSpec synthesizes the actual control plane program in the target DSL, which generates the data plane configurations, whereas NetEgg directly generates the data plane configurations. This target DSL can be used to verify and check for errors in the control plane program, whereas NetEgg can only provide counter-examples to indicate that the input examples are incorrect. NetSpec mitigates one inherent weakness of NetEgg in its reliance on the user to provide all possible examples that meet the scenarios. Facon [45] is a programming-by-example tool for synthesizing SDN programs. NetSpec employs a more scalable synthesis strategy, targets a more general logical model, and can handle more complex protocols and incomplete examples.

Network configuration synthesis. Taking high-level routing policies as input, Net-

Complete [54] synthesizes BGP configurations that comply with these policies, and Genesis [112] synthesizes forwarding tables in multi-tenant networks. Avenir [40] synthesizes SDN data plane operations from high-level forwarding specifications. Propane [32] compiles high-level routing policies into distributed router BGP configurations. In contrast, NetSpec uses input-output examples, or execution traces from legacy programs, as input, and generate executable protocol specifications that are consistent with given input-output examples.

Config2Spec [35] takes network configurations and a failure model as input, and generates network policies that should hold for all possible concrete data planes derived from the given configurations and failure model. On the other hand, NetSpec generates executable specifications (analysis rules), instead of the static policies (facts derived from analysis rules). NetSpec can complement Config2Spec when analysis tools for the interested policies are not available. NetSpec takes the concrete data plane and the set of satisfied policies as input, and generates data plane analysis rules that can be applied to all concrete data planes. For example, in section 3.7.1, we show that NetSpec can generate reachability analysis rules similar to what is used by Config2Spec when inferring reachability policies.

Datalog and logic program synthesis. A large body of work has proposed techniques to synthesize logic programs [52] from input-output examples. With the exception of GenSynth, existing techniques require the user to syntactically constrain the search space by means of specifications such as mode declarations (e.g. ILASP [78]), meta-rules (e.g. Metagol [51]), candidate rules (e.g. ALPS [104] and ProSynth [98]), or templates (e.g. NTP [99] and δILP [56]). NetSpec does not require the user to provide any such specifications, but with the trade-off to require more input-output examples to fully specify an intended program. However, with the help of active-learning, as our evaluation demonstrates, NetSpec synthesizes more general programs than GenSynth, and is more efficient and robust.

Network verification and domain-specific languages. There is significant prior

work on network verification [73, 59, 33, 116, 117] and DSLs for networking [92, 60, 74, 30]. NetSpec generates a logical network specification that can be verified using existing techniques. Hence, verification should be viewed as a complementary technology to NetSpec. The same benefits of having a restricted language, such as scalable synthesis and automated example augmentation, apply to other DSLs as well.

3.9 Conclusion

NetSpec addresses a long-standing problem in network verification: the widening gap between formal models and actual implementations. As a step towards closing the gap, we have proposed a new *specification by example* (SBE) toolkit where users can build formal models of their network protocols from input-output examples either supplied by the network designer or extracted from a legacy implementation. Our synthesized models are declarative logic programs which are amenable to formal verification and even generation of distributed implementations.

Our initial forays and experimental results are promising. The SBE approach can efficiently synthesize a wide range of network protocols, and is robust to missing examples. NetSpec should be viewed as a first step towards understanding the SBE paradigm and its application in different domains of networking, with limitations in the size of synthesized specifications and complexities. In the future, we plan to explore how to synthesize more complex specifications, methods for parallelizing the synthesis algorithms to handle larger specifications, and how SBE can interact with different formal verification techniques.

CHAPTER 4

DECLARATIVE SMART CONTRACTS

4.1 Introduction

Smart contracts are programs stored and executed on blockchains. They have been used in a wide range of blockchain-enabled distributed applications to manage digital assets, including auctions [67], financial contracts [36], elections [85], trading platforms [94], and permission management [27]. Unfortunately, today's smart contracts are error-prone and this has led to significant financial losses resulting from attacks such as Dice2win [95], King of Ether [10], Parity Multisig Bug [113], Accidental [38] and DAO [9, 105].

Several analysis and verification techniques have been proposed for known vulnerabilities of smart contracts such as re-entrancy attack and transaction-order dependency over the past few years [115, 63, 96, 21, 115, 101]. However, when it comes to high-level properties that are specific to individual smart contracts, today programmers typically have to rely on hand-written assertions [3], which is hard to maintain and error-prone. For example, for a smart contract that manages digital tokens, one may want to make sure that all account balances add up to the total supply of tokens. To monitor this property during run-time, one has to instrument the code to maintain a state that keeps track of the sum of all account balances, and add assertions about their equivalence wherever either account balances or token supplies are updated.

To make smart contracts easier to analyze and verify, we introduce *DeCon*, a declarative programming language for smart contract implementation and property specification. DeCon is based on Datalog (Chapter 2), a logic programming

language. Datalog frees programmers from low-level implementation details, e.g. data structures, algorithms, etc., and allows them to reason about the contract on specification level, via inference rules [34]. In addition, such relational representation serves as a high-level abstraction of the contract, which enables efficient formal analysis and verification [107, 121].

A typical smart contract provides two kinds of interfaces: *transactions* and *views*. Transactions are function calls that alter the contract states, e.g., a token transfer that updates both sender and recipient balance. Views are read-only functions that return particular states of the contract, e.g., the balance of an account.

Smart contract properties and operations can be naturally mapped to relational logic. For example, transactions, the main element in smart contracts, can be modeled as relational tables, where table schema contains transaction parameters, e.g., sender, recipient, and amount. Similarly, the balance of each account can be expressed as sum aggregation on transaction records and looking up an account balance can be expressed as a constraint on the address column of the balance table.

Given this relational view of transactions, committing a transaction can be interpreted as appending a new row to the corresponding table. The commit and abortion logic of a pending transaction is specified by declarative rules based on Datalog. Views can be specified as declarative queries on these tables. For example, an account balance is its total income subtracted by total expense, each of which is a query on relevant transaction records.

Contract properties are also specified as inference rules. They are interpreted as property violation queries, a special kind of views, and are expected to be always empty during correct executions. For example, if a smart contract forbids overspending, then a query on accounts with negative balances should always be empty. Such unification of implementation and property specification language saves programmers' effort to learn another language to formally specify properties.

DeCon complies declarative specifications into executable Solidity [2] programs

that runs on blockchains, e.g., Ethereum, and monitors the specified properties at run-time. When a property (violation) view is derived non-empty after executing a pending transaction, the transaction is aborted. Such automatic code generation not only saves implementation effort, but also eliminates the gap between the program specification and implementation, providing stronger guarantee on the verification result.

The key insight to generate efficient executable code from declarative specifications is that smart contract transactions are executed in sequence. In other words, new rows are appended to the transaction tables one at a time. Therefore, DeCon borrows the idea of incremental view maintenance in databases [66] to generate efficient update procedure. On committing a new transaction, instead of evaluating the queries on the whole tables, only the difference in query results are computed and applied to existing views.

In addition, DeCon is easy to debug with data provenance [39]. Provenance is a mechanism for explaining how certain tuples or facts are derived, right down to the input values. In an imperative language like Solidity [2], dependency information is difficult to be captured automatically (through data-flow analysis). In contrast, inference rules in DeCon give explicit dependency information, where each tuple can be directly attributed to one rule, thus providing more clarity to the execution process.

The key contributions of this chapter are as follows:

- We design DeCon, a declarative language that unifies smart contract implementation and specification. We demonstrate its expressiveness via case studies on representative smart contracts and their high-level correctness properties.

- We design an algorithm to compile these high-level specifications into executable Solidity programs, with instrumentation for run-time verification.

- We implement and experimentally evaluate DeCon. Evaluation shows that the generated executable code has comparable efficiency with equivalent open-source implementation of the same contract (14% median gas overhead), and the overhead of run-time verification is moderate (16% median gas overhead). The prototype implementation and evaluation benchmarks will be open-sourced for future studies and comparisons.

The rest of this chapter is organized as follows. Section 4.2 motivates DeCon using a wallet example. The declarative smart contract language is presented in Section 4.3. Section 4.4 demonstrates how to translate declarative rules into an executable Solidity program. The expressiveness of DeCon is demonstrated in Section 4.5 using two case studies. Section 4.6 experimentally evaluates DeCon. Section 4.7 discusses related work, and Section 4.8 concludes the chapter.

4.2 Illustrative example

In this section, we show how to use DeCon to implement a smart contract, specify its properties, and debug via provenance using a Wallet smart contract that manages digital tokens.

4.2.1 Contract Implementation

A smart contract offers two kinds of interfaces: *transactions* and *views*. Transactions are the function calls that update the contract states. Views, on the other hand, are read-only functions that return one or more contract states.

In declarative smart contracts, transaction records are the only states. Transactions are modeled as relational tables. A new row is appended to the table when a new transaction is committed, with column entries storing the transaction parameters. Transaction rules, i.e., the condition on which a new transaction can be

committed, are specified as declarative rules. Finally, views are specified as declarative queries over the transaction tables.

We use the Wallet example, shown in Listing 4.1, to explain how relational tables and declarative rules can be specified. The Wallet contract manages token transactions between Ethereum addresses, where the contract owner can mint or burn tokens to addresses, and different addresses can transfer tokens to each other.

```
// Transaction event triggers
.decl recv_mint(p:address, amount:int)
.decl recv_burn(p:address, amount:int)
.decl recv_transfer(from:address, to:address, amount:int)

// Views
.decl *totalSupply(n:int)
.decl balanceOf(p:address, n:int)[0]
.public totalSupply,balanceOf

// Transaction rules
.decl mint(p: address, amount: int)
.decl burn(p: address, amount: int)
.decl transfer(from: address, to: address, amount: int)

r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),n>0.
r2: burn(p,n):-recv_burn(p,n),msgSender(s),owner(s),balanceOf(p,m),
    n<=m.
r3: transfer(s,r,n):-recv_transfer(s,r,n),balanceOf(s,m),n>=m.

// View rules
r4: totalSupply(n):-allMint(m),allBurn(b),n:=m-b.
r5: balanceOf(p,s):-totalOut(p,o),totalIn(p,i),s:=i-o.
```

```
23
24 // Auxiliary relations and rules ...
25 .decl totalIn(p: address, n: int)[0]
26 .decl totalOut(p: address, n: int)[0]
27 r6: transfer(0,p,n) :- mint(p,n).
28 r7: transfer(p,0,n) :- burn(p,n).
29 r8: totalOut(p,s):-transfer(p,_,_),s=sum n:transfer(p,_,n).
30 r9: totalIn(p,s):-transfer(_,p,_),s=sum n:transfer(_,p,n).
31
32 .decl *allMint(n: int)
33 .decl *allBurn(n: int)
34 r10: allMint(s) :- s = sum n: mint(_,n).
35 r11: allBurn(s) :- s = sum n: burn(_,n).
```

Listing 4.1: Wallet smart contract

Relations and interfaces. Lines 1 to 14 declare the relations, with schema in the parenthesis, and, optionally, primary key indices in the bracket (e.g., `balanceOf` on line 8). Primary keys uniquely identify a row in the table. For instance, `balanceOf` records the balance of each account, and thus has unique account column. Without explicit specification, all columns are treated as primary keys. Relation `totalSupply` (line 7) is a singleton relation, a kind of relation that contains only one row and is annotated by a star symbol.

Given these relation declarations, transaction and view interfaces are generated. First, transaction interfaces are generated from relations with `recv_` prefix, where the the input parameters define the schema and a boolean return value indicates the success of the transaction. For example, relation `recv_mint` is translated into the following interface in Solidity, the target executable language:

```
    function mint(address p, int amount) returns (bool);
```

Second, view functions are generated from the relations that appear in the public

interface annotations (line 9). The input parameters are the primary keys, and output is the remaining values. Note that a singleton relation, e.g., `totalSupply`, becomes a function without parameters since it has no primary keys. If all columns are primary keys, then the function returns a boolean value indicating the existence of the row. For example, `balanceOf(p:address, n:int)[0]` is translated into the following function interface:

```
function balanceOf(address p) returns (int);
```

Rules and functions. The rest of the program shows the rules that process transactions and define the views. Each rule is of the form `<head> :- <body>`, interpreted as follows. For all valuation of the variables that satisfies all constraints in the body, generate a row as specified in the head. For example, `r1` on line 16 says that a `mint` transaction can only be sent by the contract owner, and the amount should always greater than 0. This rule is compiled into the following Solidity code (with simplification):

```
function mint(address p, int n) (returns bool) {
    bool ret = false;
    if (msg.sender == owner && n>0) {
        // call functions to update dependent views...
        ret = true;
    }
    return ret;
}
```

When a `mint` transaction is committed, `r5` will be triggered through a chain of rules (`r1->r6->r9->r5`). It specifies the balance of an account `p`, as the total income `totalIn(p,i)` subtracted by the total expense `totalOut(p,o)`, with `totalIn` and `totalOut` further defined by `r8` and `r9`, respectively. This rule is compiled into two Solidity functions, each updates `balanceOf[p]` when either `totalIn` or `totalOut` is updated.

```
function updateBalanceOfOnTotalIn(address p, int i) {
    int o = totalOut[p];
    balanceOf[p] = i-o;
}
function updateBalanceOfOnTotalOut(address p, int o) {
    int i = totalIn[p];
    balanceOf[p] = i-o;
}
```

To get the balance of a given account, one could call `balanceOf`, a view function that takes the account address as parameter, and returns an integer value as the account balance. In DeCon, relational tables are stored in maps, mapping primary keys to values in remaining columns. This view function is generated as follows.

```
function balanceOf(address p) public view returns (int) {
    // Read the row by primary key p
    BalanceOfTuple memory balanceOfTuple = balanceOf[p];
    // Return the value
    return balanceOfTuple.n;
}
```

4.2.2 Specification and Run-time Verification

In DeCon, properties are specified the same way as views, but with additional annotation in order for DeCon to know what to monitor at run-time. For example, in the wallet contract, one may want to make sure that all account balances are always non-negative, which can be specified as follows.

```
.decl negativeBalance(p:address,n:int)[0]
.violation negativeBalance
r14: negativeBalance(p,n) :- balanceOf(p,n), n < 0.
```

Rule `r14` specifies the violation instance of the property: for each row in `balanceOf` table with `n<0`, insert a row `(p,n)` in `negativeBalance` table. During the execution of the transaction, the `negativeBalance` table is incrementally updated when its dependent relations are updated, the same as other views.

A property is satisfied if and only if the *violation* table is empty. The keyword `.violation` annotates that every row in the `negativeBalance` table is a property violation instance. Given such annotations, DeCon instruments the program to check the emptiness of all violation tables before each transaction is committed.

Note that properties are monitored on the granularity of transactions. As we show in Section 4.4, due to the underlying update procedure, transient violations could occur during the execution of a transaction, but disappear at the end. Therefore, instead of aborting right after a violation tuple is derived, a transaction is only aborted if, at the end of its execution, any violation table remains non-empty. Such interpretation allows programmers to reason at the transaction level, without worrying about the underlying update procedure. The violation checking at the end of each transaction is performed using `checkViolations()` function.

```
function checkViolations() {
    if negativeBalance is not empty:
        revert("Negative balance.")
    // check other violations ...
}
```

4.2.3 Debugging via Provenance

Data provenance is a feature of declarative programs that records the data flow from input to output and enables rule-wise debugging.

Suppose the original program has an incorrect $r2$, which misses a predicate to check that the account has enough balance to be burnt. The incorrect version of `r2`

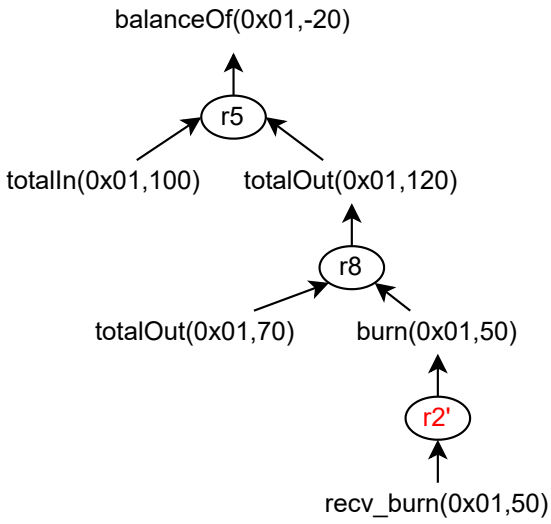


Figure 4.1: Provenance of a violation of negative balance

is shown as `r2'` in the following.

```
r2': burn(p,n):-recv_burn(p,n),msgSender(s),owner(s).
```

An account with balance n would have negative balance if more than n tokens are burnt. Suppose during an execution, the account $0x01$ is detected to have negative balance of -20. To understand why this violation happens, one could query the violation tuple's provenance tree, as shown in Figure 4.1. The provenance tree is read from top to bottom. On the top is a tuple `balanceOf(0x01,-20)` that triggers the violation in `negativeBalance`. Below shows that it is derived by `r5`, based on `totalIn(0x01,100)` and `totalOut(0x01,120)`, which are the total tokens received and sent by address `0x01`. The tuple `totalOut(0x01,120)` is further derived by `r8`. This back tracing continues for another step until one finds the derivation of `r2'` is incorrect, which suggests that the condition `balanceOf(p,m),m>=n` should be added. With this provenance, programmers can debug contracts in a visual and interactive manner.

4.3 Language

A DeCon contract consists of three elements: relations, rules, and relation annotations. A *relation* declaration specifies the name of a relational table and its schema. Each relational table can store either transaction records, with the transaction parameters being the column values, or views, the summary information of these transaction records. A *rule* specifies either the conditions on which a new transaction gets approved or the derivation of a view from the transaction records. Finally, *relation annotations* specify whether a relational table is a public view or a violation. Public views are compiled into public interfaces that take the relation's primary keys as parameters, and return the remaining values in the matching row. Violation will be monitored during run-time, and a transaction is reverted if the violation relation is non-empty after the transaction execution.

4.3.1 Relation Declarations and Annotations

The formal syntax of relation declarations and annotations is defined in Figure 4.2. It is based on the Datalog language (Chapter 2), with special language extensions for smart contracts.

Schema. Schema of a relation is specified as a list of $c_i : T_i$, where c_i is the column name for the ith column, and T_i is the data type.

Primary keys. Primary keys K are a list of indices in the relation schema. Primary key specification is optional. If a simple relation is specified without primary keys, then all columns are treated as primary keys. Primary keys uniquely identify a row in each table. On inserting a new row, if an existing row has the same primary key, the existing row is replaced by the new row.

Singleton relations are relations with only one row, which are annotated with $*$ in the specification. When a new row of a singleton relation is inserted, it replaces the existing row.

$$
\begin{aligned}
(Type)\ T &:= int\ |uint\ |bool\ |address \\
(Schema)\ S &:= c1 : T1, c2 : T2, ... \\
(Primary\ keys)\ K &:= k1, k2, ... \\
(Reserved\ relation)\ RS & \\
(Singleton\ relation)\ SG &:= .decl\ *r(Schema) \\
(Simple\ relation)\ SP &:= .decl\ r(Schema)[K] \\
(Transaction\ relation)\ TR &:= .decl\ recv\_[r](Schema) \\
(Relation)\ R &:= RS|SG|SP \\
(Annotation)\ A &:= .public\ R\ |\ .violation\ R
\end{aligned}
$$

Figure 4.2: Syntax of relation declarations and annotations

Transaction relations are relations with prefix `recv_`. As explained in the next section, when used in a rule, these relations are treated as an event trigger, and are compiled into smart contract interfaces that handle incoming transaction requests.

Reserved relations. The following relations are reserved to handle smart contract specific constructs:

- `msgSender(a:address)` stores the address of message sender.
- `msgValue(v:uint)` stores the values of Ethers sent along a message.
- `send(to:address, n:uint32)` triggers a transaction that sends n token to another account.
- `constructor(*)` is translated into the constructor function, with schema being function parameters.

$$
\begin{aligned}
(Variable)\ x & \\
(Aggregation)\ Agg &:= sum \mid max \mid min \mid count \\
(Function)\ F &:= + \mid - \mid \times \mid \div \\
(Condition)\ C &:= > \mid < \mid >= \mid <= \mid == \mid != \\
(Transaction\ relation)\ TR &:= recv\_[r] \\
(Other\ relation)\ R & \\
(Head\ literal)\ h &:= R(\bar{X}) \\
(Body\ literal)\ b &:= R(\bar{X}) \mid C(\bar{X}) \mid y = F(\bar{X}) \mid y = Agg\ x : R(\bar{X}) \\
(Transaction\ Rule)\ Tx &:= h\ :-\ TR(\bar{X}), b_1, ..., b_n \\
(View\ Rule)\ V &:= h\ :-\ b_1, ..., b_n
\end{aligned}
$$

Figure 4.3: Syntax of rules

4.3.2 Rules

As shown in Figure 4.3, we distinguish two kinds of rules: transaction rules and view rules. A *transaction rule* contains a transaction relation in its body. Transaction relations are relations with a prefix $recv\_$ in names. These rules are only fired on receiving the corresponding transaction request, and the transaction is approved if rest of the constraints in the rule body are satisfied. A *view rule*, on the other hand, does not contain any transaction relations. It is evaluated whenever one of the relations in the body is updated.

Syntax restrictions. DeCon does not support recursions. That is, no dependency loop exists between any two relations. The dependency relationship in DeCon is defined as follows.

Definition 4.3.1 (Relation dependency)**.** *Relation R_a is dependent on relation R_b, if there exists a view rule where R_a is in the head and R_b is in the body, or a transaction rule where R_a is in the head, and R_b is a transaction relation (with a prefix `recv_`) in the body.*

Rule semantics. A rule is evaluated as follows. For each variables valuation π that satisfies the *rule constraint*, generate the head tuple with all variables assigned to its corresponding values in π. A variable valuation is a mapping from the set of variable names V to the variable domain D ($\pi : V \to D$). Rule constraint is a conjunction of all body literal constraints. As described in Figure 4.3, there are four kinds of body literals. For literals in the form of relational tuples $R(\bar{X})$, the constraint is satisfied if row $\bar{X}$ exists in the relational table R. Other kinds of literals (i.e., conditions, functions, and aggregations) are directly interpreted as constraints on the variables.

Take `r5` in the wallet example (listing 4.1) for instance.

```
r5: balanceOf(p,s):-totalOut(p,o),totalIn(p,i),s:=i-o.
```

This rule is interpreted as follows: "for all values of variable p, o, i such that there exists a tuple $totalOut(p, o)$ and $totalIn(p, i)$, derive the head tuple $balanceOf(p, s)$, where $s = i - o$".

Aggregation literal $Agg\ x : R(\bar{X})$ computes the aggregate for all rows in relation R that satisfy the rule constraint. For example, in the wallet example (listing 4.1), line 31 shows a rule with aggregation.

```
r8:totalOut(p,s):-transfer(p,_,_),s=sum n:transfer(p,_,n)
```

For each unique value of p in the first column of `transfer` table, this rule computes the sum of the third column for rows in `transfer` table that has the value p in the first column. In other words, this rule groups the table by the first column, and then computes the sum of the third column within each group.

4.4 Compilation to Solidity

DeCon translates a set of declarative rules into an executable Solidity program that (1) processes transactions following the conditions in transaction rules, (2) updates views incrementally as new transactions are committed, and (3) monitors property violations.

The compilation process involves three major steps.

(1) Abstract update functions. First, each rule is translated into a set of abstract update functions, each of which performs incremental updates to the head relation when one of the body relations is updated. These functions are abstract in that they do not implement concrete data structures. For example, recall that in the wallet contract in Section 4.2, the following rule processes `mint` transactions:

```
r1: mint(p,n):-recv_mint(p,n),msgSender(s),owner(s),n>0.
```

It is translated into the following abstract update function:

```
1   on insert recv_mint(p,n) {
2     search owner where {
3       address s = owner.p;
4       search msgSender where s==msg.sender {
5         if(n>0) {
6           insert mint(p,n)
7         }}}}
```

This update function is triggered when a `mint` transaction is received, as indicated by the event trigger tuple `recv_mint(p,n)`. The remaining two relational literals, `owner(s)` and `msgSender(s)`, are translated into nested search statements (line 2 and line 4). A search statement has the form `(search R where C do S)`, where R is the relational table, C is the set of constrains on rows, and S is the statement to execute for each row that satisfies the constraints in C. The condition literal (`n>0`) is translated into an `if` statement (line 5). If all prior conditions are satisfied, we

arrive at line 6, where the rule head is inserted.

(2) Data structures. These abstract functions are then translated into concrete Solidity statements, where the `search` statements become efficient join algorithms on concrete data structures, and update functions for dependent views are called after an `insert` statement.

(3) Instrumentation. In the last step, the Solidity program is instrumented to monitor property violations, and abort the transaction if any violation has been detected by the end of transaction execution.

4.4.1 Abstract Update Function Generation

There are two kinds of updates that could trigger a rule: tuple insertion and tuple deletion. We use $Insert(e)$ and $Delete(e)$ to denote the update trigger on inserting and deleting a tuple e, respectively. Note that both a tuple and a literal have the form $R(\bar{X})$. It is called a tuple when $\bar{X}$ has concrete values, and called literal in a rule, where $\bar{X}$ is symbolic. In the following discussion of update triggers we use literal and tuple interchangeably.

Given a rule r, let $B(r)$ be the set of all relational literals in r's body, and e be the transaction relation in r if r is a transaction rule, the set of update triggers $T(r)$ are defined as:

$$T(r) := \begin{cases} \bigcup_{l \in B(r)} \{Insert(l), Delete(l)\} & r \text{ is View} \\ \{Insert(e)\} & r \text{ is Tx rule} \end{cases} \tag{4.1}$$

If r is a view rule, then it can be triggered by updates of any relation in its body. Otherwise, r is a transaction rule, and it is only triggered on receiving a transaction request.

For each rule r, and for each update triggers in $T(r)$, an abstract update function is generated by $\mathrm{UpdateFunction(r, t)}$, presented in algorithm 5. It first initializes the set of grounded variables by variables in the trigger literal. Grounded variables

Algorithm 5 $\mathrm{UpdateFunction}(r, t)$. Given a rule r, and a trigger t, returns an update object.

1. Initialize the set of grounded variables $G \coloneqq t.variables$.
2. Literals other than the trigger $L \coloneqq \{r.body \setminus t\}$.
3. Update procedure $S \coloneqq \mathrm{Update}(r.head, L, t, G)$.
4. Return (on t do S)

are variables that are constrained to a constant value. Variables in a trigger literal are considered grounded because the update function is always triggered by the insertion or deletion of a concrete tuple. In step(3), update procedure S is generated by a sub-routine Update, which is presented in algorithm 6. Finally, it returns the abstract update function in the form of (`on t do S`), where t is the update trigger and S is the update procedure.

As shown in Algorithm 6, $\mathrm{Update(h, L, t, G)}$ performs recursion on L, the list of literals in the rule body, with every recursion translating one literal to a layer of code block, nested within the code block generated by the previous literals.

In particular, it performs pattern matching on input L, a list of literals to be translated. If L is empty, which means all body literals have been translated, an update statement consistent with the update trigger is returned. Otherwise, L has the form $head :: tail$. It first adds all variables in $head$ into the set of grounded variables, and then generates the inner code blocks S by recursively calling itself on $tail$ and the updated set of grounded variables G'. Depending on the form of $head$, the current layer of code block is generated in different ways. By the syntax of the language in Section 4.3, $head$ could take one of the following forms:

- A relational literal $R(\bar{X})$. Given the set of grounded variables G, the search con-

Algorithm 6 $\mathrm{Update}(h, L, t, G)$. Given a rule head h, a list of body literals L, an update trigger t, and the set of grounded variables G, return statements that perform the incremental update.

match L:

  case Nil => **match** t

    case Insert => **return** Insert(h)

    case Delete => **return** Delete(h)

  case $head :: tail$ =>

    Add grounded variables $G' \coloneqq G \cup \{x | x \in head\}$

    Inner statements $S \coloneqq \mathrm{Update}(\mathrm{h}, \mathrm{tail}, \mathrm{t}, \mathrm{G'})$

    match head:

      case $R(\bar{X})$ =>

        Derive constraints $C \coloneqq \mathrm{Constraint}(R(\bar{X}), G)$

        return (Search R where C do S)

      case $C(\bar{X})$ => **return** (If C Then S)

      case $y = F(\bar{X})$ => **return** ($y = F(\bar{X}) :: S$)

      case $y = Agg\ x : R(\bar{X})$ =>

        return ($y = Agg\ x : R(\bar{X}) :: S$)

straints for rows in R is generated as follows.

$$Constraints(R(\bar{X}), G) \coloneqq \bigwedge\{(R[i] == v) | v \in G, v \in \bar{X}, \\ i = \bar{X}.\,\mathrm{indexOf}(v)\}$$

where $R[i] == v$ means filtering rows in table R whose i-th column equals to v.

- A condition literal $C(\bar{X})$, in which case, the condition is directly used in the same way as an If condition, with the inner code block S placed within the If statement.
- A function or aggregation. In either case, the literal is directly translated into an assignment statement, followed by the inner code block S.

Aggregations. The evaluation results of aggregation functions are maintained incrementally. Sums are incremented by n when a row with aggregate value n is inserted, and decremented by n when a row is deleted. Similarly, counts are incremented by 1 on row insertion, and decremented by 1 on row deletion. Maximums and minimums are slightly different. When a new row is inserted with value n, if n is greater than the current maximum, the maximum is updated to n. When the current maximum row is deleted, the maximum is updated as the second maximum value. Thus, it requires maintaining a sorted list of values. Minimum is maintained in a similar fashion.

4.4.2 Concrete Data Structures and Instructions

Given the abstract functions generated from each rule, the next step is to generate concrete and efficient data structures and search algorithms in the Solidity language.

Data structures. Each relational table R, except singleton relations, is translated into a mapping from its primary keys to a structure that stores the rest of the column values:

```
struct RTuple {
    bool valid;
    T1 field1;
    T2 field2;
    ...
};
mapping(k1 => k2 => ... => kn => RTuple) R;
```

Note that hash-maps in Solidity by default map all keys to zero. Therefore, a valid bit (`valid`) is introduced to indicate the existence of a tuple. Columns other than primary keys are the structure members. If all columns are primary keys, then its structure only contains a valid bit.

Singleton relations are directly stored in a structure with columns being the structure members.

Join index. Join index is built for each search statement in the abstract update program. Given a search statement `Search R where C do S` in the abstract update program, if all primary keys of R are constrained to constant values, no join index is generated. The matching entry can be directly looked up by primary keys.

On the other hand, if, in some rule, not all primary keys of R are constrained to constant values, a join index is built as a map from the constrained keys to a list of unconstrained keys.

Suppose relation $R_1(k1, k2, v1)$ has two primary keys $k1$ and $k2$. As described above, table $R1$ is stored as a map from primary keys to remaining values (`mapping(k1=>k2=>R1Tuple`
Given a search statement `Search R1 where R1[0]==k1 do S`, where only one primary key $k1$ is constrained, the join index for R_1 is built as the following.

```
struct R1KeyTuple {
    bool valid;
    T2 k2;
```

```
    }
    mapping(k1 => R1KeyTuple[]) R1Index;
```

where `R1Index` maps $k1$ to a list of $R1KeyTuple$, which stores value of the other primary key $k2$. During the join execution, to iterate all rows in R_1 that satisfy $R_1[0] == k1$, it first looks up all $k2$ in `R1KeyTuple[k1]`, and then for each $k2$, get the value in `R1[k1][k2]`.

Update dependent views. An insert or delete statement in the abstract update function is translated into two sets of Solidity instructions: (1) update the corresponding data structure; and (2) call the update functions for the dependent relations (Definition 4.3.1).

Inserting a relational tuple t_1 directly updates the map, as well as the join index if one exists. If a tuple t_0 with the same primary keys exists, all dependent views are updated by first calling deletion updates on t_0, and then the insertion updates on t_1. Insertion update refers to functions triggered by tuple insertion, and deletion update refers to functions triggered by tuple deletion. Otherwise, insertion updates are directly called. Since a Solidity mapping maps all keys to value zero by default, a tuple exists if its valid bit is set to true.

Deleting a relational tuple resets its valid bit to false. Then deletion updates are called for all dependent relations.

In this way, when a new transaction is committed, all dependent views are updated through this chain of update propagation. Since there is no recursion, i.e., dependency loop between relations, allowed in DeCon, update propagation is guaranteed to terminate.

Logging. Every committed transaction is logged via events [5] in Solidity. These logs constitute all states of a DeCon contract, which enables offline analysis for further insights and potential bugs.

4.4.3 Run-time Verification

Properties are specified as declarative rules that derive violation instances. Such relations are annotated with the keyword `violation`.

Note that properties are checked on the granularity of transactions, which means transient violations that occur during the transaction execution but disappear at the end are not counted as violations. Such transaction-level property interpretation allows programmers to reason about the contract at the transaction level, without worrying about the underlying update procedure.

As an example of transient violations, consider again the wallet contract in Section 4.2, and a property that all account balances add up to the total supply. The property can be specified as shown in Listing 4.2.

```
.violation unequalTotalSupply
r12: totalBalance(s) :- sum n: balanceOf(_,n).
r13: unequalTotalSupply(n,m):-totalSupply(n),
                              totalBalance(m),n!=m.
```

Listing 4.2: All account balances add up to total supply.

During the execution of a `mint` transaction, the `totalSupply` and the `totalBalance` are updated in sequence, which leads to a violation when one is updated before another, but the violation disappears when both are updated.

Given this notion of transient violations, instead of aborting the transaction right after a violation tuple is derived, the checking procedure is deferred to the end of transaction execution. If any violation view is non-empty, the transaction is aborted. Note that a Solidity mapping do not record its size. So a separate array of mapping keys are maintained and iterated for valid violation tuples.

4.4.4 Provenance generation

To debug a violation, programmers can use data provenance to visualize the derivation process of a violation tuple. As shown in Figure 4.1, provenance is a directed graph with two kinds of vertices: tuples and rules. Edges from a tuple vertex to a rule vertex denote tuple reads, and edges from a rule vertex to a tuple vertex denote tuple derivations.

To generate this provenance graph during the rule evaluation procedure, two kinds of additional records are logged: tuple read `Read(tuple, rid)` and tuple derivation `Write(rid,tuple)`, where `rid` is a unique identifier for each rule. `Read(tuple,rid)` is interpreted as an edge from `tuple` to the rule indexed by `rid`, and, conversely, `Write(rid,tuple)` is an edge from rule `rid` to `tuple`.

Note that in Solidity, a failed transaction reverts all instructions, including logging. When a transaction is reverted due to a property violation, the provenance logs would also be reverted. Therefore, to generate provenance for a violation tuple, the transaction needs to be executed in a local debugging environment instead of the deployment blockchain. This practice also saves storage space on the public blockchain.

4.4.5 Optimizations

To improve gas and storage efficiency, two optimizations have been applied to the generated codes.

Join order. Body literals in a rule are sorted by their iteration cost in an increasing order. First are reserved relations and singleton relations, since they need no iteration. Second are the relations whose primary keys have all appeared in proceeding literals. These literals can be searched via a direct mapping look-up, thus requiring no iterations either. Next are the rest of the relations, which are translated into loops. Finally come condition and function literals.

Storage space. Storage space on a blockchain is precious due to the high synchronization cost. Deriving relations on-demand, that is, delaying evaluating an inference rule until it is used, can save storage space, but may incur performance overhead. To achieve a balanced trade-off between time and space, DeCon only proactively derives and stores relations annotated as public views or violations, as well as relations that are read during their derivation. Other relations are derived on-demand. For example, in the wallet contract in Section 4.2, relation `mint` only serves as an update trigger for dependent rules, which is never queried during the update of public views or violations. Therefore, when a `mint` tuple is generated by `r1`, it only triggers the update for dependent rules, but it is not written to the persistent storage.

4.5 Case studies

In this section we demonstrate the expressiveness of DeCon and the explainability of data provenance via case studies on two typical smart contracts. For brevity, only a subset of rules are discussed. [1]

4.5.1 ERC20

ERC20 [57] is a token standard for fungible tokens. Similar to the wallet contract in Section 4.2, it also supports token transfers between users. In addition, it specifies an allowance mechanism, where users can allow other users to transfer their tokens, up to the amount set by the account owner. The allowance mechanism can be specified as follows.

```
r1: transferFrom(sender,receiver,spender,n) :-
        recv transferFrom(sender, receiver, n),
        /* Sender has enough balance. */
```

[1] The full declarative implementations are in presented in the supplementary files.

```
        balanceOf(sender,m), m>=n,
        /* Operator has enough allowance. */
        msgSender(spender),
        allowance(sender,spender,l),l>=n.
```

On receiving a `transferFrom` transaction, in addition to checking that the owner has enough balance (`m>=n`), the rule also requires the message sender to have enough allowance to spend tokens on `owner`'s behalf. The relation `transferFrom` represents transactions where the `spender` sends n tokens from the `owner` to the `recipient`.

The allowance of a `spender` on an account is specified as follows.

```
r2: spentTotal(o,s,m) :- transferFrom(o,_,s,_),
                  m = sum n: transferFrom(o,_,s,n).
r3: allowance(o,s,n) :- allowanceTotal(o,s,m),
                     spentTotal(o,s,l), n := m-l.
```

The relation `spentTotal` accounts the amount of tokens `m` that spender `s` has spent on behalf of the owner `o`. And `allowance` is derived by subtracting the total spending from the total allowance, an amount approved by the owner (defined in another rule).

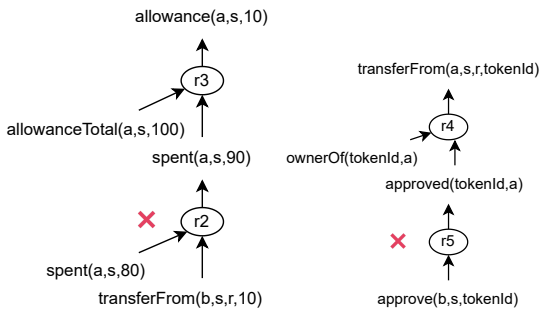
Given the definition of `allowance` and the `spentTotal`, we can specify a property that a spender never overspends as the following:

```
.violation overSpent
overSpent(o,s,n,m) :- allowanceTotal(o,s,n),
                    spentTotal(o,s,m), m>n.
```

DeCon then generates instrumentation to monitor this property at run-time.

Explain allowance changes via data provenance. Suppose the programmer made a mistake in specifying `spentTotal`:

```
r2': spentTotal(o,s,m):-transferFrom(o,_,s,_),
                  m = sum n: transferFrom(_,_,s,n).
```



(a) allowance(a,s,10) (b) transferFrom(a,s,r,tokenId)

Figure 4.4: Provenance tree for tuples.

The error is in the `sum` literal, where `transferFrom` records should be grouped by both owner `o` and spender `s`, instead of just the spender.

When a spender account s wants to transfer 20 tokens from account a to b, by submitting a transaction $\mathrm{transferFrom(a, s, r, 20)}$, it is reverted. DeCon explains that it is because the condition $l >= n$ in $r1$ is false, which means the spender s does not have sufficient allowance to transfer tokens on a's behalf.

To understand why the `spender` only have 10 allowance to a's account, one could get the provenance of the tuple `allowance(a,s,10)`, as shown in Figure 4.4a. On top of the provenance tree, `allowance(a,s,10)` is derived by `r3`, from the fact that the total allowance is 100 (`allowanceTotal(a,s,100)`), and that `a` has spent 90 already (`spent(a,s,100)`). To see why `spent(a,s,100)` is derived, the programmer continues expanding its provenance tree. A bug is revealed at this step, where a transaction from address b to r is accounted for s's allowance on address a, which points to the bug in $r2'$.

4.5.2 ERC721

ERC721 [55] is a smart contract standard for non-fungible tokens (NFTs). A main transaction for ERC721 tokens is `transfer`, which records the transfer of a token from `sender` to `recipient` at a particular `time`. The transaction time is included to specify the following views.

First is the view function `ownerOf`. Given the `transfer` relation, the owner of a token is defined as follows.

```
latestTransfer(tokenId,s,r,t) :- transfer(tokenId,s,r,t),
                    t = max s: transfer(tokenId,_,_,s).
ownerOf(tokenId, p):-latestTransfer(tokenId,_,p,_),p!=0.
```

where the first rule selects the latest transfer record for `tokenId`, and the next rule specifies that if the recipient of the latest transfer is non-zero, it is the owner of the token.

Next is the `exist` relation. A token exists if it is minted and is not burnt. In ERC721 contracts, burning a token emits a transfer record from its owner to zero address. So `exist` is defined as:

```
exists(tokenId, true) :-
    latestTransfer(tokenId,_,to,_), to!=0.
```

The rule checks that a token's latest transfer recipient is a non-zero address, which means it is not burnt.

To ensure every existing token has an owner, we could specify the following property:

```
.violation tokenNoOwner
tokenNoOwner(tokenId) :- ownerOf(tokenId,o), o==0.
```

which defines a property violation as entries in the `ownerOf` table with 0 owner address.

Explain an unexpected token transfer via data provenance. Suppose the owner wants to understand why one of her tokens has been transferred away in a transaction `transferFrom(a,s,r,tokenId)`, she expands the provenance tree for the transaction, which is shown in Figure 4.4b. On top of the provenance tree is a `transferFrom` tranaction, approved by the following rule:

```
r4: transferFrom(sender, receiver, tokenId) :-
        recv transferFrom(sender, receiver, tokenId),
        /* Sender owns the token. */
        ownerOf(sender,tokenId)
        /* Operator is approved to move the token. */
        msgSender(operator), approved(tokenId,operator).
```

where $approved(tokenId, operator)$ means that the token `tokenId` has been approved to use by `operator`. This approval is set by the token owner.

Suspicious about the `approved(tokenId,a)` tuple, the owner continues to expand the provenance tree, and finds that it is derived from the following rule:

```
r5: approved(tokenId,operator) :-
                    approve(_,operator,tokenId).
```

and the tuple `approve(b,s,tokenId)`, which means account `b`, a previous owner, has approved this token to operator `s` before transferring this token to `a`. Here, she finds the bug: `r5` does not check that the address that approves the token should be the token owner. The rule should have been updated as follows instead:

```
r5': approved(tokenId,operator) :- owner(tokenId, owner),
        approve(owner,operator,tokenId).
```

Table 4.1: Overhead of Solidity programs generated by DeCon, compared to reference implementations. Column $\#Rules$ shows the number of rules in the declarative smart contracts.

Contract	# Rules	Byte-code size (KB)		Transaction	Gas cost (K)		
		Ref.	DeCon		Ref.	DeCon	Diff
Wallet	12	3	3	mint	36	62	70%
				burn	36	47	29%
				transfer	52	38	-26%
Crowdsale	11	4	3	invest	38	33	-12%
				close	38	47	25%
				withdraw	26	29	14%
				claimRefund	29	33	13%
SimpleAuction	13	2	4	bid	69	115	66%
				withdraw	24	47	101%
				auctionEnd	54	56	4%
ERC721	13	10	11	transferFrom	59	42	-28%
				approve	49	75	53%
				setApprovalForAll	27	27	2%
ERC20	18	5	6	transfer	52	55	6%
				approve	47	50	7%
				transferFrom	43	50	15%
						median:	14%

4.6 Evaluation

We implement a prototype compiler for DeCon in Scala that generates Solidity programs with instrumentation for run-time verification. We first evaluate the compiler by comparing its output, without instrumentation, with reference contract written in Solidity. Next, we evaluate the overhead of run-time verification on these contracts and their properties.

4.6.1 Overhead to reference implementations

Reference smart contracts. We collect five reference smart contract implementations from public repositories and prior research. Wallet is the example shown in Section 4.2. CrowdSale is from prior research paper [96]. Auction is from Solidity documentation [4]. ERC20 (fungible tokens) and ERC721 (non-fungible tokens) are two of the most popular kinds of smart contract deployed on Ethereum, [2] and we use the implementation from the OpenZepplin library [1].

Declarative smart contract implementation. We implement declarative counterparts for all reference contracts with the same interfaces and functionalities without instrumentation for run-time verification or provenance. These contracts consist of 10 to 18 rules (column *#Rules* in Table 4.1).

Although DeCon can specify all the high-level logic of the these contracts, we note that the generated Solidity code has the following difference from the reference implementations. First, the reference CrowdSale contract is implemented as two separate contracts. As DeCon does not yet support contract composition, the compiler outputs a stand-alone smart contract with all the functionalities. For the ERC721 contract, there is a `safeTransferFrom` interface, which wraps the `transferFrom` function with a check: if the recipient is also a smart contract, it should implement the `onERC721Received` interface. The current implementation of DeCon does not yet support such checking procedure, which relies on calling the built-in functions of Solidity, so this interface is omitted.

Measurement metrics. We measure two metrics: (1) the size of EVM byte-code deployed on the blockchain; and (2) the gas cost for each transaction. EVM byte-code is generated by the Truffle [8] compiler. To measure gas cost, we first deploy the smart contract on Truffle's local blockchain, and then populate the smart contract

[2] According to https://etherscan.io, at the time of writing this paper, there are about 502,000 ERC20 tokens and 50,000 ERC721 tokens on Ethereum.

states by sending transactions from N test accounts, which results in N entries in the contract states. Then we call each transaction interface again and record gas cost reported by Truffle. We find that N (10 to 1000) does not impact gas cost. This is because all contracts use hash-maps to store contract states. If the hash-collision rate is low, the number of instructions is constant to the size of the hash-map, and thus the gas cost remains constant. Therefore, we report the gas cost measured with $N = 10$.

Results. As shown in Table 4.1, the median gas overhead to reference implementation is 14% across 16 transactions, with 3 of them have even lower gas cost between -28% to -12%. In the extreme case, the `withdraw` transaction from `SimpleAuction` shows 101% gas overhead.

In terms of byte-code size, DeCon's compiler output is slightly greater than the reference programs, with a 2KB (`SimpleAuction`) maximum increase. Note that on `CrowSale`, DeCon's output is smaller than the reference contract. This is because the reference implements two separate contracts, while the program generated by DeCon compiler has all functions implemented in one contract.

Contract features that are not yet supported. During the search of benchmarks, we find some contracts use features that are not yet supported by DeCon. For example, the voting contract from Solidity documentation [7] checks voting loop in a recursive manner. Although recursion can be naturally expressed in DeCon language, the execution of recursion functions requires non-trivial reasoning to ensure termination and gas efficiency, and is therefore not yet supported by DeCon. In addition, certain functions that lie outside of relational logic, including checking interfaces of another contract (e.g. `safeTransferFrom` in ERC721), and cryptographic functions[6], are not yet supported, but they can be incorporated into DeCon via user-defined functions in the future.

Table 4.2: Run-time verification overhead. Column $Size$ and Gas show the overhead in byte-code size (KB) and gas cost (K) respectively, compared to the DeCon contract without instrumentation.

Contract	Property	Size	Transaction	Gas
Wallet	No negative balance	2	mint	14%
			burn	14%
			transfer	17%
Crowdsale	No missing funds	2	invest	50%
			close	24%
			withdraw	22%
			claimRefund	33%
Simple Auction	Refund once	2	bid	2%
			withdraw	60%
			auctionEnd	4%
ERC721	Every token has owner	1	transferFrom	5%
			approve	3%
			setApprovalForAll	8%
ERC20	Account balances add up to total supply	1	transfer	96%
			approve	13%
			transferFrom	109%
			median:	16%

4.6.2 Run-time verification overhead

We measure run-time verification overhead by first specifying properties for each contract, which are generated as instrumentation in the output Solidity program. These instrumented programs are then compared to DeCon programs without instrumentation, on byte-code size and gas usage.

Contract properties are specified as follows. First, as shown in the example in Section 4.2, the wallet contract is monitored for negative account balances. The Crowdsale contract allows participants to invest in a crowd funding project with a particular funding target. The property specifies that the total amount of raised fund should equal to all participants' investments. In simple auction, bidders transfer their fund on every bid, and get refunds when the auction is ended. A property specifies that every bidder can claim refund at most once. In ERC721, the property specifies that all existing tokens should have a valid owner (non-zero address). In ERC20, all account balances should add up to the total supply of tokens.

Results. Table 4.2 shows the overhead of run-time verification. Byte-code sizes are increased by no more than 2 KB. Gas usage overhead varies across different transactions, with the median being 16%. Wallet and ERC721 contracts show small overhead, where transaction gas consumption increases by no more than 17% and 8%, respectively. Crowdsale and Simple Auction contract come with larger overhead. The highest increase in their transaction gas usage are 50% and 60%. The ERC20 contract shows the biggest overhead, where the `transferFrom` transaction shows 109% increase.

4.7 Related Work

In this section, we survey several lines of research that are related to our work.

Run-time verification. Similar to DeCon, Solythesis [79] also specifies properties as invariants and generates instrumentation for run-time monitoring. It applies to

general smart contracts implemented in Solidity, whereas DeCon targets declarative contracts only. By restricting the scope on declarative contracts, both specification and monitoring can be performed in a more straightforward manner. Invariants become violation queries, where joins are analogous to existential quantifiers, and aggregations to universal quantifiers. Detection becomes query evaluation, which reuses the same procedure for contract execution.

SODA [47] is a framework for implementing generic attack detection algorithms. Unlike DeCon, where the monitoring procedure is automatically generated from specification, the detection algorithms in SODA are implemented manually.

Sereum [100] monitors reentrancy attacks online via taint analysis. Azzopardi et al. [28] monitors contract execution against legal contract logic. These two work targets specific vulnerabilities and properties on Solidity smart contracts, whereas DeCon monitors user-specified properties on declarative contracts.

Static analysis and verification. Static analysis has been applied to detect generic vulnerabilities such as reentrancy attacks [65, 80], integer bugs [114, 108], trace vulnerability [93], and event-ordering bugs [75]. Securify [115] translates the EVM byte-code into stratified Datalog, and checks vulnerability patterns using off-the-shelf Datalog solvers.

Alt et al. [23] translates Solidity program into SMT formulas and use off-the-shelf SMT solver to verify contract properties. Zeus [72] leverages abstract interpretations and symbolic model checking to verify correctness and fairness of smart contracts.

Symbolic execution [83, 115, 21, 76, 42, 89, 22, 96] is another popular technique for smart contract verification. Oyente [83] detects generic predefined vulnerabilities including reentrancy, transaction order dependency, mishandled exceptions, etc. Verx [96], on the other hand, allows programmers to specify contract-specific properties in temporal logic.

Fuzzing has also been applied to smart contracts. For example, ContractFuzzer [71]

tests smart contracts for security vulnerabilities. Echidna [62] generates tests that triggers assertion violations. ILF [69] and Harvey [123] focus on improving code coverage.

Unlike these work, DeCon monitors properties online, which incurs run-time overhead, but does not suffer from false-positives or false-negatives. In addition, DeCon targets declarative smart contracts, while these tools analyze Solidity or EVM byte-code. Although targeting different languages, the underlying verification techniques can also be applied to DeCon and benefit from its higher-level abstraction. We believe this is an exciting direction for future research.

Domain-specific languages. Scilla [102] is a intermediate-level language for smart contracts that offers type safety and support for verification. KEVM [70] defines the formal semantics of EVM, and has been used to verify contracts against ERC20 standards. In contrast, DeCon focuses on the high level abstraction of smart contracts and specification of contract-specific properties. BitML [31] is a high-level language for Bitcoin smart contracts. Based on process calculus, it translates contracts into Bitcoin transactions. DeCon, on the other hand, is based on relational logic and targets Ethereum smart contracts.

4.8 Conclusion and future work

We present DeCon, a declarative programming language for smart contract implementation and property specification. In DeCon, smart contracts are specified in a high-level and executable manner, thus providing opportunities for efficient analysis and verification, bringing clarity to transaction execution via data provenance. Contracts implemented in DeCon demonstrate comparable efficiency to open-source reference implementation. Furthermore, run-time verification adds moderate gas overhead.

Our initial experience with DeCon suggests a few exciting future directions.

First, we find interesting contracts that require additional language features, including contract composition, recursion, user-defined functions, etc. Second, there are extreme cases where DeCon compiler generates contracts with non-negligible overhead to the reference hand-written code. DeCon compiler needs further optimization to generate more efficient executable code. Third, to save the overhead of run-time verification, we can leverage the high-level abstraction of DeCon programs to perform static verification.

CHAPTER 5

SAFETY VERIFICATION OF DECLARATIVE SMART CONTRACTS

5.1 Introduction

This chapter introduces DCV, the safety verifier for DeCon smart contracts.

Most existing solutions for smart contract verification directly verify the implementation [17, 68, 84, 72, 96]. These solutions have worked very well on verifying transaction-level properties, e.g., pre/post conditions, integer overflow, etc. However, when it comes to contract-level properties, where an invariant needs to hold across an infinite sequence of transactions, these approaches suffer from low efficiency due to state explosion issues. Some solutions [61, 26] trade soundness for efficiency, verifying properties up to a certain number of transactions.

On the other hand, in model-based verification approaches, a formal model of the smart contract is specified separately from the implementation. Given such a formal model and the implementation, two kinds of verification are performed: (1) does the formal model satisfy the desired properties [91, 41]? (2) is the implementation consistent with the formal model [48]? This verification approach is more efficient, because the formal model typically abstracts away implementation details that are irrelevant to the verification task. However, a separate model needs to be written in addition to the implementation, and is typically in a formal language that is unfamiliar to software engineers.

In DCV, we propose an alternative verification approach based on DeCon (Chapter 4), an executable specification of smart contracts. A DeCon contract is a declar-

ative specification for the smart contract logic in itself, making it more efficient to reason about than the low-level implementation in Solidity. It is also executable, in that it can be automatically compiled into a Solidity program that can be deployed on the Ethereum blockchain. When verification is completed, the automatic code generation saves developers' effort to manually implement the contract following the specification. The high-level abstraction and executability make DeCon an ideal target for verifying contract-level properties.

We implement a prototype, DCV (DeCon Verifier), for verifying declarative smart contracts. DCV performs sound verification of safety invariants using mathematical induction. A typical challenge in induction is to infer inductive invariants that can help prove the target property. A key insight is that the DeCon language exposes the exact logical predicates that are necessary for constructing such inductive invariants, which makes inductive invariant inference tractable.

As another benefit of using DeCon as the verification target, DeCon provides uniform interfaces for both contract implementation and property specification. It models the smart contract states as relational databases, and properties as violation queries against these databases. Thus, both the smart contract transaction logic and properties can be specified in a declarative and succinct way. With DCV, developers can specify both the contract logic and its properties in DeCon, and have it verified and implemented automatically.

This chapter makes the following contributions.

- A sound and efficient verification method for smart contracts, targeting contract-level safety invariants that is based on a declarative specification language and induction proof strategy (Sections 5.3, 5.4).
- A domain-specific adaptation of the Houdini algorithm [77] to infer inductive invariants for induction proof (Section 5.4).
- An open-source verification tool for future study and comparison.

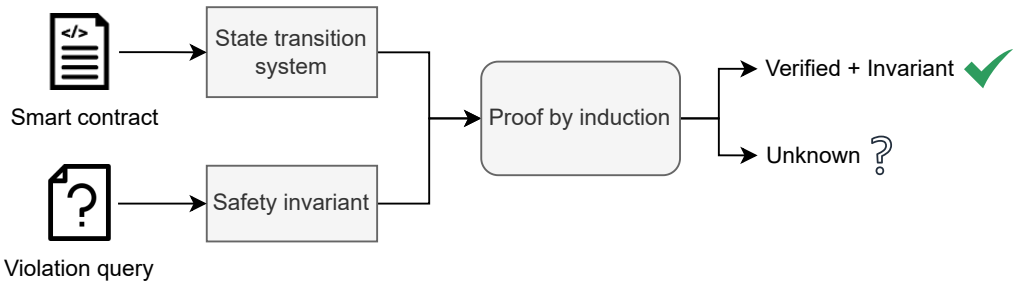


Figure 5.1: Overview of DCV.

- Evaluation that compares DCV with state-of-the-art verification tools, on ten representative benchmark smart contracts. DCV successfully verifies all benchmarks, is able to handle benchmarks not supported by other tools, and is significantly more efficient than baseline tools. In some instances, DCV completes verification within seconds when other tools timeout after an hour (Section 5.5).

5.2 Illustrative Example

Figure 5.1 presents an overview of DCV. It takes a smart contract and a property specification (in the form of a violation query) as input, both of which are written in the DeCon language (Section 4.3). The smart contract is then translated into a state transition system, and the property is translated into a safety invariant on the system states. DCV then proves the transition system preserves the safety invariant by mathematical induction. In our implementation, theorem proving is performed by Z3 [19], an automatic theorem prover.

If the proof succeeds, the smart contract is verified to be safe, meaning that the violation query result is always empty, and an inductive invariant is returned as a safety proof. Otherwise, DCV returns "unknown", meaning that the smart contract may not satisfy the specified safety invariant.

In the rest of this section, we use a voting contract (Listing 5.1) as an example to

illustrate the work flow of DCV. This example is adapted from the voting example in Solidity [16], simplified for ease of exposition.

5.2.1 A Voting Contract

Listing 5.1 shows a voting contract written in DeCon. In DeCon, transaction records and contract states are modeled as relational tables (lines 1-10). These declarations define table schemas in relational databases, where each schema has the table name followed by column names and types in a parenthesis. Optionally, a square bracket annotates the index of the primary key columns, meaning that these columns uniquely identify a row. For example, the relation `votes(proposal: uint, c: uint)[0]` on line 5 has two columns, named `proposal` and `c`, and both have type `uint`. The first column is the primary key. If no primary keys are annotated, all columns are interpreted as primary keys, i.e., the table is a set of tuples.

A special kind of relation is singleton relation, annotated by $*$. Singleton relations only have one row, e.g., `winningProposal` in line 8.

By default all relational tables are initialized to be empty, except relations annotated by the `init` keyword (line 12). These relations are initialized by the constructor arguments passed during deployment.

Each transaction is written in the form of a rule used in Datalog programs: `head :- body`. The rule body consists of a list of relational literals, and is evaluated to true if and only if all relational literals are true. If the rule body is true, the head is inserted into the corresponding relational table. For example, the rule in line 15 specifies that a `vote` transaction can be committed if there is no winner yet, the message sender is a voter, and the voter has not voted yet. The literal `recv_vote(p)` is a transaction handler that evaluates to true on receiving a `vote` transaction request. Rules that contain such transaction handlers (literal with a `recv_` prefix in the relation name) are called transaction rules.

Inserting a new `vote(v,p)` literal triggers updates to all its direct dependent

```
/* Declare relations. */
.decl recv_vote(proposal: uint)
.decl vote(p: address, proposal: uint)
.decl isVoter(v: address, b: bool)[0]
.decl votes(proposal: uint, c: uint)[0]
.decl wins(proposal: uint, b: bool)[0]
.decl voted(p: address, b: bool)[0]
.decl *winningProposal(proposal: uint)
.decl *hasWinner(b: bool)
.decl *quorumSize(q: uint)
.init isVoter
/* Transaction where voter v cast a vote to proposal p. */
vote(v,p) :- recv_vote(p), msgSender(v), hasWinner(false),
            voted(v, false), isVoter(v, true).
/* Count votes for each proposal p. */
votes(p,c) :- vote(_,p), c = count: vote(_,p).
/* A proposal wins by reaching a quorum. */
wins(p, true) :- votes(p,c), quorumSize(q), c >= q.
hasWinner(true) :- wins(_,b), b==true.
winningProposal(p) :- wins(p,b), b==true.
voted(v,true) :- vote(v,_).
/* Safety invariant: at most one winning proposal. */
.decl inconsistency(p1: uint, p2: uint)[0,1]
.violation inconsistency
inconsistency(p1,p2) :- wins(p1,true),wins(p2,true),p1!=p2.
```

Listing 5.1: A smart contract for voting, written in DeCon [46] langauge.

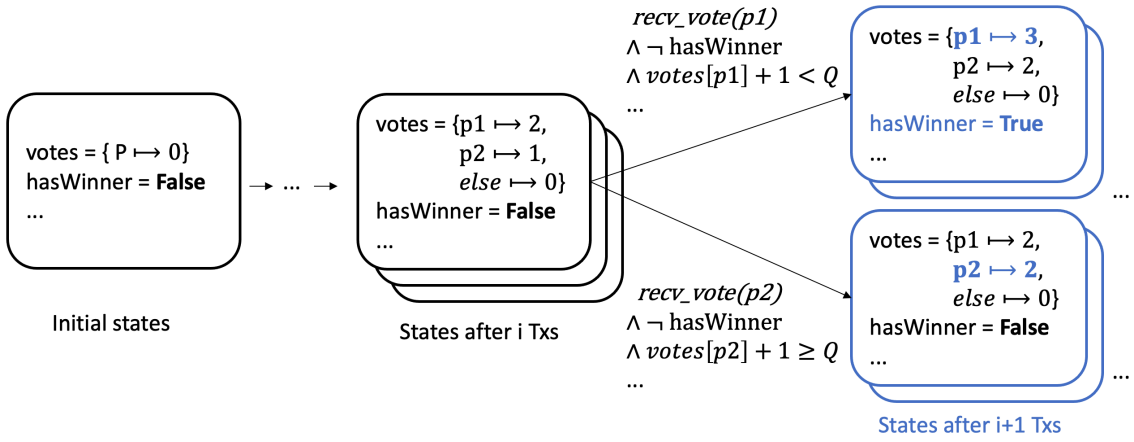


Figure 5.2: The voting contract as a state transition system.

rules. A rule is directly dependent on a relation R if and only if a literal of relation R is in its body. In this case, relation `wins` and `voted` are updated next. The chain of dependent rule updates go on until no further dependent rules can be triggered, and the transaction handling is finished.

On the other hand, if the body of a transaction rule is evaluated to false on receiving a transaction request, then no dependent rule is triggered, and the transaction is returned as failed.

Line 31 specifies a safety property in the form of a violation query. If the rule is evaluated to true, it means that there exists two different winning proposals, indicating a violation to the safe invariant that there is at most one winning proposal. Such violation query rule is expected to be always false during the execution of a correct smart contract.

5.2.2 Translating DeCon Contract to State Transition System

In order to verify the DeCon contract against the safety invariant, the declarative rules are translated into a state transition system. Figure 5.2 illustrates part of the transition system translated from the voting contract in Listing 5.1, where all relational tables are the states, and every smart contract transaction commit results

in a state transition step.

The middle portion of Figure 5.2 shows a state after i transactions from one of the initial states, where proposal p_1 has two votes, proposal p_2 has one vote, and there is no winner yet.

Two outgoing edges from this state are highlighted. Suppose the quorum size Q in this example is three. On the top is the transaction `vote(p1)`, where p_1 gets another vote, making it reach the quorum and become the winner. The edge annotates the conditions for this transaction to go through (only a fraction of the condition is shown in the figure due to space limit). It is translated from the transaction rule r in Listing 5.1 line 15 ($recv\_vote(p_1) \wedge \neg hasWinner \wedge ...$), as well as r's dependent rules from line 19 to 26 ($votes[p_1] \geq Q \wedge ...$). This edge leads to a new state where proposal p_1's votes is incremented by one, and it becomes the winner, which is also translated from line 19 to 26.

Similarly, the bottom right shows another transaction where proposal p_2 gets a vote, but $hasWinner$ remains $False$ since there is no proposal reaching the quorum.

Section 5.3.2 formally describes the algorithm to translate a DeCon smart contract into a state transition system.

Property. The violation query rule (line 31) is translated into the following safety invariant: $\neg[\exists p_1, p_2.\ wins(p1) \wedge wins(p2) \wedge p1 \neq p2]$, It says that there do not exist proposals $p1$ and $p2$ such that the violation query is true, which means that there is at most one winning proposal.

5.2.3 Proof by Induction

Given the state transition system translated from the DeCon smart contract, the target property $prop(s)$ is proven by mathematical induction. In particular, let S be the set of states in the transition system, and E be the set of transaction types. Given $s, s' \in S, e \in E$, let $init(s)$ indicate whether s is in the initial state, and $tr(s, e, s')$ indicate whether s can transition to s' via transaction e. The mathematical induction

is as follows:

$$\begin{aligned}
\text{ProofByInduction}(init, tr, prop) &\triangleq \text{Base}(init, prop) \wedge \text{Induction}(tr, prop) \\
\text{Base}(init, prop) &\triangleq \forall s \in S.\ init(s) \implies prop(s) \\
\text{Induction}(tr, prop) &\triangleq \forall s, s' \in S, e \in E.\ inv(s) \wedge prop(s) \\
&\quad \wedge tr(s, e, s') \implies inv(s') \wedge prop(s')
\end{aligned} \tag{5.1}$$

where $inv(s) \wedge prop(s)$ is an inductive invariant inferred by DCV such that $prop(s)$ is proved to be an invariant of the transition system.

To find such an inductive invariant, DCV first generates a set of candidate invariants using predicates extracted from transaction rules in the DeCon contract, and then applies the Houdini algorithm [77] to find the inductive invariant. The detailed steps are as follows:

(1) Extract predicates from all transaction rules. Take the transaction rule in line 15 as an example. The following predicates can be extracted from it: $\neg hasWinner$, $\neg voted[v]$, $isVoter[v]$.

(2) Generate candidate invariants. Given the extracted predicates, candidate invariants are generated in the form, $\forall x \in X.\ \neg init(s) \implies \neg p(s, x)$, where X is the set of all possible values of the local variables (variables other than the state variables) in predicate $p(s, x)$. And $p(s, x)$ is one of the predicates extracted from the transaction rules. $\neg init(s)$ is introduced as the premise of the implication so that the candidate invariant is trivially implied by the system's initial constraints. Having $\neg p(s, x)$ in the implication conclusion is based on the heuristics that in order to prove safety invariants, the lemma should prohibit the system from making an unsafe transition.

For example, the following invariant is generated following the above pattern:

$$\forall u \in Proposal.\ wins[u] \implies hasWinner \tag{5.2}$$

The invariant expresses that if any proposal $u \in Proposal$ is marked as winner, the

predicate $hasWinner$ must also be true.

(3) Infer inductive invariants. Given the set of candidate invariants, DCV applies the Houdini algorithm [77] and returns the formula in Equation 5.2 as an inductive invariant. Applying the inductive invariant inv to the induction procedure (Equation 5.1), the target property can be proven.

5.3 Program Transformation

5.3.1 Declarative Smart Contracts as Transition Systems

This section introduces the algorithm to translate a DeCon smart contract into a state transition system $\langle S, I, E, Tr \rangle$ where

- S is the state space: the set of all possible valuations of all relational tables in DeCon.
- $I \subseteq S$ is the set of initial states that satisfy the initial constraints of the system. All relations are by default initialized to zero, or unconstrained if they are annotated to be initialized by constructor arguments.
- E is the set of transaction types. Each element in E correspond to a type of transaction in DeCon (analogous to a transaction function definition in Solidity).
- $Tr \subseteq S \times E \times S$ is the transition relation, generated from DeCon rules. $Tr(s, e, s')$ means that state s can transit to state s' via transaction e.

In the rest of this section, we introduce the algorithm to generate the transition relation from a DeCon smart contract.

5.3.2 Transition Relation

The transition relation Tr is defined by a formula $tr : S \times E \times S \mapsto Bool$. Given $s, s' \in S, e \in E$, s can transition to s' in one step via transaction type e if and only if $tr(s, e, s')$ is true. Equation 5.3 defines tr as a disjunction over the set of formulas encoding each transaction rule. R is the set of rules in the DeCon contract. Γ is a map from relation to its modeling variable, e.g., the relation `vote(proposal:uint,c:uint)[0]` is mapped to $votes : uint \mapsto uint$. Recall from Section 4.3 that transaction rules are rules that listen to incoming transaction and is only triggered by the incoming transaction request. Therefore, $r.trigger$ is the literal with `recv_` prefix in r's body.

$$tr \triangleq \bigvee_{r \in TransactionRules} [\text{EncodeDeConRule}(r, R, \Gamma, r.trigger) \wedge e = r.TxName] \quad (5.3)$$

Algorithm 7 EncodeDeConRule(r, R, Γ, τ).

Input: (1) a DeCon rule r, (2) the set of all DeCon rules R, (3) a map from relation to its modeling variable Γ, (4) a trigger τ, the newly inserted literal that triggers r's update.

Output: A boolean formula over $S \times S$, encoding r's body condition, and all state updates triggered by inserting r's head literal.

1: $Body \leftarrow$ EncodeRuleBody(Γ, τ, r)
2: $Dependent \leftarrow \{\text{EncodeDeConRule}(dr, R, \Gamma, \tau') \mid (dr, \tau') \in \text{DependentRules}(r, R)\}$
3: $(H, H') \leftarrow$ GetStateVariable$(\Gamma, r.head)$
4: $Update \leftarrow H' =$ GetUpdate(H, r, τ)
5: $TrueBranch \leftarrow Body \wedge Update \wedge (\bigwedge_{d \in Dependent} d)$
6: $FalseBranch \leftarrow \neg Body \wedge (H' = H)$
7: **return** $TrueBranch \oplus FalseBranch$

The procedure EncodeDeConRule is defined by Algorithm 7. We explain it using the voting contract in Listing 5.1 as an example.

It takes four inputs: (1) a DeCon rule r, (2) the set of all DeCon rules R, (3) a map from relation to its modeling variable Γ, (4) and a trigger τ , the newly inserted literal that triggers r's update. In particular, a trigger τ takes the form `insert [literal]` or `delete [literal]`. It is used to inform the subroutine EncodeRuleBody how a relation is updated, and that the rest of the rule body in r needs to be encoded as a logical formula. For example, when a new `vote` transaction is received, the trigger τ is `insert recv_vote(p)`, where `p` is the transaction parameter.

In step 1, r's body is encoded as a boolean formula, $BodyConstraint$, by calling a procedure $EncodeRuleBody$ (Section 5.3.3). Take the rule for `vote` transaction in Listing 5.1 line 15 as an example. Its body is encoded as:

$$\neg hasWinner \wedge \neg hasVoted[v] \wedge isVoter[v]$$

Step 2 first selects direct dependent rules of r from the set of all DeCon rules R, by calling a subroutine $DependentRules(r, R)$. It returns a set of tuple (dr, τ'), where dr is a direct dependent rule of r, and τ' is the corresponding trigger for dr. A rule dr is directly dependent on rule r if and only if r's head relation appear in dr's body. For example, rules in line 19 and 26 of Listing 5.1 are directly dependent on the `vote` transaction rule in line 15. For the next trigger τ', literal insertion results from a new relational tuple being derived from one of the rules. For example, if the rule for transaction `vote` is evaluated to true, the next trigger τ' is `insert vote(v,p)`. Literal deletion happens when literals with primary keys are inserted: inserting such literals implicitly deletes the literals with the same primary keys, if exist. Next, for each direct dependent rule dr of r and trigger τ', it gets dr's encoding by recursively calling itself on dr and τ'.

Step 3 generates state variables for the head relation, where H is for the current step, and H' is for the next transition step. Step 4 generates the head relation update constraint: H' equals inserting or deleting r's evaluation result from H.

$GetUpdate(H, r, \tau)$ is defined as follows:

$$\text{GetUpdate}(H, r, \tau) = \begin{cases} H.insert(r.head), & \text{if r is agg. rule} \vee \tau = \text{insert \_} \\ H.delete(r.head), & \text{if r is join rule} \wedge \tau = \text{delete \_} \end{cases}$$

If r is an aggregation rule, the update is directly encoded as insertion since new aggregation results implicitly overwrites the old ones. If r is a join rule, and the trigger τ is a tuple deletion, then r's join result with the deleted tuple needs to be deleted as well. Otherwise, τ is an insertion, and the update for relation $r.head$ is also an insertion. Suppose we are in the recursion step for encoding the `votes` rule in line 19, its update constraint is generated as: $votes' = Store(votes, p, votes[p]+1)$, where the votes for proposal p is incremented by one.

Step 5 generates the constraint where r's body is true, in conjunction with the update constraint and all dependent rules' constraints. Step 6, on the other hand, generates constraints where r's body is false, no dependent rule is triggered, and the head relation remains the same. Step 7 returns the final formula as an exclusive-or of the true and false branches, which encodes r's body and how its update affects other relations in the contract.

5.3.3 Encoding Rule Bodies

The procedure EncodeRuleBody is defined by two sets of inference rules. The first judgment of the form $\Gamma, \tau \vdash r \hookrightarrow \phi$ states that a DeCon rule r is encoded by a boolean formula ϕ under context Γ and τ. The second judgment of the form $\Gamma, \tau \vdash Pred \rightsquigarrow \phi$ states that a predicate $Pred$ is encoded by a formula ϕ under context Γ and τ. The contexts (Γ and τ) of both judgement forms are defined the same as the input of Algorithm 7.

The judgment $\Gamma, \tau \vdash r \hookrightarrow \phi$ is defined by the following inference rules:

$$\frac{\Gamma, \tau \vdash R(\bar{x}) \rightsquigarrow \phi}{\Gamma, \tau \vdash H(\bar{y}) \text{ :- } R(\bar{x}) \hookrightarrow \phi} \; (Join1)$$

$$\frac{\Gamma, \tau \vdash Pred \rightsquigarrow \phi_1 \quad \Gamma, \tau \vdash H(\bar{y}) :\text{-} Join \hookrightarrow \phi_2}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} Pred, Join \hookrightarrow \phi_1 \wedge \phi_2} \ (Join2)$$

A $Join$ rule is encoded as a conjunction of the predicates, each of which is encoded from a literal in the rule body. The encoding of individual literals is introduced later in this section.

$$\frac{\tau = insert\_ \quad s' = \Gamma(H)[\bar{k}].value + n}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), s = sum\ n : R(\bar{z}) \hookrightarrow s = s'} \ (Sum\text{-}insert)$$

$$\frac{\tau = delete\_ \quad s' = \Gamma(H)[\bar{k}].value - n}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), s = sum\ n : R(\bar{z}) \hookrightarrow s = s'} \ (Sum\text{-}delete)$$

$$\frac{\tau = insert\_ \quad \phi \coloneqq c = \Gamma(H)[\bar{k}].value + 1}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), c = count : R(\bar{z}) \hookrightarrow \phi} \ (Count\text{-}insert)$$

$$\frac{\tau = delete\_ \quad \phi \coloneqq c = \Gamma(H)[\bar{k}].value - 1}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), c = count : R(\bar{z}) \hookrightarrow \phi} \ (Count\text{-}delete)$$

Unlike $Join$ rules, aggregation rules (Sum and $Count$) have separate inference rules for tuple insertion and deletion. Because the relation between new and old aggregation results needs to be encoded. In these reference rules, $\bar{k}$ represents the primary keys of relation H, extracted from the array $\bar{y}$, and $\Gamma(H)[\bar{k}].value$ reads the current aggregate result. Note that, unlike the $Join$ rules, the literal $R(\bar{x})$ here does not join with the aggregation literal, because it is only introduced to obtain valid valuations for the rule variables (every row in table R is a valid valuation). For each valid valuation, the aggregator computes the aggregate summary for the matching rows in table R (Section 4.3).

$$\frac{\tau = insert\_ \quad m' = \Gamma(H)[\bar{k}].value \quad \phi \coloneqq (n > m' \wedge m = n) \oplus (n \leq m')}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), m = max\ n : R(\bar{z}) \hookrightarrow \phi} \ (Max)$$

$$\frac{\tau = insert\_ \quad m' = \Gamma(H)[\bar{k}].value \quad \phi \coloneqq (n < m' \wedge m = n) \oplus (n \geq m')}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), m = min\ n : R(\bar{z}) \hookrightarrow \phi} \ (Min)$$

For Max and Min aggregation rules, DCV only encodes the their update for tuple insertions, based on the assumption that they only apply to transaction relations (tables that stores the transaction records), which are append only and has no primary keys. In other words, they have no tuple deletion.

This assumption is made for two reasons. First, updating Max and Min for tuple deletion is complicated, because if the current maximum or minimum is deleted, the next biggest or smallest element needs to be fetched and become the new aggregation result. Such update requires storing the whole table and even maintaining sorted table entries. Second, Ethereum has strict limits on the computation and storage of each smart contract and its transactions. Maintaining maximum and minimum for tables with delete operation is too expensive to execute on Ethereum. We survey smart contracts in public repositories and find no contract with such logic. Therefore, DCV adds such assumption and greatly simplify the rule encoding.

Encoding individual literals. Following are the inference rules for judgment: $\Gamma, \tau \vdash Pred \rightsquigarrow \phi$, which encodes individual literals.

$$\frac{\tau.rel = R}{\Gamma, \tau \vdash R(\bar{x}) \rightsquigarrow \tau = R(\bar{x})}\ (Lit1) \quad \frac{\tau.rel \neq R}{\Gamma, \tau \vdash R(\bar{x}) \rightsquigarrow \Gamma(R)[\bar{k}] = \bar{v}}\ (Lit2)$$

where $\bar{k}$ represents the primary keys in relational literal $R(\bar{x})$, extracted from $\bar{x}$, and $\bar{v}$ represents the remaining fields in $\bar{x}$.

$$\frac{}{\Gamma, \tau \vdash C \rightsquigarrow C}\ (Condition) \quad \frac{}{\Gamma, \tau \vdash y = F(\bar{x}) \rightsquigarrow y = F(\bar{x})}\ (Function)$$

Recursions. DCV assumes that on every new incoming transaction request, there is at most one new tuple derived by each rule, and that there is no recursion in the rules.

Recursion means that there is a mutual dependency between rules. A rule r_a is dependent to another rule r_b ($r_a \rightarrow r_b$) if and only if r_b's head relation appears in r_a's body, or there exists another rule r_c such that $r_a \rightarrow r_c \wedge r_c \rightarrow r_b$.

This assumption keeps the size of the transition constraint linear to the number of rules in the Decon contract, thus making the safety verification tractable. We find this assumption holds for most smart contracts in the financial domain, and is true for all of the ten benchmark contracts in our evaluation.

Multi-contract Interactions. Multi-contract interaction is specified implicitly by DeCon rules that join relations from different contracts. Such interactions are per-

formed via message passing. Unlike prior work that check for message handling errors, DCV assumes the message delivery and handling is always successful, and instead focuses on the functional correctness. Note that such interactions are limited to functions without mutual recursions. Mutual recursions are not supported because it breaks the atomicity assumption of a transaction. To illustrate, suppose contract A's transaction calls contract B's transaction *Foo*, which in turn calls another transaction of contract A. In this case, two transactions of contract A are executed in parallel, breaking the atomicity of transactions.

5.3.4 Safety Invariant Generation

Each violation query rule qr in a DeCon contract is first encoded as a formula ϕ such that $\Gamma, \tau \vdash qr \hookrightarrow \phi$. Note that the context Γ is the same mapping used in the transition system encoding process. The second context, trigger τ, is a reserved literal $check()$, which triggers the violation query rule after every transaction.

Next, the safety invariant is generated from ϕ as follows:

$$Prop \triangleq \neg(\exists x \in X.\ \phi(s, x))$$

where X is the state space for the set of non-state variables in ϕ. The property states that there exists no valuations of variables in X such that the violation query is non-empty. In other words, the system is safe from such violation.

5.4 Verification Method

5.4.1 Proof by Induction

Given a state transition system $\langle S, I, E, Tr \rangle$ transformed from the Decon contract, DCV uses mathematical induction to prove the target property $prop$. The induction procedure is defined in Equation 5.1, where $init$ and tr are the Boolean formulas

that define the set of initial states I and the transition relation Tr. The rest of this section introduces the algorithm to infer the inductive invariant inv used by the induction proof.

Algorithm 8 Procedure to find inductive invariants.

Input: a transition system ts, a map from relation to its modeling variable Γ, and a set of DeCon transaction rules R.

Output: an inductive invariant of ts.

```
1: function FindInductiveInvariant(C,ts)
2:     for inv in C do:
3:         if refuteInvariant(inv, C, ts) then
4:             return FindInductiveInvariant(C \ inv, ts)
5:         end if
6:     end for
7:     return ⋀_{c_i ∈ C} c_i
8: end function
9: P ← ⋃_{r ∈ R} ExtractPredicates(r, Γ)
10: C ← GenerateCandidateInvariants(P)
11: return FindInductiveInvariant(C, ts)
```

Algorithm 8 presents the procedure to infer inductive invariants. It first extracts a set of predicates P from the set of transaction rules R (Section 5.4.2). Then it generates a set of candidate invariants using predicates in P, following two heuristic patterns (Section 5.4.3). Finally, it invokes a recursive subroutine FindInductiveInvariant to find an inductive invariant.

The procedure FindInductiveInvariants is adopted from the Houdini algorithm [77]. It iteratively refutes candidate invariants in C, until there is no candidate that can be refuted, and returns the conjunction of all remaining invariants. The subroutine refuteInvariant is defined in Equation 5.4, which refutes a candidate invariant if it is not inductive.

$$\text{refuteInvariant}(inv, C, ts) \triangleq \begin{array}{l} \vee\neg(ts.init \implies inv) \\ \vee\neg[(\bigwedge_{c\in C} c) \wedge ts.tr \implies inv'] \end{array} \tag{5.4}$$

where inv' is adopted by replacing all state variables in inv with their corresponding variable in the next transition step.

A property of this algorithm is that, given a set of candidate invariants C, it always returns the strongest inductive invariant that can be constructed in the form of conjunction of the candidates in C [77].

5.4.2 Predicate Extraction

Algorithm 9 ExtractPredicate(r, Γ).

Input: a transaction rule r, a map from relation to its modeling variable Γ.

Output: a set of predicates P.

1: $\tau \leftarrow r.trigger$

2: $P_0 \leftarrow \{p \mid l \in r.body, \Gamma, \tau \vdash l \rightsquigarrow p\}$

3: $P_1 \leftarrow \{p \wedge q \mid p \in P_0, q \in \text{MatchingPredicates}(p, r)\}$

4: **return** $P_0 \cup P_1$

Algorithm 9 presents the predicate extraction procedure. It first transforms each literal in the transaction rule into a predicate, and puts them into a set P_0.

Some predicates in P_0 do not contain enough information on their own, e.g., predicates that contain only free variables. Because the logic of a rule is established on the relation among its literals (e.g. two literals sharing the same variable v means joining on the corresponding columns). On the contrary, predicates that contain constants, e.g. hasWinner == true, convey the matching of a column to a certain concrete value, and can thus be used directly in candidate invariant construction.

Therefore, in the next step, each predicate p in P_0 is augmented by one of its matching predicates in $matchingPredicates(p, r)$, which is the set of predicates in

rule r that share at least one variable with predicate p. This set of augmented predicates is P_1. Finally, the union of P_0 and P_1 is returned.

5.4.3 Candidate Invariant Generation

Given the set of predicates in P, DCV generates candidate invariants in the following patterns:

$$\{\forall x \in X.\ \neg init(s) \implies \neg p(s, x) \mid p \in P\}$$
$$\{\forall x \in X.\ \neg init(s) \wedge q(s, x) \implies \neg p(s, x) \mid p, q \in P\}$$

where X is the set of non-state variables in the body of the formula. $\neg init(s)$ is used as the implication premise so that the whole formula can be trivially implied by the transition system's initial constraints. Having $\neg p$ as the implication conclusion is based on the observation that, in order to prove safety invariants, a lemma is needed to prevent the system from unsafe transitions. In the second pattern, we add another predicate $q \in P_0$ in the implication premise to make the pattern more robust.

5.5 Evaluation

Benchmarks. We survey public smart contract repositories [13, 18, 16], and gather 10 representative contracts as the evaluation benchmarks. Each selected contract either has contract-level safety specifications annotated, or has proper documentation from which we can come up with a contract-level safety specification. They cover a wide range of application domains, including ERC20 [57] and ERC721 [55], the two most popular token standards. Table 5.1 shows all contract names and their target properties.

Baselines. We use solc [2] and solc-verify [68] as the comparison baselines. Solc is a Solidity compiler with a built-in checker to verify assertions in source programs.

Table 5.1: Benchmark properties.

Benchmarks	Properties
wallet	No negative balance.
crowFunding	No missing fund.
ERC20	Account balances add up to totalSupply.
ERC721	All existing token has an owner.
ERC777	No default operators is approved for individual account.
ERC1155	Each token's account balances add up to that token's totalSupply.
paymentSplitter	No overpayment.
vestingWallet	No early release.
voting	At most one winning proposal.
auction	Each participant can withdraw at most once.

It has been actively maintained by the Ethereum community, and version 0.8.13 is used for this experiment. Solc-verify extends from solc 0.7.6 and performs automated formal verification using strategies of specification annotation and modular program verification. We have also considered Verx [96] and Zeus [72], but neither is publicly available.

Experiment setup. We modify certain functionalities and syntax of the benchmark contracts so that they are compatible with all comparison tools. In particular, the delegate vote function of the voting contract contains recursion, which is not yet supported by DeCon, and is thus dropped. In addition, solc and solc-verify do not support inline assembly analysis. Therefore, inline assembly in the Solidity contracts are replaced with native Solidity code. Minor syntax changes are also made to satisfy version requirements of the two baseline tools.

With these modifications, for each reference contract in Solidity, we implement its counterpart in DeCon. Then we conduct verification tasks on three versions of benchmark contracts: (1) DeCon contracts with DCV, (2) reference Solidity con-

tracts with solc and solc-verify, and (3) Solidity contracts generated from DeCon with solc and solc-verify. For each set of verification tasks, we measure the verification time and set the time budget to be one hour. All experiments are performed on a server with 32 2.6GHz cores and 125GB memory.

Table 5.2: Verification efficiency measured in time (seconds). TO stands for time-out after 1 hour. Unknown means the verifier cannot verify the contract property. Errors from solc are caused by a known software issue [15]. Solc-verify fails to analyze part of the OpenZeppelin libraries, and thus returns error.

Benchmarks	#Rules	LOC	DCV	Solc		Solc-verify	
				reference	DeCon	reference	DeCon
wallet	12	67	1	1	unknown	4	unknown
crowFunding	14	85	1	1	error	unknown	7
ERC20	19	389	1	26	error	error	8
ERC721	13	520	1	TO	TO	error	8
ERC777	31	562	1	TO	unknown	10	unknown
ERC1155	18	645	3	12	TO	6	8
paymentSplitter	6	166	1	TO	TO	4	unknown
vestingWallet	7	113	1	TO	unknown	9	4
voting	6	36	1	error	2	unknown	unknown
auction	13	146	54	error	TO	unknown	unknown

Results. Table 5.2 shows the evaluation results. DCV verifies all but two contracts in one second, with ERC1155 in three seconds and auction in 54 seconds. In particular, the properties for the voting and auction contract are not inductive, and thus require inductive invariant generation. Auction takes more time because it contains more rules and has a more complicated inductive invariant.

On the other hand, solc only successfully verifies four reference contracts, with comparable efficiency. It times out on four contracts, and reports SMT solver in-

vocation error on another two. This error has been an open issue according to the GitHub repository issue tracker [15], which is sensitive to the operating system and the underlying library versions of Z3.

Similarly, solc-verify verifies five reference contracts, and reports unknown on three others. It also returns errors on two contracts because it cannot analyze certain parts of the included OpenZepplin libraries, although the libraries are written in compatible Solidity version.

For Solidity contracts generated from DeCon, solc verifies one and solc-verify verifies five. The performance difference between the reference version and the DeCon-generated version is potentially caused by the fact that DeCon generates stand-alone contracts that implement all functionalities without external libraries. On the other hand, DeCon implements contract states (relations) as mappings from primary keys to tuples, which may incur extra analysis complexity compared to the reference version.

In summary, DCV is highly efficient in verifying contract-level safety invariants, and can handle a wider range of smart contracts compared to other tools. By taking advantage of the high-level abstractions of the DeCon language, it achieves significant speedup over the baseline verification tools. In several instances, alternative tools timeout after an hour or report an error, while DeCon is able to complete verification successfully.

5.6 Related work

Verification of Solidity smart contracts. Solc [84], Solc-verify [68], Zeus [72], Verisol [119], and Verx [96] perform safety verification for smart contracts. Similar to DCV, they infer inductive invariants to perform sound verification of safety properties. They also generate counter-examples as a sequence of transactions to disprove the safety properties. SmartACE [120] is a safety verification framework that

incorporates a wide variety of verification techniques, including fuzzing, bounded model checking, symbolic execution, etc. In addition to safety properties, SmartPulse [111] supports liveness verification. It leverages the counterexample-guided abstraction refinement (CEGAR) paradigm to perform efficient model checking, and can generate attacks given an environment model.

DCV differs from these work in that it uses a high-level executable specification, DeCon, as the verification target. Such high-level modeling improves verification efficiency, but it also means that DCV can only apply to smart contracts written in DeCon, which is a new language, while the other tools can work on most existing smart contracts in Solidity.

The Move Prover [53] (MVP) is a formal verifier for smart contracts written in the Move language [12]. Similar to DCV, MVP also verifies safety properties of smart contracts. However, they target different languages and blockchain platforms. DCV is based on DeCon, which is declarative and more abstract, while Move is imperative. In addition, Move contracts work on the Diem blockchain, while DeCon currently supports Ethereum and Solidity. Despite the differences, we believe DCV could also benefit Move. An interesting future direction would be to implement a Move compiler for DeCon, so that DeCon can serve as a declarative specification for smart contracts on the Diem blockchain, while Move as the implementation language can provide better support for other verification tasks.

Formal semantics of smart contracts. KEVM [48] introduces formal semantics for smart contracts, and can automatically verify that a Solidity program (its compiled EVM bytecode) implements the formal semantics specified in KEVM. This verification is also sound, but it focuses on the functional correctness of each Solidity function, instead of the state invariants across multiple transactions.

Formal semantics of EVM bytecode have also been formalized in F* [64] and Isabelle/HOL [25]. Scilla [102] is a type-safe intermediate language for smart contracts that also provides formal semantics. They offer precise models of the smart

contract behaviors, and support deductive verification via proof assistants. However, working with a proof assistant requires non-trivial manual effort. On the contrary, DCV provides fully automatic verification.

Vulnerability detection. Securify [115] encodes smart contract semantic information into relational facts, and uses Datalog solver to search for property compliance and violation patterns in these facts. Oyente [83] uses symbolic execution to check generic security vulnerabilities, including reentrancy attack, transaction order dependency, etc. Maian [11] detects vulnerabilities by analyzing transaction traces. Unlike the sound verification tools, which require some amount of formal specification from the users, these work require no formal specification and can be directly applied to any existing smart contracts without modification, offering a quick and light-weight alternative to sound verification, although may suffer from false positives or negatives.

Fuzzing and testing. Fuzzing and testing techniques have also been widely applied to smart contract verification. They complement deductive verification tools by presenting concrete counter-examples. ContractFuzzer [71] instruments EVM bytecodes to log run-time contract behaviors, and uncovers security vulnerabilities from these run-time logs. Smartisan [49] uses static analysis to predict effective transaction sequences, and uses this information to guide fuzzing process. SmartTest [109] introduces a language model for vulnerable transaction sequences, and uses this model to guide the search path in the fuzzing phase.

5.7 Conclusion

We present DCV, an automatic safety verification tool for declarative smart contracts written in the DeCon language. It leverages the high-level abstraction of DeCon to generate succinct models of the smart contracts, performs sound verification via mathematical induction, and applies domain-specific adaptations of the

Houdini algorithm to infer inductive invariants. Evaluation shows that it is highly efficient, verifying all 10 benchmark smart contracts, with significant speedup over the baseline tools.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Declarative specification is a powerful tool for system design and verification. The abstraction makes it easy to analyze and verify high-level design properties. At the same time, program synthesis technique automates the implementation process, improving system implementation efficiency.

This dissertation addresses two challenges in the adoption declarative system specification. NetSpec focuses on improving the system specification efficiency. It automates the network specification process via input-output example based program synthesis technique. To make the tool robust to incomplete input-output examples, it further introduces active learning that queries the user for additional examples.

DSC introduces declarative program in an emerging domain: Ethereum smart contracts. It makes the specification and verification of smart contracts easier via a high-level declarative language. And it makes the implementation more efficient by generating executable programs from the specification automatically.

There are many open and exciting questions remain to be answered in the future. First is the interaction between program synthesis and program verification. Given a correctness specification of a network protocol, and input-output examples that describe the execution logic of the protocol design, can we synthesize a protocol that satisfies both the input-output examples and the correctness properties?

Second, could we utilize the declarative specification of smart contract to generate gas efficient implementation? Executing smart contracts on the blockchain is expensive, and gas cost has been one of the most important optimization metric for smart contracts. Database query execution plan generation is one of the most suc-

cessful application of performance optimization for declarative specification (SQL). It is an interesting direction to explore opportunities to optimize smart contract gas consumption based on their declarative specification.

APPENDIX A

PROOF

A.1 Proof sketch for completeness

We first present the following lemmas, and their proof sketches. With these lemmas, we then prove the completeness property, defined in Theorem 3.4.6.

Lemma A.1.1. *Given a set of input tuples and a set of output tuples (I, O), if all rules in a program p are output contributing rules (Definition 3.4.4), then* $\mathrm{Score}(p) > 0$.

Let $\mathrm{OutCtrb}(p)$ *denotes that all rules in program p are output-contributing rules, the lemma is defined as:*

$$\mathrm{OutCtrb}(p) \implies \mathrm{Score}(p) > 0 \tag{A.1}$$

Proof sketch. By the semantics of Datalog, a program's output is the union of all rules' output. Thus $p(I) \cap O \neq \emptyset$. By the definition of Score, we have that $\mathrm{Score}(p) > 0$.

Lemma A.1.2. *Given a set of input tuples and a set of output tuples (I, O), if all rules in a program p are output-contributing rules, then for every p's predecessors, all rules are also output-contributing rules:*

$$\forall q \to p, \mathrm{OutCtrb}(p) \implies \mathrm{OutCtrb}(q) \tag{A.2}$$

Proof. We enumerate all ways to generate offspring $(q \to p)$:

1. If $p \in \mathrm{AddRule}(q)$ (equation 3.4), and let r_0 be the minimal rule added in p, we have $p = q \cup r_0$. Given $\mathrm{OutCtrb}(p)$, that is, all rules in p produces some desired output in O, and $p = q \cup r_0$, we have $\mathrm{OutCtrb}(q)$.

2. If $p \in \mathrm{ExtRule}(q)$ (equation 3.7), let r be the rule in q that have been extended as r' in p. Let $r(I)$ and $r'(I)$ denote the direct derivation output of r and r' on input I, respectively.

   (a) Given $\mathrm{OutCtrb}(p)$, and that $r' \in p$, we have that $r'(I) \cap O \neq \emptyset$.

   (b) By the semantics of Datalog, adding a predicate to a rule monotonically reduces the output of the rule. Thus we have $r'(I) \subseteq r(I)$.

   (c) Given that $r'(I) \cap O \neq \emptyset$, and that $r'(I) \subseteq r(I)$, we have that $r(I) \cap O \neq \emptyset$.

   (d) Given that all other rules in q are also in p, we have that $\mathrm{OutCtrb}(q)$.

3. If $p \in \mathrm{MkAgg(q)}$, given that NetSpec only introduces $argMax$ and $argMin$ aggregations, $p(I) \subseteq q(I)$.

   (a) For rules $r \in q$ whose output are aggregated and renamed as $r' \in p$, because NetSpec introduces only $argMax$ and $argMin$ aggregations, $r'(I) \subseteq r(I)$.

   (b) Following the same reasoning from 2c) to 2d), we have that $\mathrm{OutCtrb}(q)$.

□

Lemma A.1.3. *Given a set of input tuples and a set of output tuples (I, O), if all rules in a program p are output-contributing rules, then there exists a lineage of programs $p_0 \to p_1 \to p_2 \to ... \to p_n \to p$, such that p_0 is the empty programs, and $p_1, ..., p_n$ contain only output-contributing rules:*

$$\begin{aligned} \mathrm{Lineage}(p) \coloneqq \mathrm{OutCtrb}(p) \implies \\ \exists p_1, ..., p_n, \ [(p_0 \to p_1 \to ... \to p_n \to p) \\ \wedge \forall i \in \{1, ..., n\}, \mathrm{OutCtrb}(p_i)] \\ \forall p, \mathrm{Lineage}(p) \end{aligned} \tag{A.3}$$

Proof sketch. We prove by well-founded induction on the successor relation $\rightarrow$ on the program space (Definition 3.4.3). $\rightarrow$ is a well-founded relation on program space, because rules or literals cannot be taken away from a program indefinitely. To prove $\forall p, \text{Lineage}(p)$ it is suffice to show that:

$$\forall p, [\forall q \rightarrow p, \text{Lineage}(q)] \implies \text{Lineage}(p) \tag{A.4}$$

If OutCtrb(p) is true, then by Lemma A.1.2, we have that $\forall q \rightarrow p, \text{OutCtrb}(q)$. By the antecedent of the induction hypothesis (equation A.4), we have that there exists a lineage of programs $p_0 \rightarrow p_1 \rightarrow ... \rightarrow q$, and $\forall i \in 1, ..., n, \text{OutCtrb}(p_i) \wedge \text{Score}(p_i)$. Let $p_{n+1} = q$, and given that $q \rightarrow p$, we have that $\text{Lineage}(p)$.

By well-founded induction, we have that $\forall p, \text{Lineage}(p)$.

Lemma A.1.4 (Termination)**.** *For all finite set of input tuples and output tuples (I,O), NetSpec always terminates.*

Proof sketch. We first show that the search space of NetSpec is finite. First, suppose there are N_R input relations, then according to the syntax constraints in Chapter 2, each rule contains at most $2N_R$ literals. Second, in offspring generation, we only add a rule to a candidate program if it has imperfect recall. In the worst case, each rule generates a tuple in O. Therefore, a program contains at most $|O|$ rules.

Let E_n be the set of all programs that have been popped from Q_n at the beginning of iteration n. Let P be the search space of NetSpec. We construct a function on iteration number n: $f(n) = |P| - |E_n|$. We then show that $f(n) >= 0$ and $f(n+1) < f(n)$. By the principle of well-founded induction, NetSpec terminates.

Proof sketch for weak completeness (Theorem 3.4.6). We first prove the case where valid solution exists. We prove by induction on iterations in Algorithm 1. We use subscript n to denote the state variable values at the beginning of the n^{th} iteration, e.g., Q_n is the set of candidate programs at the beginning of iteration n.

Given a solution p, and by Lemma A.1.3, we have that there exists a lineage of programs: $p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_k \rightarrow p$, such that p_0 is the empty programs, and

$p_1, ..., p_k$ contain only output-contributing rules.

Induction hypothesis: In every iteration, either p is in the solution set S_n, or one of p's ancestors in the lineage from p_0 to p is in the set of candidate programs Q_n:

$$\forall n, p \in S_n \vee (\exists i \in \{0, 1, 2, ..., k\}, p_i \in Q_n) \tag{A.5}$$

Base case: In iteration 0, by algorithm 1 step 1, Q is initialized with only the empty program p_0. Thus induction hypothesis holds.

Induction: Suppose in the n^{th} iteration, the induction hypothesis holds, which implies either of the following:

1. If $p \in S_n$, by algorithm 1 step 2b, we have $S_n \subseteq S_{n+1}$. This implies that $p \in S_{n+1}$. Thus induction hypothesis holds in iteration $n + 1$.

2. Otherwise, $\exists i \in \{0, 1, 2, ..., k\}, p_i \in Q_n$. We discuss by two cases on the value of the current program P_n:

   (a) If $P_n \neq p_i$, by step 2b, every program in Q_n is copied into Q_{n+1} except P, thus p_i remains in Q_{n+1}. Induction hypothesis holds in iteration n+1.

   (b) Otherwise, $P_n = p_i$.

      i. In step 2a, all offspring of p_i are generated. By the definition of the successor relation $\rightarrow$ (Definition 3.4.3), $p_{i+1} \in \text{Offspring}(p_i)$.

      ii. By Lemma A.1.1 and Lemma A.1.3, $\text{Score}(p_{i+1}) > 0$.

      iii. In step 2b, all offspring with score greater than 0 is added to Q_{n+1}. Given $\text{Score}(p_{i+1}) > 0$, we have that $p_{i+1} \in Q_{n+1}$. Induction hypothesis holds in iteration n+1.

This induction hypothesis, in conjunction with the termination condition that $Q = \emptyset$, implies that $p \in S$ when NetSpec terminates.

For the second case, when no valid solution exists, by Theorem 3.4.1 (soundness) and Lemma A.1.4 (termination), we have that NetSpec will terminate with $S = \emptyset$.

BIBLIOGRAPHY

[1] Openzeppelin contracts. https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/token. [Cited on page 78]

[2] Solidity. https://docs.soliditylang.org, . [Cited on pages 50, 51, and 102]

[3] Smtchecker and formal verification. https://docs.soliditylang.org/en/v0.8.12/smtchecker.html#assert, . [Cited on page 49]

[4] Simple auction. https://docs.soliditylang.org/en/v0.5.3/solidity-by-example.html#simple-open-auction, . [Cited on page 78]

[5] Solidity events. https://docs.soliditylang.org/en/v0.8.13/abi-spec.html?highlight=events#events, . [Cited on page 69]

[6] Safe remote purchase. https://docs.soliditylang.org/en/v0.5.3/solidity-by-example.html#safe-remote-purchase, . [Cited on page 79]

[7] Voting. https://docs.soliditylang.org/en/v0.5.3/solidity-by-example.html#voting, . [Cited on page 79]

[8] Truffle. https://trufflesuite.com. [Cited on page 78]

[9] The dao. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413, 2016. [Cited on page 49]

[10] King of the ether throne — post-mortem investigation. https://www.kingoftheether.com/postmortem.html, 2016. [Cited on page 49]

[11] Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 653–663, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365697. doi: 10.1145/3274694.3274743. URL https://doi.org/10.1145/3274694.3274743. [Cited on page 107]

[12] The move team: The move programming language. https://diem.github.io/move/, 2020. [Cited on page 106]

[13] Openzeppelin. https://github.com/OpenZeppelin/openzeppelin-contracts, 2022. [Cited on page 102]

[14] Netspec synthesis result validation. https://github.com/HaoxianChen/netspec/blob/master/synthesis-validation, 2022. [Cited on page 34]

[15] Cannot replicate smtchecker example output. https://github.com/ethereum/solidity/issues/13073, 2022. [Cited on pages x, 104, and 105]

[16] Solidity by example. https://docs.soliditylang.org/en/v0.8.17/solidity-by-example.html, 2022. [Cited on pages 88 and 102]

[17] Smtchecker and formal verification. https://docs.soliditylang.org/en/v0.8.17/smtchecker.html, 2022. [Cited on page 85]

[18] Verx smart contract verification benchmarks. https://github.com/eth-sri/verx-benchmarks, 2022. [Cited on page 102]

[19] Z3. https://github.com/Z3Prover/z3, 2022. [Cited on page 87]

[20] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Pearson, 1st edition, 1994. [Cited on pages 6, 7, 12, 17, and 18]

[21] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In *Int. symposium on automated technology for verification and analysis (ATVA)*, pages 513–520. Springer, 2018. [Cited on pages 49 and 82]

[22] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Safevm: a safety verifier for ethereum smart contracts. In *ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 386–389, 2019. [Cited on page 82]

[23] Leonardo Alt and Christian Reitwiessner. Smt-based verification of solidity smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*, pages 376–388. Springer, 2018. [Cited on page 82]

[24] Peter Alvaro, Tyson Condie, Neil Conway, Joseph M Hellerstein, and Russell Sears. I do declare: Consensus in a logic language. *ACM SIGOPS Operating Systems Review*, 2010. [Cited on pages 1, 7, and 34]

[25] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77, 2018. [Cited on page 106]

[26] Pedro Antonino and AW Roscoe. Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity. *arXiv preprint arXiv:2002.02710*, 2020. [Cited on page 85]

[27] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *Int. Conf. on Open and Big Data (OBD)*, pages 25–30. IEEE, 2016. [Cited on page 49]

[28] Shaun Azzopardi, Gordon J Pace, and Fernando Schapachnik. On observing contracts: deontic contracts meet smart contracts. In *Legal Knowledge and Information Systems*, pages 21–30. IOS Press, 2018. [Cited on page 82]

[29] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, et al. Reachability analysis for aws-based networks. In *International Conference on Computer Aided Verification*, 2019. [Cited on pages 1, 6, 7, and 34]

[30] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, 2014. [Cited on pages 1, 6, 7, 34, and 48]

[31] Massimo Bartoletti and Roberto Zunino. Bitml: a calculus for bitcoin smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 83–100, 2018. [Cited on page 83]

[32] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016. [Cited on pages 1 and 47]

[33] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017. [Cited on pages 1 and 48]

[34] Aaron Bembenek, Michael Greenberg, and Stephen Chong. Formulog: Datalog for smt-based static analysis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020. [Cited on page 50]

[35] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Config2spec: Mining network specifications from network configurations. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020. [Cited on page 47]

[36] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: Secure derivative contracts for ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 453–467. Springer, 2017. [Cited on page 49]

[37] Nikolaj Bjørner and Karthick Jayaraman. Checking cloud contracts in microsoft azure. In *International Conference on Distributed Computing and Internet Technology*, 2015. [Cited on page 1]

[38] Ryan Browne. 'accidental' bug may have frozen $ 280 million worth of digital coin ether in a cryptocurrency wallet. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html, 2017. [Cited on page 49]

[39] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *International conference on database theory*, pages 316–330. Springer, 2001. [Cited on page 51]

[40] Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *NSDI*, 2021. [Cited on page 47]

[41] Franck Cassez, Joanne Fuller, and Horacio Mijail Antón Quiles. Deductive verification of smart contracts with dafny. In *International Conference on Formal Methods for Industrial Critical Systems*, pages 50–66. Springer, 2022. [Cited on page 85]

[42] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. scompile: Critical path identification and analysis for smart contracts. In *Int. Conf. on Formal Engineering Methods*, pages 286–304. Springer, 2019. [Cited on page 82]

[43] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016. [Cited on pages 1 and 7]

[44] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. A program logic for verifying secure routing protocols. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, 2014. [Cited on page 1]

[45] Haoxian Chen, Anduo Wang, and Boon Thau Loo. Towards example-guided network synthesis. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, 2018. [Cited on pages 36, 37, and 46]

[46] Haoxian Chen, Gerald Whitters, Mohammad Javad Amiri, Yuepeng Wang, and Boon Thau Loo. Declarative smart contracts. In *ESEC/FSE '22*, 2022. [Cited on page 89]

[47] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. Soda: A generic online detection framework for smart contracts. In *NDSS*, 2020. [Cited on page 82]

[48] Xiaohong Chen, Daejun Park, and Grigore Roşu. A language-independent approach to smart contract verification. In *International Symposium on Leveraging Applications of Formal Methods*, pages 405–413. Springer, 2018. [Cited on pages 85 and 106]

[49] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 227–239. IEEE, 2021. [Cited on page 107]

[50] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, 2007. [Cited on page 34]

[51] Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016. URL https://github.com/metagol/metagol. [Cited on page 47]

[52] Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. Turning 30: New ideas in inductive logic programming. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2020. [Cited on pages 12 and 47]

[53] David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 183–200. Springer, 2022. [Cited on page 106]

[54] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018. [Cited on pages 1, 7, 34, and 47]

[55] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. Eip-721: Non-fungible token standard. https://eips.ethereum.org/EIPS/eip-721. [Cited on pages 75 and 102]

[56] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61, 2018. [Cited on page 47]

[57] Vitalik Buterin Fabian Vogelsteller. Eip-20: Token standard. https://eips.ethereum.org/EIPS/eip-20. [Cited on pages 72 and 102]

[58] Floodlight. 2020. http://www.projectfloodlight.org/floodlight/. [Cited on pages 9 and 45]

[59] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th {USENIX} symposium on networked systems design and implementation ({NSDI} 15)*, 2015. [Cited on pages 1, 6, 7, 12, 34, and 48]

[60] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM Sigplan Notices*, (9), 2011. [Cited on page 48]

[61] Joel Frank, Cornelius Aschermann, and Thorsten Holz. {ETHBMC}: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774, 2020. [Cited on page 85]

[62] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, pages 557–560, 2020. [Cited on page 83]

[63] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In *Int. Conf. on Computer Aided Verification*, pages 51–78. Springer, 2018. [Cited on page 49]

[64] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer, 2018. [Cited on page 106]

[65] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–28, 2017. [Cited on page 82]

[66] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2): 157–166, 1993. [Cited on page 51]

[67] Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. Smart contract-based campus demonstration of decentralized transactive energy auctions. In *2017 IEEE Power & energy society innovative smart grid technologies conference (ISGT)*, pages 1–5. IEEE, 2017. [Cited on page 49]

[68] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *Working conference on verified software: theories, tools, and experiments*, pages 161–179. Springer, 2019. [Cited on pages 85, 102, and 105]

[69] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to

smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conf. on Computer and Communications Security*, pages 531–548, 2019. [Cited on page 83]

[70] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018. doi: 10.1109/CSF.2018.00022. [Cited on page 83]

[71] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018. [Cited on pages 82 and 107]

[72] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018. [Cited on pages 82, 85, 103, and 105]

[73] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013. [Cited on page 48]

[74] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015. [Cited on page 48]

[75] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In *ACM SIGSOFT Int.*

symposium on software testing and analysis (ISSTA), pages 363–373, 2019. [Cited on page 82]

[76] Johannes Krupp and Christian Rossow. {teEther}: Gnawing at ethereum to automatically exploit smart contracts. In *USENIX Security Symposium*, pages 1317–1333. USENIX, 2018. [Cited on page 82]

[77] Shuvendu K Lahiri and Shaz Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *International Conference on Automated Deduction*, pages 214–229. Springer, 2009. [Cited on pages 86, 92, 93, 100, and 101]

[78] Mark Law, Alessandra Russo, and Krysia Broda. The ilasp system for inductive learning of answer set programs. *arXiv preprint arXiv:2005.00904*, 2020. [Cited on pages 8 and 47]

[79] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 438–453, 2020. [Cited on page 81]

[80] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Int. Conf. on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018. [Cited on page 82]

[81] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 2009. [Cited on pages 1, 6, 7, 9, and 34]

[82] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th {USENIX}*

Symposium on Networked Systems Design and Implementation ({NSDI} 15), 2015. [Cited on pages 1, 6, 7, 12, and 34]

[83] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *ACM SIGSAC Conf. on computer and communications security (CCS)*, pages 254–269, 2016. [Cited on pages 82 and 107]

[84] Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti EJ Hyvärinen, and Natasha Sharygina. Accurate smart contract verification through direct modelling. In *International Symposium on Leveraging Applications of Formal Methods*, pages 178–194. Springer, 2020. [Cited on pages 85 and 105]

[85] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In *International conference on financial cryptography and data security*, pages 357–375. Springer, 2017. [Cited on page 49]

[86] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 1998. [Cited on page 17]

[87] Jonathan Mendelson, Aaditya Naik, Mukund Ragothaman, and Mayur Naik. Gensynth: Synthesizing datalog programs without language bias. *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2021. [Cited on pages 9, 36, and 37]

[88] Mininet. http://mininet.org/. [Cited on page 45]

[89] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts.

In *2019 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019. [Cited on page 82]

[90] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19, 1994. [Cited on page 12]

[91] Zeinab Nehai, Pierre-Yves Piriou, and Frederic Daumas. Model-checking of smart contracts. In *Int. Conf. on Blockchain*, pages 980–987. IEEE, 2018. [Cited on page 85]

[92] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014. [Cited on pages 1, 6, 7, 9, 34, and 48]

[93] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *computer security applications Conf.*, pages 653–663, 2018. [Cited on page 82]

[94] Benedikt Notheisen, Magnus Gödde, and Christof Weinhardt. Trading stocks on blocks-engineering decentralized markets. In *International Conference on Design Science Research in Information System and Technology*, pages 474–478. Springer, 2017. [Cited on page 49]

[95] Zhiniang Peng. Not a fair game – fairness analysis of dice2win. https://blogs.360.net/post/Fairness_Analysis_of_Dice2win_EN.html, 2018. [Cited on page 49]

[96] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020. [Cited on pages 49, 78, 82, 85, 103, and 105]

[97] POX. 2020. https://github.com/noxrepo/pox. [Cited on pages 9 and 45]

[98] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *Proceedings of the ACM on Programming Languages*, 2020. [Cited on pages 8 and 47]

[99] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [Cited on page 47]

[100] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *Network and Distributed Systems Security (NDSS)*, 2019. [Cited on page 82]

[101] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 621–640, 2020. [Cited on page 49]

[102] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019. [Cited on pages 83 and 106]

[103] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory (COLT'92)*, 1992. [Cited on page 31]

[104] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of Datalog programs. In *Proceedings of the Joint Meeting on European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering (ESEC/FSE), 2018. [Cited on page 47]

[105] David Siegel. Understanding the dao attack. 2016. [Cited on page 49]

[106] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *NSDI*, 2008. [Cited on pages 1 and 7]

[107] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop*, pages 245–251. Springer, 2010. [Cited on page 50]

[108] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *Symposium on Security and Privacy (SP)*, pages 1678–1694. IEEE, 2020. [Cited on page 82]

[109] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model- Guided} symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1361–1378, 2021. [Cited on page 107]

[110] Souffle. 2020. https://souffle-lang.github.io/index.html. [Cited on page 32]

[111] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. Smartpulse: automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 555–571. IEEE, 2021. [Cited on page 106]

[112] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th*

ACM SIGPLAN Symposium on Principles of Programming Languages, 2017. [Cited on page 47]

[113] Parity Technologies. Parity security alert. 2017. [Cited on page 49]

[114] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conf.*, pages 664–676, 2018. [Cited on page 82]

[115] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 67–82, 2018. [Cited on pages 49, 82, and 107]

[116] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative network verification. In *International Symposium on Practical Aspects of Declarative Languages*. Springer. [Cited on pages 1, 7, and 48]

[117] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. Fsr: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking*, 2012. [Cited on pages 1, 6, 7, 12, 34, and 48]

[118] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017. [Cited on page 9]

[119] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In *Working Conference on Verified*

Software: Theories, Tools, and Experiments, pages 87–106. Springer, 2019. [Cited on page 105]

[120] Scott Wesley, Maria Christakis, Jorge A Navas, Richard Trefler, Valentin Wüstholz, and Arie Gurfinkel. Verifying solidity smart contracts via communication abstraction in smartace. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 425–449. Springer, 2022. [Cited on page 105]

[121] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer, 2005. [Cited on page 50]

[122] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated network repair with meta provenance. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, 2015. [Cited on pages 1 and 7]

[123] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. In *ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering (ESE/SFSE)*, pages 1398–1409, 2020. [Cited on page 83]

[124] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. Scenario-based programming for sdn policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015. [Cited on page 46]

[125] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-

scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010. [Cited on pages 6 and 12]