A Validation Methodology for OCaml-to-PVS Translation[⋆]

Xiaoxin An, Amer Tahat[✉], and Binoy Ravindran

Virginia Tech, Blacksburg VA 24061, USA
{xxan15,antahat,binoy}@vt.edu

Abstract. We present a methodology, called OPEV, to validate the translation between OCaml and PVS, which supports non-executable semantics. This validation occurs by generating large-scale tests for OCaml implementations, generating test lemmas for PVS, and generating proofs that automatically discharge these lemmas. OPEV incorporates an intermediate type system that captures a large subset of OCaml types, employing a variety of rules to generate test cases for each type. To prove the PVS lemmas, we developed automatic proof strategies and discharged the test lemmas using PVS Proof-Lite, a powerful proof scripting utility of the PVS verification system. We demonstrated our approach on two case studies that include two hundred and fifty-nine functions selected from the Sail and Lem libraries. For each function, we generated thousands of test lemmas, all of which are automatically discharged. The methodology contributes to a reliable translation between OCaml and PVS.

Keywords: Translation Validation · PVS · OCaml

1 Introduction

Verifying a "translator" that translates a specification written in one language to another language is of fundamental interest in many settings, such as compilers, assemblers, and interpreters. A rigorous methodology that can be used to verify the translation is *refinement proving*. This method requires a translation into a formal verification language to generate a formal certificate. The translated model, whether it was generated manually or mechanically, must comply with the intended meaning of the program being certified for the certificate to be valid. For example, seL4's formal certification used a translation from a subset of C called C_0 into Isabelle/HOL [1]. The conformance relationship was established based on a refinement proof that required significant human effort [2].

[⋆] This is the authors version of the work posted here per publisher's guidelines for your personal use. Not for redistribution. The final authenticated version was published in the Proceedings of the 12th International Symposium on NASA Formal Methods, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020 and is available online at: `https://doi.org/10.1007/978-3-030-55754-6_12`.

However, the validation of translation between different languages is exacerbated when languages at either end of the "translation pipe" have no formal semantics, which is the case in many settings. This precludes establishing a two-way equivalence relationship between the source and the destination languages. In such cases, a testing methodology is perhaps a more effective verification strategy to establish equivalence between the two specifications. For example, Lem [3] is a specification language that used to translate mathematically rigorous models of multiple ISAs into different theorem provers and into OCaml. For instance, Lem can be translated to OCaml for emulation of testing as well as to Isabelle/HOL, Coq, HOL4, and other languages. The OCaml translation was validated via predefined tests written in the Lem language [3].

Though translators can be validated by testing [4], this does not have the same level of rigor as refinement proofs and does not require formal semantics for the target languages. Testing requires that both languages are executable. However, some specifications with formal semantics can be either executable or non-executable, and the results of the non-executable specification cannot be directly calculated. For example, in the Prototype Verification System (PVS) [5], PVSio [6], the emulator utility in PVS, can only execute a subset of the functional specifications in PVS. This is a limitation of many theorem provers, not just PVS – their specification languages are designed to state and prove theorems, but not execute. In fact, large subsets of many provers' powerful specifications are non-executable. This downside can be overcome by stating theorems on these specifications that capture the intended behaviors and proving them, mostly interactively – a highly labor-intensive effort. For example, validation of the CompCert compiler [7] involved 100K lines of Coq proof.

Motivated by these concerns, we present a test-and-proof methodology to validate the translation between two different languages with one of them supporting non-executable semantics. Our methodology (Section 2), folded into a tool called OPEV (for "OCaml-to-PVS Equivalence Validation"), takes an OCaml program and a corresponding PVS implementation as input. From these inputs, OPEV automatically generates large-scale test cases, which are directly executed on the OCaml program and also used for constructing a large number of test lemmas on the PVS specification. The test lemmas are proved automatically using proof strategies. The results are compared to establish equivalence.

We demonstrate OPEV by using it to validate a manually implemented OCaml-to-PVS translation and a Sail-to-PVS parser (Section 3.2) that we manually developed. This parser includes 2,763 LOC and was used to translate 7,542 LOC of Lem code to 10,990 LOC of PVS implementation. OPEV generated and proved 458,247 test lemmas for these two case studies, and detected 11 errors (Section 3). The development of OPEV took 3 person-months and the effort to develop and validate the translator took 5 person-months.

This paper's central contribution is the proposed, semi-automatic test-and-proof methodology for validating translators supporting non-executable specifications. In principle, the OPEV methodology can be applied to any pair of target languages where one has non-executable semantics.

2 OPEV: OCaml-to-PVS Equivalence Validation

OPEV's methodology increases the trust in the translated OCaml code into PVS. The translation can be automatic (for a subset of OCaml) or manual. Moreover, OPEV enables proving auto-generated test cases from the target language OCaml to PVS, where the inputs/ouputs have identical names and arguments.

2.1 OPEV Workflow

Figure 1 shows the OPEV workflow. In OPEV, we have designed an intermediate type system, Subsection 2.2, to capture the commonality of OCaml and PVS types, which are restricted to a subset of the complete OCaml and PVS types. OPEV parses the PVS and OCaml sources to construct the intermediate type annotations for each function. With these annotations, OPEV generates random test cases for every OCaml and PVS function. OPEV then runs the OCaml test cases to obtain the test results, translates the OCaml test results to PVS, and constructs PVS test lemmas using the PVS test cases and translated results. The test lemmas are directly employed as *test oracles*, which can be automatically verified using manually implemented, generic PVS proof strategies. If the test lemmas are proved to be false, we know that there are mismatches in the OCaml-to-PVS translation. Thus, we investigate the cases and try to detect the reasons. The total codebase of OPEV is 3,783 LOC.

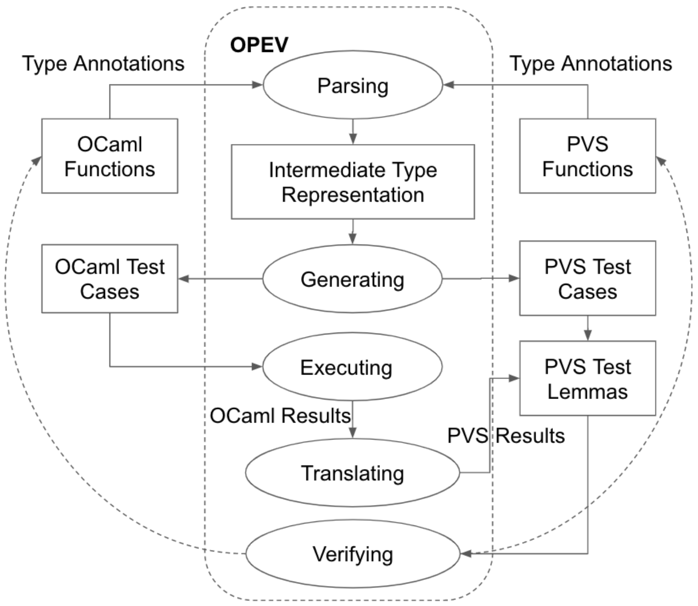

Fig. 1. The OPEV workflow.

Extensibility. OPEV has already incorporated the semantics of a large subset of OCaml and PVS for automatic test-generation. To ensure that OPEV can be extended to incorporate more types in the future, we represent the generated test cases and testing results in the `string` format to circumvent the real type system of OCaml and PVS.

Listing 1.1. A sample PVS reverse function.

```
rev[A:TYPE](l:list[A]) : RECURSIVE list[A] =
CASES l OF
cons(x, xs): append(rev(xs), cons(x, null))
ELSE null
ENDCASES
MEASURE length(l)
```

For example, in Listing 1.1, suppose we randomly generate [1, 6, 8] as the test value for the argument `l` of function `rev`. We then construct a string "let res = rev [1; 6; 8];;" as the OCaml command and delegate it to the OCaml `Toploop` library to execute the command. The result can be fetched from the `res` variable, which has the value [8; 6; 1]. Then OPEV parses the result according to its type and composes a PVS test lemma, such as `th_rev` in Listing 1.2.

Listing 1.2. A sample of OPEV PVS test lemmas for `rev` function.

```
 th_rev: LEMMA rev((: 1, 6, 8 :)) = ((: 8, 6, 1 :))
```

The lemma is also written in the string format. This string-format representation allows us to avoid writing various functions for different argument types and simplifies the extension of OPEV.

Non-Executable Semantics. We construct PVS test lemmas rather than directly executing the PVS test cases because the semantics of some testing functions are non-executable. That is, in PVS, functions with non-executable semantics cannot be executed using the PVS ground evaluator and PVS built-in strategies. For instance, most functions with set-theoretic semantics in PVS are non-executable, including relational specifications, which are represented as predicates on sets in PVS. For example, the semantics of the function `filter`,

Listing 1.3. A PVS function with non-executable semantics.

```
filter[A:TYPE](p:[A->bool])(s:set[A]):set[A]=
    {x: A | member(x, s) AND p(x)}
```

shown in Listing 1.3, is non-executable. This is because the `filter` function simply describes what kind of elements should be in the result set after the execution of the function but does not specify the steps of how to execute the function in PVS executable syntax. For instance, trying to execute this function directly in PVSio will issue an error message that indicates the `filter` function includes a non ground expression.

2.2 Intermediate Type Classification

To generate tests for OCaml and PVS functions respectively, we have to determine the commonality and difference between the two languages. Therefore, we design an intermediate type system to fill the gap between the type systems of the two languages. Since the types of the two languages cannot be matched with each other one-to-one, we classify the types of the two languages into five different classes and design rules to handle them separately.

OPEV's intermediate type system is categorized into 6 different classes: `PEmpty`, `PBasic`, `PComplex`, `PDef`, `PExt`, and `PSpec`. In this classification, `PEmpty` represents a dummy type that is used as a placeholder to occupy some blank space in the type notation. The existing OCaml types are then grouped according to the remaining five classes. Namely, basic built-in types such as `bool`, `nat`, and `int`; complex data types such as `string`, `tuple`, and `list`; user-defined types including `datatype`, `record`, and others; external library types; and types requiring special treatment such as `functional` types.

For each intermediate type, we design a generating rule and parsing rule according to the class of the type. Currently, OPEV only handles a subset of the OCaml type system. To extend the current OPEV type system into new types, one has to manually add specialized test generating heuristics in OPEV for the new types.

2.3 Test Generation

Types in the `PBasic` and `PComplex` classes have corresponding built-in types in OCaml and PVS. Thus, the test generating rules are simple and straightforward. OPEV generates multiple values for every function argument according to its type and then denotes the values to fit them into OCaml and PVS formats.

For example, for the `int` type, OPEV randomly generates an integer in a predefined range ([-10, 10] by default). The integer follows a uniform distribution, and the predefined range can be modified by the user. For instance, for the range [-5, 5], the corresponding command is as follows:

```
./opev --range -5 5 library_path
```

For types in the `PDef`, `PExt`, and `PSpec` classes, we develop more intricate and complex rules to generate the test cases. For example, OPEV only generates test cases for concrete types. Thus, for an arbitrary type, we define a rule that each arbitrary type must be instantiated to `bool` or `nat`, following the built-in test rules in the Lem source code.

Complex Data Types. For complex data types such as `list` and `string`, we set a length parameter that constrains the maximum length of the type element:

```
./opev --length 16 library_path
```

Since these complex data types have corresponding built-in definitions in OCaml and PVS, we do not need to consider the termination problem for some recursively defined data types because we design specific rules for each of these data types.

For example, if the argument type is `list`, OPEV first randomly generates an integer which is the length of the list, constrained by the predefined maximum length parameter. Then OPEV generates elements for the list, following the rules for the list type. The test value for the list is constructed for OCaml and PVS, respectively, following their list representations. For instance, for a list of length `n`, if the list elements are x_0, x_1, ..., and x_{n-1}, OPEV composes an OCaml list as $[x_0;\ x_1;\ ...;\ x_{n-1}]$ and a PVS list as (: x_0, x_1, ..., x_{n-1} :).

User-Defined Types. In OCaml, developers can apply the `type` keyword to define a new type that represents a `record` or a `datatype`. The newly defined type may have various fields, and each field is denoted with a specific constructor and the corresponding type annotation. OPEV sequentially constructs test-cases for each field of the user-defined type. However, this may cause an infinite loop when there are recursive definitions in the user-defined type; thus, we set a maximum limit of recursive times to prevent infinite construction. Additionally, if the return type is a user-defined type, OPEV requires additional construction rules to directly translate the return results from OCaml to PVS, which means that, if a developer intends to use OPEV to generate tests for a new user-defined type, he/she needs to implement the construction function in the source code of OPEV.

External Types. To automatically generate test cases for the case studies (Section 3), we define generation rules for some external types that are used in these libraries. External types are the OCaml types imported from external libraries, which means we do not know the detailed implementations of the interfaces regarding these types. We have to manually design specific mapping functions from the OPEV intermediate type to OCaml external types and PVS types.

For instance, in our case studies, a typical external type is `Nat_big_num.num`, which is introduced in the library file `nums.cma`. This type is employed to handle the situation where there are large integer operations. However, in PVS, there are no limitations on the range of the default `int` and `nat` types. Thus, in PVS, the test cases can be generated following the rules for `int` and `nat`. On the other hand, in OCaml, we introduce a mapping function named `Nat_big_num.of_int`, which converts an integer into a `Nat_big_num.num` number.

Functional Types. The challenge of constructing a functional argument lies in that the function domain and range are potentially infinite. We initially considered applying the methods in Haskell QuickCheck [8] to generate a functional argument; however, the generated function might have different behaviors in OCaml and PVS because they take random generation seeds. Since we have to generate equivalent functions for OCaml and PVS, we designed a comparatively simple method to generate the functional argument.

First, we define multiple functions in PVS with some specific function patterns. Then OPEV randomly selects a predefined function and applies the function name as the PVS argument. Meanwhile, the OCaml argument is the corresponding function name related to the PVS one.

However, if there are no predefined PVS functions for certain patterns or there are no matching OCaml functions, OPEV constructs a `LAMBDA` expression to take symbolic arguments as the inputs and return a randomly generated constant as the output. This `LAMBDA` expression directly serves as the PVS argument, and a corresponding `fun` expression is built as the OCaml argument.

Dependent Types. The generation tactic for a dependent type is to construct the arguments according to its supertype, complying with the constraints of the dependent type. Right now, the supported constraints include arithmetic and comparison operations. Aside from these types of constraints, OPEV will directly generate test cases according to the supertype.

For example, a dependent type in PVS named `word` is defined as follows. `word` is a subtype of `nat`, and the `word` type is constrained by the constant `N`. OPEV uses the constraint to set up a new range for the natural number and generate a natural number within the range as a `word` type argument.

```
word : TYPE = {i: nat | i < exp2(N)}
```

This test construction strategy does not support more complicated constraints than arithmetic and comparison operations, as those would result in some redundant test lemmas that OPEV would reject. Although such test lemmas do not cause any inconsistency for the OCaml and PVS equivalence, they narrow the test coverage for functions with arguments of these dependent types.

2.4 Proof Automation

For each PVS function, OPEV can automatically generate thousands of test lemmas. It is impractical to manually prove all of them. To automate the proof process, we prove 392 general theorems that support fundamental properties of many translated functions, such as the commutativity and associativity of add operations for bit-vectors with the same length, Listing 1.4.

Listing 1.4. A general PVS theorem.

```
minus_eq_plus_neg: LEMMA FORALL (n:nat, m:nat, bv1:bvec[n],
    bv2:bvec[m]): m = n IMPLIES bv1 - bv2 = bv1 +
    add_vec_range[m]((bv2), 1)
```

Then we implement generic PVS strategies using these general theorems according to the patterns of the functions that are being tested.

For example, in Listing 1.4, the theorem named `minus_eq_plus_neg` proved that the subtraction of two bit-vectors is equivalent to the addition of the first bit-vector and the negation of the second bit-vector. With this theorem, testing regarding bit-vector subtraction operation can be rewritten to addition operation and negation operation.

With the pre-implemented PVS strategies, we then leverage a utility in PVS called Proof-Lite [6] to prove the test lemmas on these functions. The strategies will be able to instantiate these general theorems with concrete numbers as need be in the test lemmas. Moreover, Proof-Lite verifies the test lemmas sequentially. Therefore, we design a `memory management algorithm` to prove the test lemmas concurrently while efficiently utilizing memory. In the memory management algorithm, OPEV calls multiple processes to verify the test lemmas concurrently, monitors the status of the running machine, and automatically adjusts the number of activated processes according to the memory usage of the machine.

Automatic Proof Strategies. To automatically prove large-scale test lemmas with non-executable semantics in PVS, we implement a set of generic PVS strategies. To construct a generic PVS strategy for different functions, we start from a single test lemma and prove it manually. During the manual proof procedure, we extract a simple PVS strategy for this test lemma pattern. Then we try to prove other tests with different patterns using this PVS strategy. If this strategy does not work, we manually prove the new tests and get new PVS strategies. Then we try to combine the PVS strategies for different test patterns together using branching, backtracking, or feature extracting and summarizing. By repeatedly carrying out this process, we synthesize the unified pattern behind the verification of the test lemmas. We then construct a generic PVS strategy using the unified pattern. (It is possible to automate this proof generation, possibly using SMT solvers; we scope that out as future work.)

For instance, in the basic OCaml-to-PVS translation (Section 3.1) library, functions mainly focus on bit-vector operations. The functions in this library involve conversions between natural numbers and their corresponding bit-vector representations. This conversion from natural number to bit-vector in PVS is defined as follows (the source code is in [9]):

```
nat2bv(val: below(exp2(N))): {bv: bvec[N] | bv2nat(bv) = val}
```

The `nat2bv` function is non-executable since it just declares that it is the inverse function of `bv2nat`, which defines the conversion from bit-vector to natural number. Meanwhile, most of the functions in the OPEV_Value library call this `nat2bv` function. Thus, we can exploit the relation between `nat2bv` and `bv2nat` to circumvent the execution of `nat2bv` function, which is non-executable, and to prove test lemmas containing `nat2bv` function.

For example, the `case-split-strat` strategy, as illustrated in Listing 1.5, applies the injectivity and invariance properties of the `nat2bv` and `bv2nat` functions. This PVS strategy can be grandly applied to test lemmas for functions in the OPEV_Value (Section 3.1) library.

Listing 1.5. A generic PVS strategy.

```
(defstep case-split-strat (fname &optional (fnum 1))
  (let ((rewritestr1 (format nil "~a_inj" fname))
       (rewritestr2 (format nil "~a_inv" fname)))
    (branch (case-insert-fname fname fnum)
            ((then (rewrite rewritestr1)(grind)(eval-formula))
             (then (hide 2)(rewrite rewritestr2)(grind)(eval-formula))
             (then (grind)(eval-formula)))))
  "" "")
```

After implementing the generic strategy, we apply Proof-Lite, augmented with our memory management algorithm, and the PVS strategy to prove all the test lemmas generated for the functions in the library. We are able to efficiently prove hundreds of thousands of test lemmas automatically. The statistics are illustrated in Section 3.

3 Case Studies

We now illustrate the application of OPEV on two case studies: a manually implemented OCaml-to-PVS translation and a Sail-to-PVS parser. We detected 11 mismatches during the validation of these case studies. Documentation on these errors is available in [10]. The verification was carried out on an AMD Opteron server (2.3GHz, 64 core, 128GB).

3.1 Manually Implemented OCaml-to-PVS Translation

OPEV validated a manually implemented PVS library for which the source is a single OCaml file in the Sail source code [11], which supplies Sail with definitions and operations of bits and bit-vectors. Since the translation is done manually, the translated PVS library is error-prone. It is desirable to increase the reliability of the translation.

Table 1 illustrates the statistics for this validation. We verified ∼200K test lemmas and found 6 mismatches. An example mismatch: in the implementation of `add_overflow_vec_bit_signed` function in PVS, if the second operand is false, we then assume that there is no overflow and no carry bit for the addition operation. However, in one version of sail_values.ml [11] (commit ce962ff), overflow is set to true. Thus, there is a conflict in the two implementations and the results parsed from the execution of the OCaml function cannot be verified in the PVS test lemmas. OPEV detected this difference in intention as an error.

Table 1. Statistics on validating the OCaml-to-PVS translation.

OCaml Source Code Size	1,488 LOC
PVS Destination Code Size	1,533 LOC
# of Validated Functions	150
# of Manually Proved Generic Lemmas	268
# of Auto-Generated Test Lemmas	215,562
# of Missmatches Found	6

3.2 Sail-to-PVS Parser

The Sail language [12], which is a first-order imperative language, has been used to describe the semantics of ISAs such as x86, ARM, RISC-V, and PowerPC [12]. To facilitate the reasoning on these semantics, we implemented a Sail-to-PVS Parser to expose the semantics of many ISAs and their multitudes of variants – already available in Sail – to the community of PVS users.

The architecture of the parser is shown in Figure 2. First, we rely on the Sail compiler [11] to automatically translate Sail source code to Lem [13], which was designed to serve as a semantic model that was mathematically rigorous [3] and can be translated to OCaml for emulation of testing as well as to Isabelle/HOL, Coq, HOL4, and other languages. Then we employ the Lem compiler to translate the resulting Lem source code into a typed Abstract Syntax Tree (AST). Both the Sail and Lem compilers are in our trusted computing base. (We argue that trusting these two compilers is reasonable due to their small codebase. Besides, they have undergone intensive unit testing in prior work [13].)

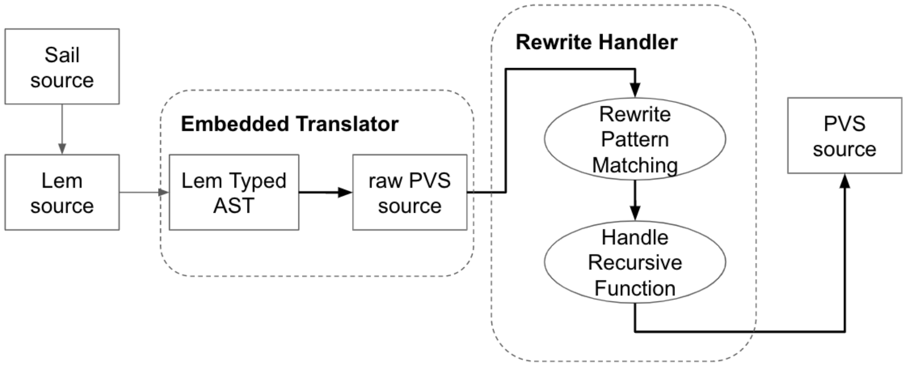

Fig. 2. Architecture of Sail-to-PVS parser.

Our Sail-to-PVS parser takes this typed AST as input and implements two independent parts: an embedded translator and a rewrite handler. The translator is embedded in the Lem source and translates the typed AST into corresponding

PVS code using PVS syntax. The Lem type system does not support dependant types and originally was designed to translate Sail specifications into theorem provers that do not support dependant types, such as HOL4 and Isabelle [13]. In addition to this challenge, at this stage, the generated PVS code is challenging and error-prone due to other differences between PVS and Lem specification languages. For example, the method of reasoning about the termination of recursive functions and various formats of pattern matching for different pattern types. To solve the problems, we apply a rewrite handler, written in Python, to adjust the problematic PVS code. The rewrite handler performs two tasks: rewrite the pattern matching to ensure that the PVS code has consistent types and add `measure` functions for all the recursive functions. The total LOC of the Sail-to-PVS parser, including the embedded translator (1,730 lines of OCaml code) and the rewrite handler (1,033 lines of Python code), is 2,763. However, with these modifications Sail-to-PVS parser is still restricted to pure functions of Sail.

An important use case of the Sail-to-PVS parser is program verification at the assembly level (using PVS). For such a use case, it is critically important that the translation is provably correct. We automatically translate a Lem basic library [13] respectively to PVS and OCaml using the Sail-to-PVS parser and Sail's built-in compiler. Although Sail and Lem are executable, the generated PVS code would call some built-in PVS functions, some of which are non-executable; however, all of them are pure. Since the generated OCaml code is within the scope of OPEV's OCaml subset, it enables us to validate the equivalence between the generated OCaml and PVS code using OPEV. If the equivalence is validated, our trust in that the Sail-to-PVS parser carries out similar functionality as the Sail built-in compiler will increase significantly.

We generated small-scale test cases at the beginning, namely 10 test cases for each function, and attempted to prove all the test lemmas by a default PVS strategy called **grind**. For the test lemmas that cannot be proved, we designed the PVS strategies by proving auxiliary lemmas or by combining multiple strategies together according to the steps described in Section 2.4. Then we generated large-scale test lemmas and verified them using the corresponding strategies.

Table 2. Statistics on validation of Sail-to-PVS parser.

Lem Source Code Size	7,542 LOC
PVS Destination Code Size	10,990 LOC
# of Validated Functions	109
# of Manually Proved Generic Lemmas	124
# of Auto-Generated Test Lemmas	242,685
# of Missmatches Found	5

Table 2 shows the statistics for the library. OPEV determined multiple unprovable test lemmas in the PVS implementation. In turn, we modified the source code of the Sail-to-PVS parser, which generated the test lemmas reported in the table. Due to the gap between the semantics of the Lem and PVS languages, OPEV detected 5 mismatches. Doing this translation validation is practically impossible to achieve manually without OPEV.

4 Past and Related Work

Significant literature exists on translation validation. In [14], the authors show that the seL4 source code and its binary code have the same behavior. The translation validation relies on *refinement proofs*. A refinement proof is possible here due to formal semantics that was created for both the source and target languages. However, the semantics of OCaml and PVS cannot be mapped to each other one-to-one. Besides, refinement proofs, in general, are labor-expensive due to the significant human intervention required. The seL4 refinement proof [1] took 8 person-years; the seL4 total verification effort [1] is more significant and took $\sim$20 person-years.

CompCert [7,15] uses a *formally verified compiler* to establish the correctness of compilation from a subset of C to PowerPC, ARM, RISC-V, or x86 assembly code. The compilation guarantees that the assembly code executes with the behavior that was designated by the original C program [16]. However, the formal proofs of CompCert did not cover the correctness of the formal specifications of C and assembly [15]. In addition, it took six person-years of effort and involved 100,000 lines of Coq code [7].

In contrast with compiler verification and refinement proofs, OPEV is a light-weight approach for the validation of a translation from a high-level language into a theorem prover using `random testing`. OPEV is therefore significantly less labor-expensive. Additionally, OPEV allows non-executable specifications and proofs for generic theorems after translating the code for further verification.

OPEV also differs from some other test-based light-weight verification techniques. For instance, Haskell's QuickCheck mechanism [8] is designed to aid in the verification of properties of a given function. The tests are randomly generated until either a counterexample is discovered in a given domain or a preset threshold is reached. Likewise, AutoTest for Eiffel [17] checks program annotations based on randomly generated test suites. Similar methods exist for theorem provers. Besides, QuickCheck [18] and Nitpick [19] for Coq and Isabelle/HOL uses random testing [20] to support counterexample discovery for a given conjecture. These mechanisms work well with executable specifications. OPEV differs from these efforts by its focus on validating the translation into a theorem prover and the supporting of non-executable semantics. Additionally, our translation into PVS allows the user to *verify* properties and specified conjectures for the translated functions using PVS's built-in test-generator [21] to assist in proving these properties or reaching a counterexample once applicable. But like the other built-in translations, it is restricted to generated PVS's executable specifications from our tool.

The closest work to OPEV is MINERVA [22], which provides a practical and rigorous general approach to produce high assurance software systems using model animation on mirrored implementations for verified algorithms [22]. However, MINERVA is limited to the executable subset of PVS. OPEV can be viewed as complementary to MINERVA when the specification is not executable.

5 Conclusions

In this paper, we presented a validation methodology, called OPEV, that provides a high assurance on the equivalence between OCaml and PVS specifications. OPEV employs an intermediate type system to capture the commonality of the subset of OCaml and PVS and generate test cases for both OCaml and PVS implementations. The reliability of the validation is ensured by executing large-scale stress tests and automatically proving test lemmas using generic PVS strategies. OPEV tool generated more than three hundred thousand test cases and proofs. We demonstrated the OPEV methodology on two case studies, namely a manual OCaml-to-PVS translation and a Sail-to-PVS parser. OPEV significantly increases our trust in the translations.

Currently, OPEV handles a subset of OCaml types and pure functions. In the future, we aim to extend the functionality of OPEV and incorporate more test generation rules for it. We also intend to increase automation in the proof process of OPEV. These enhancements would allow us to translate multiple mainstream instruction sets (ISA) specifications written in Sail into PVS, a necessary step to reason about the binary code of these architectures in PVS [23]. For instance, the methodology of [23] of lifting ARMv8 binaries into PVS7 based on translating ARM specification language ASL [24] into PVS, is interesting to us to generalize for other architectures. It allows the translation of system binary code of ARMv8 into PVS, based on PVS generic theories, theory parameters, and dependent types, in place of monad theory. Therefore, our work would open the door for more future research to verify the binary code of several mainstream instruction sets based on translating Sail ISAs specifications into the prototype verification system PVS.

Artifacts of the OPEV methodology are open-source and publicly available at: https://ssrg-vt.github.io/Renee/.

6 Acknowledgements

This material is based upon work supported by the US Office of Naval Research (ONR) under grant N00014-18-1-2665.

References

1. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: ACM Symposium on Operating Systems Principles. pp. 207–220. ACM (2009)

2. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: International Conference on Interactive Theorem Proving. pp. 99–115. Springer (2012)
3. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: Reusable engineering of real-world semantics. SIGPLAN Not. **49**(9), 175–188 (Aug 2014). https://doi.org/10.1145/2692915.2628143, `http://doi.acm.org/10.1145/2692915.2628143`
4. Conrad, M.: Testing-based translation validation of generated code in the context of IEC 61508. Formal Methods in System Design **35**(3), 389–401 (2009)
5. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer-Verlag, Saratoga, NY (jun 1992), `http://www.csl.sri.com/papers/cade92-pvs/`
6. Munoz, C.: Batch proving and proof scripting in PVS. NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, Report NIA Report (2007-03) (2007)
7. Kästner, D., Barrho, J., Wünsche, U., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S.: Compcert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In: ERTS2 2018-Embedded Real Time Software and Systems (2018)
8. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 268–279. ICFP '00, ACM, New York, NY, USA (2000). https://doi.org/10.1145/351240.351266, `http://doi.acm.org/10.1145/351240.351266`
9. PVS source code. `http://www.csl.sri.com/users/owre/drop/pvs-snapshots/`
10. OPEV bug report. OPEVBugReport
11. Sail project. `https://github.com/rems-project/sail`, last accessed: 2019-05-31
12. Gray, K.E., Sewell, P., Pulte, C., Flur, S., Norton-Wright, R.: The sail instruction-set semantics specification language (2017)
13. Lem project. `https://github.com/rems-project/lem`, last accessed: 2019-05-31
14. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified os kernel. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 471482. PLDI 13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2491956.2462183, `https://doi.org/10.1145/2491956.2462183`
15. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: Compcert-a formally verified optimizing compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress (2016)
16. Kästner, D., Leroy, X., Blazy, S., Schommer, B., Schmidt, M., Ferdinand, C.: Closing the gap–the formally verified optimizing compiler compcert. In: SSS'17: Safety-critical Systems Symposium 2017. pp. 163–180. CreateSpace (2017)
17. Ciupa, I., Pretschner, A., Oriol, M., Leitner, A., Meyer, B.: On the number and nature of faults found by random testing. Softw. Test., Verif. Reliab. **21**(1), 3–28 (2011). https://doi.org/10.1002/stvr.415, `https://doi.org/10.1002/stvr.415`
18. Tanter, É., Tabareau, N.: Gradual certified programming in coq. In: ACM SIGPLAN Notices. vol. 51, pp. 26–40. ACM (2015)
19. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 131–146. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

20. Wada, Y., Kusakabe, S.: Performance evaluation of A testing framework using quickcheck and hadoop. JIP **20**(2), 340–346 (2012). https://doi.org/10.2197/ipsjjip.20.340, `https://doi.org/10.2197/ipsjjip.20.340`
21. Crow, J., Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: Evaluating, testing, and animating pvs specifications (03 2019)
22. Narkawicz, A., Munoz, C.A., Dutle, A.M.: The minerva software development process (2017)
23. Tahat, A., Joshi, S.P., Goswami, P., Ravindran, B.: Scalable translation validation of unverified legacy os code. 2019 Formal Methods in Computer Aided Design (FMCAD) pp. 1–9 (2019)
24. Trustworthy Specifications of Arm v8-A and v8-M system Level Architecture. In: Proceedings of Formal Methods in Computer-Aided Design (FMCAD 2016). pp. 161–168 (October 2016), `https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf`