Proof-Producing Symbolic Execution for P4

Didrik Lundberg[1,2][0000−0001−9921−3257], Roberto Guanciale[1][0000−0002−8069−6495], and Mads Dam[1][0000−0001−5432−6442]

[1] KTH Royal Institute of Technology, Lindstedtsvägen 5, 100 44 Stockholm, Sweden
`{didrikl, robertog, mfd}@kth.se`
[2] Saab AB, Nettovägen 6, 175 41 Järfälla, Sweden

Abstract. We introduce a proof-producing symbolic execution tool for formal verification of P4 programs. The tool has been implemented using the interactive theorem prover HOL4 and results are proved sound with respect to the the HOL4P4 formalisation of the P4 language. Most notably, this is a general tool for proving functional correctness that can be applied to entire real-world P4 programs.

Keywords: Theorem Proving · Formal Verification · Domain-Specific Languages.

1 Introduction

P4 is the most popular domain-specific language for the data planes of programmable network elements, the hardware which the software-defined networking (SDN) paradigm is built on. A software-defined network element separates packet processing functionality into the control plane (which updates tables with network topology and routing information) and the data plane (which performs the actual bit-by-bit processing of packets). The interface between these two planes is kept minimal. There exists a diverse range of targets for P4 from terabit-bandwidth switches to network interface controllers and the Linux kernel [23, 9].

Network elements that govern communication in critical systems must undergo strict high-assurance certification. For the highest levels of certification, formal verification is required - and for this, using an interactive theorem prover (ITP) is ideal. The simplicity of P4 (absence of pointers, unbounded loops and recursion) means that exhaustive formal reasoning needs to perform less computation, rendering heavy-duty formal methods feasible for large programs.

The contributions[3] presented in this paper are:

1. A concurrent, language-agnostic HOL4 symbolic execution framework (Section 4).

[3] The code for the tool presented in this paper can be found in the Github repository at https://github.com/kth-step/HOL4P4 and the version as of writing this paper at the tag VSTTE2024. The parts related to symbolic execution are found in the `hol/symb_exec` directory.

2. A formal verification tool for P4 programs that only requires minimal annotation (Section 5), including automated overapproximation methods for interaction with external functions and data (Section 5.2), and usage of multiple small-step semantics of different scope and granularity for increased efficiency (Section 5.3).
3. Performance evaluation for a case study for the above (Section 7).

2 Background and Related Work

2.1 The P4 Language

P4 programs are structured as a pipeline consisting of programmable blocks. These programmable blocks are either parser blocks or control blocks. Parser blocks are similar to finite state machines: they consist of parser states which typically extract the bits of the input packet into P4 headers (similar to C structs), with unstructured jumps between them. Control blocks consist of (possibly branching) sequences of look-ups in tables configured by the control plane, without loops. When verifying a P4 program, the content of some tables may be known, while the content of others may be unknown.

Throughout P4 programs, external functions and objects (typically implemented by the architecture in an FPGA or directly in ASICs) provide more complex functionality such as hashing and checksum computation.

To illustrate significant language features, Figures 1 and 2 contain snippets of a simplified version of the VSS example found in the P4 Specification [22]. The program parses an Ethernet and an IPv4 header of an input packet, validates the IPv4 checksum, then sets the output port based on the IPv4 destination address. Figure 1 shows the parser block `TopParser` with the initial parser state `start`. The `extract` method fills its struct arguments with bits from the raw input packet `b`. The fields of the structs can then be accessed, as seen on line 8 of Figure 1. The `select` expression matches the argument - here, the `etherType` field of `p.ethernet` - to value sets in a list (here the singleton set {`0x8000`} and the set of all 16-bit values), and chooses the corresponding next parser state for the transition statement based on this match.

The external `Checksum16` object `ck` in Figure 1 can compute checksums from IPv4 headers. External objects and functions do not have implementations written in P4: they are provided by the target platform.

The `verify` statement checks that the predicate given as first argument holds. If not, the parser block will transition to the `reject` state and set `parseError` outside the programmable blocks to `verify`'s second argument.

The parser block is finished upon transitioning to the `accept` or `reject` state, after which the out-directed block parameter `p` is copied out, to be used later. Then, execution of the control block `TopPipe` shown in in Figure 2 commences: `p` and `parseError` are copied in, and the content of the apply block between lines 18 and 24 is executed.

The table application `ipv4_match.apply()` on line 20 looks up the value in the `key` of `ipv4_match` (`p.ip.dstAddr`) in a table stored in the control plane:

```
parser TopParser(packet_in b,
                 out Parsed_packet p) {
  Checksum16() ck;

  state start {
    b.extract(p.ethernet);
    transition
    select(p.ethernet.etherType) {
      0x0800: parse_ipv4;
      _: reject;
    }
  }

  state parse_ipv4 {
    b.extract(p.ip);
    ck.clear();
    ck.update(p.ip);
    verify(ck.get() == 16w0,
           error.IPv4ChecksumError);
    transition accept;
  }
}
```

Fig. 1. Parser block snippet

```
control TopPipe(inout Parsed_packet p,
                in error parseError,
                out OutControl outCtrl) {
  action Drop_action() {
    outCtrl.outputPort = DROP_PORT;
  }

  action Set_oport(PortId port) {
    outCtrl.outputPort = port;
  }

  table ipv4_match {
    key = {p.ip.dstAddr: lpm;}
    actions = {Drop_action;
               Set_oport;}
  }

  apply {
    if (parseError == error.NoError) {
      ipv4_match.apply();
    } else {
      Drop_action();
    }
  }
}
```

Fig. 2. Control block snippet

note that the entries of the table are left unspecified by the P4 program. `lpm` signifies a longest-prefix match, and the outcome of this match is the invocation of either of the `actions`: `Drop_action` or `Set_nhop`, with arguments provided by the control plane.

2.2 Related Work

Symbolic execution [13] summarises many execution traces into one, at the expense of having to fork execution into multiple paths upon encountering branches in the control flow. While the classic notion of symbolic execution allows omitting traces from analysis (i.e. underapproximation), this trace exclusion does not allow for proving safety properties. Conversely, safety properties can be proved using trace overapproximation. As such, the flavour of symbolic execution used in this work is in a sense "trace-complete" or "overapproximating".

At least since the 90s [21], it has been known that theorem provers offer a shortcut to implementing something akin to symbolic execution by evaluating terms with native free variables in place of concrete values. In the literature, this shallow embedding of symbolic execution is also known as *symbolic simulation* or *symbolic evaluation*. This technique implements symbolic branching in the metalanguage, which does not yield a formal definition for which to prove e.g. termination (of the symbolic execution itself). However, the absence of a termination proof has no adverse effect on the soundness of the analysis.

Symbolic Execution Using HOL4 Holfoot [24, 25] is a tool that uses shallowly embedded symbolic execution and an abstract separation logic to verify functional correctness of programs in the Smallfoot language. Note that using separation logic for P4 is unnecessary, since the language does not involve pointer arithmetic.

Collavizza and Gordon [4] have used shallowly embedded symbolic execution to formally verify properties of Java programs, using an approach that depends in part on unverified reasoning.

Campbell and Stark [3], and more recently Kanabar et al. [11], use shallowly embedded symbolic execution based on the `step` library of Anthony Fox [6] for test generation, compiler verification and cross-validation of ISA models.

Lindner [14] et al. describe a deeply embedded symbolic execution for the unstructured BIR language used for binary analysis. The deeply-embedded approach allows to prove a formal metatheory about the symbolic execution itself.

In comparison to the above, our work is unique through the usage of multiple small-step semantics of different scope and granularity to improve efficiency of symbolic evaluation, overapproximation techniques and the contribution of a language-agnostic symbolic execution framework.

There exists a plethora of tools for other ITPs that use symbolic execution-based techniques for formal verification[16, 12, 19, 8]: to the authors' knowledge no comprehensive review comparing symbolic execution in different ITPs exists. The authors could also not identify any other ITP-based work that combines multiple symbolic semantic styles and uses automated overapproximation.

P4 Verification P4Cub [18] is an intermediate representation for P4 verification with both big-step and small-step semantics formalised in Coq that has been used for a non-proof-producing program verifier. Wang et al. introduce Verifiable P4 [26], a verification system implemented in Coq that uses semi-automatic symbolic execution techniques together with reasoning in program logic to prove correctness of P4 control blocks. In contrast, the symbolic execution presented in this paper is fully automatic and covers the entire P4 pipeline. Leapfrog [5] is an equivalence checker for P4 parsers implemented in Coq.

Vera [20] and P4pktgen [17] both use underapproximating symbolic execution to find bugs in P4 programs. ASSERT-P4 [7] and p4v [15] are other non-proof-producing verification tools for P4.

3 HOL4P4 Semantics

To implement the symbolic execution approach of this paper in an ITP, an executable semantics is needed: here, "executable" means that reduction results can be computed directly using standard evaluation facilities of the ITP. This executable semantics is a deeply embedded function $\mathrm{small}(E, \rho, n)$ that computes the result of n reductions (i.e. the transitive closure of n small-step reductions) of the initial state ρ in the static environment E according to the semantics' reduction rules. This work is built on the HOL4P4 semantics of Alshnakat et al. [1][4], presented in brief below, simplified for the sake of presentation. For brevity, this executable semantics will be referred to simply as the "small-step semantics" in the rest of the paper.

The semantics consists of four layers: architecture, frame, statement, and expression. The architecture layer connects the programmable blocks and governs input and output. The frame layer ensures that the frame resulting from the most recently called function is passed along to the statement semantics, and handles function return.

All architecture-level reductions are made in the presence of a static environment E, which contains the program and models of external functions. The architecture-level state $\rho = (\overline{io}, \alpha, i, \gamma_G, \overline{\Phi}, t)$ consists of lists of incoming and outgoing packets $\overline{io}$, an external state (of the runtime) α, the current block index i, a global store γ_G, a frame stack $\overline{\Phi}$ and a status t. The status is used for signaling function return and parser block transition from the statement-level to the frame- and architecture-level semantics. Each frame in $\overline{\Phi}$ corresponds to a called function, with the top frame popped upon function return: functions are restricted to manipulating their own frames. A frame $\Phi = s \,{}^{f}_{\gamma}$ consists of the associated function's name f, the statement currently being reduced s, and a variable store γ holding the values of local variables.

The frame layer reduces (α, γ_G, $\overline{\Phi}$, t) to tuples of the same type, with table and function signatures from E in the local context. The statement layer has the same signature as the frame layer, but with a single frame Φ instead. The expression layer reduces expressions e to expressions (and a new frame in case of function call), with the function signatures from E and the current scopes (γ and γ_G) in the local context.

Figure 3 showcases the statement-level semantic rule for table application using a fully reduced key $v_1,...,v_n$ (a separate rule reduces the key using the expression-level semantics). As an example, consider line 20 of Figure 2. Here, the key consists of a single element: the value of `p.ip.dstAddr`. Using the static table information stored in T (the match kind `lpm`), the key will be matched to entries of `match_ipv4` stored in α. The match result is an action f' with arguments $v'_1,...,v'_m$, and the **apply** statement reduces to a call to this action. The status is retained as **run**, signifying regular execution.

Apply

$$\frac{T\,(\mathit{tbl}, v_1,...,v_n, \alpha) = (f'\,,\ v'_1,...,v'_m)}{T\;F \vdash (\alpha, \gamma_G, [\mathbf{apply}\ \mathit{tbl}\ v_1,...,v_n]\,{}^{f}_{\gamma}, \mathbf{run}) \rightarrow (\alpha, \gamma_G, [\mathbf{null} := f'(v'_1,...,v'_m)]\,{}^{f}_{\gamma}, \mathbf{run})}$$

Fig. 3. Semantic rule for table application

The statements consist of the standard assignment, block, conditional and return and the P4-specific extern statement ■, **transition** and **apply**.

[4] A small extension to HOL4P4 is used, which adds value set types used for matching in **select** expressions.

The extern statement ■ can implement any behaviour that modifies α and the local γ. Reduction of ■ uses the function name f of the current frame, and looks up the implementation in the static environment: $E(f) = ext$, then uses it to update α and γ: $ext(\alpha, \gamma_G, \gamma) = (\alpha', \gamma', t')$, where t' is the resulting status.

transition is similar to a jump, with possible targets being different parser states in the same programmable block.

Finally, **apply** matches a list of expressions $\overline{e}$ against a table *tbl* stored in α: $\text{match}(\overline{e}, tbl, \alpha) = a(\overline{arg})$. Then, **apply** is reduced to a call to the result $a(\overline{arg})$. Notably, ■ and **apply** are the only statements that interact with α.

The expressions consist of standard arithmetic and Boolean operators together with function calls (generating new frames when reduced) and the **select** expression, which is typically found together with the **transition** statement. **select** can be thought of as matching against membership in value sets. Values are modeled with bit-level granularity, and we represent e.g. a bit-string of width 4 as $b_1b_2b_3b_4$, with $b_1 \dots b_4$ being the individual bits.

4 Symbolic Execution Framework

4.1 Shallow Symbolic Execution

The goal of the symbolic execution approach presented here is to provide a generic tool that requires only an executable small-step semantics and minimal proof additions to enable symbolic execution, while still allowing for incremental performance improvements.

The basic idea for re-using an executable semantics for symbolic execution is as follows:

1. To model symbolic values, use HOL4 native free variables in place of concrete values.
2. Prevent state explosion by restricting evaluation of arithmetic dependent on free variables when reducing expressions to values: for example, not unfolding the definition of (primitive) addition of bitstrings in $b_1b_2b_3{+}010$, where b_1, b_2 and b_3 are HOL4 free variables of bit type. The result can then get assigned to a variable as-is.
3. Fork the symbolic execution whenever a branch is dependent on a HOL4 free variable, maintaining a list of n-step theorems with path conditions:

$$P \implies small(\rho, n) = \rho'$$

   stating that given the path condition P, n small-step execution steps from ρ results in ρ'. Note that P and ρ can share HOL4 free variables. *small* is a partial function that is undefined when a run-time error occurs, which can be implemented using an option type.

With this approach, it is not necessary to modify the executable semantics in any way. The maintenance of separate n-step theorems for separate paths means that the semantics does not have to formalise the notion of paths or path conditions.

Around 200 additional lines of HOL4 proof scripts are needed for a minimal implementation of HOL4P4 symbolic execution. Note that the symbolic execution can not in general produce a strongest postcondition, since it allows overapproximation. If no overapproximation was used, the strongest predicate obeyed by the final state of every path can be informally thought of as the strongest postcondition.

4.2 N-chotomy Theorems

To prove no execution traces are dropped by the symbolic execution, it is necessary to prove an *n-chotomy theorem* every time the symbolic execution is forked. This theorem exhaustively enumerates as disjuncts all possible outcomes that might result from values of the free variables in the construct that the semantics branches on. For example, the n-chotomy theorem of the `select` expression on line 8 of Figure 1 would state that $b_1 \ldots b_{16} \in \{0x8000\} \vee b_1 \ldots b_{16} \notin \{0x8000\}$, where $b_1 \ldots b_{16}$ is the result of reducing `p.ethernet.etherType`.

After the n-chotomy theorem has been proved, some disjuncts may be immediately ruled out (pruned) using the path condition. Then, the symbolic execution is forked, with each new path condition gaining (by conjunction) one disjunct from the n-chotomy theorem. The n-chotomy theorems themselves are stored in a tree structure that keeps track of which paths resulted from which choices: every non-leaf node holds an n-chotomy theorem and every leaf holds a unique identifier that pairs it with a path (n-step theorem). The n-chotomy tree is used later when proving properties about the result of the symbolic execution.

4.3 Abstract Symbolic Execution Algorithm

Part of the symbolic execution machinery is language-agnostic and has therefore been written as an abstract framework that can be instantiated to perform symbolic execution of other languages. The language parameters are the following functions:

1. `regular_step`: takes the current n-step theorem and performs m regular execution steps starting from the current state, then composes the result with the previous, yielding an $n+m$-step theorem. The simplest possible implementation would just rely on direct evaluation of the executable semantics used, doable in less than 10 LoC in HOL4.
2. `should_branch`: looks at the current n-step theorem and decides whether to fork the symbolic execution or not. If yes, it returns an n-chotomy theorem and a list of forked n-step theorems, to whose path conditions the cases of the n-chotomy theorem have been added.
3. `is_finished`: looks at the current n-step theorem and decides whether execution on that path has finished or not.

The final output of the symbolic execution algorithm is a list of n-step theorems (as described in Section 4) and a tree with n-chotomy theorems (as described in Section 5.2). Provided the implementations of `regular_step`, `should_branch` and `is_finished` are thread-safe, the abstract symbolic execution framework is capable of running them concurrently. Concurrency is achieved via a simple mutex-based job scheduler that synchronizes the common datastructure holding the n-chotomy tree and n-step theorems.

4.4 Coarse-Grained Semantics

The semantics of a language may have notions of locality that permit simplified reductions inside different locales. Perhaps the most obvious one is the locale of functions: restricting the semantics to reductions inside individual functions can strip the semantics of much complexity. Another optimization can be done for semantics with multiple layers, where collapsing multiple steps into one can effectively "cache" computations from layers above. The HOL4P4 case of this optimization is shown in Figure 4. This paper uses "coarse-grained semantics" to refer to the type of semantics with restrictions and optimizations described by the above paragraph relative to the base small-step semantics.

To use a coarse-grained together with a fine-grained small-step semantics, it is necessary to compose their executions: this can be done via the usual composition theorems for small-step execution so long as a soundness theorem, like Proposition 1, is proved. This is stated relative to some *update* and *proj*: the function *proj* extracts information from ρ to construct a coarse-grained state s, and *update* takes a resulting coarse-grained state and modifies ρ accordingly.

Proposition 1 (Soundness of Coarse-Grained Semantics). *If* $proj(\rho) = s$*,* $coarse(s) = (s', n)$ *and* $update(\rho, s') = \rho'$*, then* $small(\rho, n) = \rho'$ *.*

5 HOL4P4 Symbolic Execution

5.1 Regular HOL4P4 Execution Steps

"Regular" steps are steps where no overapproximation or fork of the symbolic execution occurs. However, regular steps may still involve operations using symbolic variables. The regular execution step implementation is based on the call-by-value reduction engine `CBV_CONV` [2] with a custom `compset` of rewrites and conversions. In addition to this, rewriting using the path condition is also performed as needed: typically this involves restricting evaluation of some functions (using `RESTR_CBV_CONV`), rewriting, and then resuming evaluation without this restriction. Some functions involving arithmetic are never evaluated in the main loop to prevent state explosion - instead, they are selectively extracted and evaluated separately. At the end, the result of evaluating one step from the current final state is composed with the existing n-step theorem, forming an $n + 1$-step theorem.

The main issue with this approach is the large size of the theorems involved. We have solved this in part by introducing definitions for the (very large) static environment E and its components, which are unfolded and folded back as needed. For parts of the mutable state that may contain HOL4 variables, this is more difficult, but is in part solved by a type of approximation described in Section 5.2. Note that in practice, the HOL4P4 symbolic execution will always terminate using repeated regular steps: the control blocks are loop-free, and realistic P4 programs either hard-code bounds on variable-length headers and extension headers, or limit their size by a preceding field determining length.

5.2 HOL4P4 N-chotomy Theorems

Other than the conditional statement, the two other branching HOL4P4 language constructs are **select** expressions (in **transition** statements) and **apply** statements. **select** expressions match a value v to value sets $V_1, \ldots, V_n$ in a list ranked from 1 to n, with the outcomes being parser state names $st_1, \ldots, st_n$. In that case, the n-chotomy theorem would be

$$v \in V_1 \lor (v \in V_2 \land v \notin V_1) \lor \ldots \lor (v \in V_n \land v \notin V_{n-1} \cup \ldots \cup v \notin V_1)$$

with every outcome ruling out the outcomes ranked above it. Note that the value v can be a complex HOL4 term involving multiple free variables.

The **apply** case exposes some quirks of P4 and the networking setting, due to the complex match kinds and reading external table content (key-value pairs) from the control plane. For tables with known content, the resulting n-chotomy theorem looks rather similar to that for **select**.

The case when table entries are not known at verification time (for example, the table `ipv4_match` in Figure 2) is more interesting. Since in P4, a table *tbl* must list all possible actions $a_1 \ldots a_n$ that can result from a match, the n-chotomy can be stated in terms of the resulting action. Such an n-chotomy theorem can only be stated assuming the entries of *tbl* in α are well-formed in this respect:

$$\begin{aligned}
&\text{well-formed}(\mathit{tbl}, \alpha) \Rightarrow \\
&(\exists b_1 \ldots b_{m1}.\ \text{match}(v, \mathit{tbl}, \alpha) = a_1(b_0 \ldots b_{m1})) \lor \ldots \\
&\ \lor (\exists b_1 \ldots b_{mn}.\ \text{match}(v, \mathit{tbl}, \alpha) = a_n(b_0 \ldots b_{mn}))
\end{aligned}$$

The arguments to the resulting actions have been restricted to single bitstrings of length m for the sake of presentation. Typically, the well-formedness assumptions are inserted into the initial path condition: this is justified by the P4 Specification stating the control-plane runtime is responsible for only inserting valid entries into tables. Assumptions on the values of arguments in the match result can also be inserted into the initial path condition. This is used when one has partial information of the table content: for example, that a match result will occur for all IP addresses in the local network.

Although the concrete semantics of extern objects can be modeled from their written descriptions, we often choose to overapproximate the outcome of e.g. checksum computations by using fresh HOL4 variables for the individual bits of their result. This is done by providing a list of the externs to potentially be approximated. Then it is decided whether to approximate or not based on the

γ of the top frame, and theorems are proved that overapproximate the effect of the extern (some sub-function of the extern implementation) for that γ, using existentially quantified HOL4 variables. This approximation theorem is then used like a 1-chotomy theorem by the rest of the framework.

5.3 HOL4P4 Coarse-Grained Semantics

For the HOL4P4 small-step semantics, many design choices were made that make it more suitable for a formal metatheory, but less efficient for computation. The most severe issue is that every single execution step must traverse every level from top (architecture) to bottom: first, an architectural reduction is chosen, then a frame reduction, then a statement and expression-level reduction, every layer forwarding appropriate information to the next. A more efficient approach

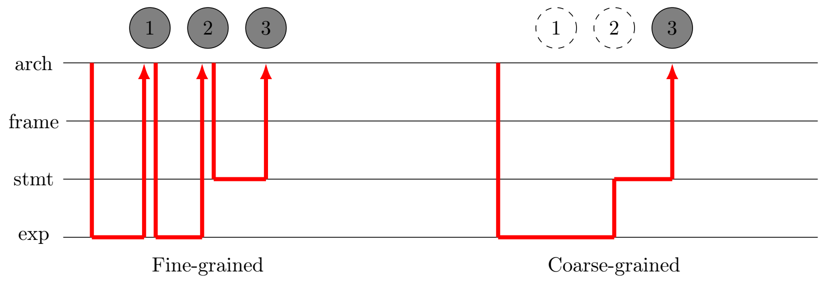

Fig. 4. Reductions on semantic layers: fine-grained vs. coarse-grained

that can be implemented as a coarse-grained executable semantics is to keep reducing on a lower layer (for example, the expression layer) until the expression is fully reduced, similar to a reduction in big-step semantics, and only then return to the layer above. This is illustrated in Figure 4.

Additionally, if the coarse-grained semantics is limited to a simple fragment of the HOL4P4 syntax (i.e., no ■, no **apply**, no function call and no **return**), it is possible to simplify the reductions greatly: the external state α can be disregarded since it is only interacted with by ■ and **apply**. The static environment E is not needed, since on the statement level it is only used for function call and **apply**. Since no function return is allowed, the status and all frames except the topmost can also be disregarded. This partiality is implemented by having the coarse-grained semantics halt once a disallowed statement is about to be reduced and return the current statement, allowing for the small-step semantics to continue from the resulting state. In practice, it is preferable to use multiple coarse-grained semantics that first treat a more complex statement using the necessary elements of ρ and E, then continue using the above minimal coarse-grained semantics.

6 Proving Contracts for HOL4P4

One use case of the symbolic execution result is to prove contracts, as defined in Definition 1: flexible enough to accommodate any functional correctness property. Note that the symbolic execution result is independent of this notion of contracts and can be used for other ends, e.g. for proving simulation theorems or in interactive proofs.

Definition 1 (Contract). *The contract $\{P\}\, E\, \{Q\}$ holds iff for the precondition P, static environment E and postcondition Q*

$$\forall \rho.\ P(\rho) \Rightarrow \exists n > 0.\ \exists \rho'.\ \mathrm{small}(E, \rho, n) = \rho' \wedge Q(\rho')\,.$$

Note that ρ and ρ' contain not only variable maps, but also the program currently being reduced. To avoid spurious proofs where Q only holds for some intermediate states, Q can include a finish condition (e.g. "last block was just reduced, no more input packets pending"). Also, since HOL4 functions are deterministic ρ' can be existentially quantified.

Contracts are obtained by proving that the postcondition holds for every final state of the n-step theorems resulting from the symbolic execution, after which all resulting contracts are unified to one by using the n-chotomy theorems. Should the postcondition not be provable for some path, this will be printed, providing a possible counterexample. The contract derivation procedure is fully automatic. Note that the overapproximation described in Section 5.2 allows proving a valid contract as long as it does not introduce new paths and if any free variables introduced does not affect the postcondition.

If the end goal is not to prove a contract with a provided postcondition, another approach could be to instead prove the postcondition of the symbolic execution: $Q(\rho) = (P_1(\rho) \Rightarrow \rho = \rho_1) \wedge \ldots \wedge (P_n(\rho) \Rightarrow \rho = \rho_n)$, where n is the number of paths. This could then also be used to prove the contract.

7 Evaluation

As a small case study, the tool has been applied to the P4 implementation of IPsec tunneling by Hauser et al. [10].[5]

The overapproximation techniques have been implemented for externs from the V1Model architecture, allowing to abstract from large structures and computations which would otherwise make verification infeasible. During development, we have also constructed a small validation suite of P4 programs with pre- and postconditions. Some performance numbers are shown in Table 1: lines of code, total verification time, average reduction times for fine-grained and coarse-grained, and the number of paths at the end. The coarse-grained average reduction time is obtained by dividing the sum of all coarse-grained reduction

[5] See `hol/symb_exec/basicScript.sml`: the contract establishes the postcondition `postcond` given the precondition `path_cond`.

times with the corresponding number of fine-grained reductions. To get an idea of how the tool scales, the IPsec example (`basic.p4`) is compared to one of the validation tests (`table-unknown.p4`), an order of magnitude smaller in size. To verify programs an order of magnitude larger than `basic.p4` in reasonable time, one could prove multiple contracts for different parts of the P4 pipeline and combine them. The experiments were ran on a laptop with an Intel i7-8550U CPU.

Program	LoC	Total Time	Avg. Red. Time (fine-grained)	Avg. Red. Time (coarse-grained)	Paths
`basic.p4`	396	30m	250ms	120ms	82
`table-unknown.p4`	78	34s	80ms	30ms	12

Table 1. Performance measurements

The generic symbolic execution framework[6] is 500 LoC and the P4-specific part 7500 LoC, of which 5000 for the coarse-grained semantics and its soundness proof. To validate the claim that the symbolic execution framework is language-agnostic, it was instantiated for a simple imperative language with while loops, which took about an hour of work and required 300 LoC.

8 Conclusions

We have presented a theorem-grade symbolic execution tool for P4 that is able to deal with quirks of the networking domain, by using overapproximation for external implementations and unknown table configurations.

We show how, in the ITP setting, symbolic execution can be obtained easily from the executable small-step semantics of a real language. Furthermore, a significant part of the symbolic execution machinery is generic and can be re-used for executable models of other languages. Also, we show how the performance of this style of symbolic execution can be increased by pairing it with a coarse-grained semantics.

For future work, the construction of a benchmark suite of P4 programs with functional correctness properties would be helpful when comparing the performance of different verification tools.

Acknowledgments. This work has been supported by the SEMLA project financed by Vinnova (Sweden's Innovation Agency). We would also like to thank the reviewers for their valuable feedback.

[6] The framework consists of `hol/symb_exec/symb_execScript.sml` and `hol/symb_exec/symb_execLib.sml`.

References

1. Alshnakat, A., Lundberg, D., Guanciale, R., Dam, M.: HOL4P4: Mechanized small-step semantics for P4. Proceedings of the ACM on Programming Languages **8**(OOPSLA1), 223–249 (2024)
2. Barras, B.: Programming and computing in HOL. In: International Conference on Theorem Proving in Higher Order Logics. pp. 17–37. Springer (2000)
3. Campbell, B., Stark, I.: Extracting behaviour from an executable instruction set model. In: 2016 Formal Methods in Computer-Aided Design (FMCAD). pp. 33–40 (2016). https://doi.org/10.1109/FMCAD.2016.7886658
4. Collavizza, H., Gordon, M.: Integration of Theorem-Proving and Constraint Programming for Software Verification. Ph.D. thesis, Laboratoire I3S (2009)
5. Doenges, R., Kappé, T., Sarracino, J., Foster, N., Morrisett, G.: Leapfrog: Certified equivalence for protocol parsers. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 950–965. PLDI 2022 (2022). https://doi.org/10.1145/3519939.3523715
6. Fox, A.: Improved tool support for machine-code decompilation in HOL4. In: 6th International Conference on Interactive Theorem Proving (ITP 2015). pp. 187–202 (2015). https://doi.org/10.1007/978-3-319-22102-1_12
7. Freire, L., Neves, M., Leal, L., Levchenko, K., Schaeffer-Filho, A., Barcellos, M.: Uncovering bugs in P4 programs with assertion-based verification. In: Proceedings of the Symposium on SDN Research. SOSR '18 (2018). https://doi.org/10.1145/3185467.3185499
8. Gourdin, L., Bonneau, B., Boulmé, S., Monniaux, D., Bérard, A.: Formally verifying optimizations with block simulations. Proceedings of the ACM on Programming Languages **7**(OOPSLA2), 59–88 (2023)
9. Hadi Salim, J., Chatterjee, D., Nogueira, V., Tammela, P., Osinski, T., Haleplidis, E., Sambasivam, B., Gupta, U., Jain, K., Sethuramapandian, S.: Introducing P4TC-a P4 implementation on Linux kernel using traffic control. In: Proceedings of the 6th on European P4 Workshop. pp. 25–32 (2023)
10. Hauser, F., Häberle, M., Schmidt, M., Menth, M.: P4-IPsec: Site-to-site and host-to-site VPN with IPsec in P4-based SDN. IEEE Access **8**, 139567–139586 (2020)
11. Kanabar, H., Fox, A.C., Myreen, M.O.: Taming an authoritative Armv8 ISA specification: L3 validation and CakeML compiler verification. In: 13th International Conference on Interactive Theorem Proving (ITP 2022). Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2022)
12. Keller, C.: Tactic program-based testing and bounded verification in Isabelle/HOL. In: Tests and Proofs: 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings 12. pp. 103–119. Springer (2018)
13. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)
14. Lindner, A., Guanciale, R., Dam, M.: Proof-producing symbolic execution for binary code verification. arXiv preprint arXiv:2304.08848 (2023)
15. Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N., Foster, N.: p4v: Practical verification for programmable data planes. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. p. 490–503. SIGCOMM '18 (2018). https://doi.org/10.1145/3230543.3230582

16. Matthews, J., Moore, J.S., Ray, S., Vroon, D.: Verification condition generation via theorem proving. In: Logic for Programming, Artificial Intelligence, and Reasoning: 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006. Proceedings 13. pp. 362–376. Springer (2006)
17. Nötzli, A., Khan, J., Fingerhut, A., Barrett, C., Athanas, P.: P4pktgen: Automated test case generation for p4 programs. In: Proceedings of the Symposium on SDN Research. pp. 1–7 (2018)
18. Peterson, R., Campbell, E.H., Chen, J., Isak, N., Shyu, C., Doenges, R., Ataei, P., Foster, N.: P4cub: A little language for big routers. In: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 303–319 (2023)
19. Ravindran, B.: Formal verification of memory preservation of x86-64 binaries. In: Computer Safety, Reliability, and Security: 38th International Conference, SAFECOMP 2019, Turku, Finland, September 11–13, 2019, Proceedings. vol. 11698, p. 35. Springer Nature (2019)
20. Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., Raiciu, C.: Debugging P4 programs with Vera. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. pp. 518–532 (2018)
21. Strother Moore, J.: Symbolic simulation: An ACL2 approach. In: International Conference on Formal Methods in Computer-Aided Design. pp. 334–350. Springer (1998)
22. The P4 Language Consortium: $P4_{16}$ language specification (2023), https://p4.org/p4-spec/docs/P4-16-v1.2.4.html
23. Tu, W., Ruffy, F., Budiu, M.: Linux network programming with P4. In: Linux Plumb. Conf (2018)
24. Tuerk, T.: A formalisation of Smallfoot in HOL. In: International Conference on Theorem Proving in Higher Order Logics. pp. 469–484. Springer (2009)
25. Tuerk, T.: A separation logic framework for HOL. Tech. rep., University of Cambridge, Computer Laboratory (2011)
26. Wang, Q., Pan, M., Wang, S., Doenges, R., Beringer, L., Appel, A.W.: Foundational verification of stateful P4 packet processing. In: 14th International Conference on Interactive Theorem Proving (ITP 2023). Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2023)