vanHelsing: A Fast Proof Checker for Debuggable Compiler Verification

Roland Lezuo*, Ioan Dragan*†, Gergö Barany*‡, Andreas Krall*
*Institute of Computer Languages
Vienna University of Technology
{rlezuo,ioan,gergo,andi}@complang.tuwien.ac.at
†Institute e-Austria and
Victor Babeş University of Medicine and Pharmacy
Timişoara, Romania
idragan@ieat.ro
‡CEA, LIST, Software Reliability Laboratory
Gif-sur-Yvette, France
gergo.barany@cea.fr

***Abstract*—In this paper we present vanHelsing, a fully automatic proof checker for a subset of first-order problems tailored to a class of problems that arise in compiler verification. vanHelsing accepts input problems formulated in a subset of the TPTP language and is optimized to efficiently solve expression equivalence problems formulated in first-order logic. Being a practical tool vanHelsing provides also graphical debugging help which makes the visualization of problems and localization of failed proofs much easier. The experimental evaluation showed that this specialized tool performs up to a factor of 3 better than state of the art theorem provers.**

I. Introduction

This section gives a short introduction into compiler correctness and describes the specific translation validation framework vanHelsing has been developed for. Although proofs are the most time consuming part of the framework they are just another component in the whole system. This section gives an overview over the whole translation validation framework and justifies the problem formulation vanHelsing is optimized for.

Compiler correctness deals with the problem whether a source program has been correctly translated into machine code by the compiler. Compilers themselves are a large and complex piece of software and recent research [1], [2] shows that they are far from bug-free. For safety-critical systems those bugs can result in catastrophic events, even if the software itself has been verified (as the bug was introduced by the compiler). The formal verification of compilers is therefore an important research area. One possibility to assure compiler correctness is to have a verified compiler, i.e. a compiler that is proven bug-free and thus correct. The most relevant implementation is Leroy's CompCert compiler [3] which, in large parts, is generated from verified Coq code.

While the CompCert project demonstrates that an industrial grade verified compiler can be created, it fundamentally changes the way a compiler is implemented. Traditional vendors are therefore interested in techniques assuring correctness without the need to rewrite their compiler from scratch in language unfamiliar to the team. Translation validation [4] is an approach allowing traditional compiler construction. It treats the compiler as a black box and only observes the input (source code) and the output (machine code). Similar to program checking [5], the idea is that the correctness criterion is easier to implement than a verified compiler. Translation validation thus splits the code into untrusted parts (i.e. the compiler) and trusted parts (i.e. the verification code).

vanHelsing was designed and implemented as part of a translation validation framework for the back-end of an optimizing C compiler [6]. In this framework each individual pass of the compiler is proven locally correct by applying the translation validation method. Each pass may induce global constraints which are collected by the framework and validated when the full compilation becomes available. In most cases the input and output languages of the single passes are intermediate languages used by the compiler. Only the very first pass operates on the source language and only the very last pass creates machine code. The input to each pass is the output of its predecessor and has thus been validated. In this manner a chain-of-trust [6] is created.

The majority of translation validation work is done when proving local correctness of single passes. In the back-end the intermediate languages are assembly-like languages which operate on registers and are organized in so called basic blocks (sequences of non-branching instructions). Formal semantics of each intermediate-language operation is given by abstract state machine (ASM) [7] models. Uninterpreted functions are used to describe their arithmetic effects. To compute the semantics of a whole basic block symbolic execution is applied to the ASM models. This is feasible as basic blocks are free from loops, and only a small number of intermediate-language operations are data dependent, e.g. conditional branches, predicated execution. Thus the number of paths is low and can be handled for realistic programs.

The symbolic execution engine emits first-order formulas in TPTP format [8] utilizing a technique called direct symbolic execution [6]. These *translation facts* contain the so called *data flow* of a basic block which describes its arithmetic effects. (Side-effects of the block, i.e. memory and control flow changes are handled by the framework, which is beyond

978-1-5090-0461-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SYNASC.2015.34

the scope of this paper.) A *data flow* consists of variables (i.e. registers in our application) and arithmetic functions operating on those variables (i.e. a 3 address machine). Variables which are used in a basic block, without being defined are called *live-in*. Variables which are defined in a basic block are called *live-out*. Variables which only hold intermediate results of an expression's evaluation are called *temporaries*. A medium sized basic block can easily result in more than 10.000 TPTP formulas. An example will be given in section II.

A simulation proof technique is applied to each basic block before and after each single pass [6]. A pass specific mapping correlates *live-in* variables of the block before to *live-in* variables of the block after the translation. *Live-out* variables are mapped as well. A translation is locally correct if mapped *live-out* variables of the block are calculated by equivalent expressions using the same mapped *live-in* variables. It is important to know that the symbolic execution engine is part of the verification and thus of the trusted code. For commercial reasons it is important to keep the amount of trusted code as small as possible because this directly translates into costs for certifications. Thus its reasonable to have a not-ideal problem formulation when it avoids implementing a pre-processor (and thus have additional certification costs).

The intermediate-language models use uninterpreted functions, hence the first-order formulas created by symbolic execution contain them as well. We use uninterpreted functions to model the semantics of machine instructions and compiler intermediate language to match the semantics of the underlying hardware. A microprocessor implementation may e.g. define division-by-zero to result to zero. To be able to model such aspects of the arithmetic units one can not rely on built-in integer arithmetics of theorem provers. Uninterpreted functions also allow to easily model complex instructions like multipy-and-accumulate with saturation semantics. In certain cases using a prover's built-in theories could be a feasiable optimization. The potential of such optimization was not studied in this work.

Summarizing the constraints affecting the problem formulation i) the use of uninterpreted functions to cover implementation aspects of realistic microprocessors and ii) no problem pre-processing, because such code is part of the trusted code and thus expensive in certifications of the product.

The rest of this paper is structured as follows. First we give an overview of what motivated our tool and introduce the notions that are used trough the paper (see Section II). A more formal description of the problem we are addressing with the help of vanHelsing is presented (see Section III). Next implementation details and optimisations used in order to obtain these results are presented (see Section IV). In Section V we preset the debugging capabilities of vanHelsing. We thoroughly evaluated the vanHelsing tool and compared its performance against best off-the-shelf provers (see Section VI).

II. Motivation and preliminaries

Let us start by giving an example. Consider the constant folding pass, which evaluates expressions containing constant values at compile time. In Listing 1 we can observe the structure of problems that are emitted by our research compiler. The output is in TPTP [8] format and encodes three independent data flows. Written as expressions, these are:

$$\begin{aligned} sym_5 &= ((sym_1 + 1) + 1) + sym_4 \\ sym_9 &= (sym_6 + 2) + sym_8 \\ sym_{12} &= sym_{11} f_{\text{unrelated}} sym_{10} \end{aligned}$$

For now ignore sym_{12}, which is unrelated to our problem. For various reasons the output of our compiler contains a large number of formulas which are unrelated to a specific proof. We will later show how vanHelsing debugging mode treats such unrelated formulas. Live-in variables are sym_1, sym_4, sym_6, sym_8, sym_{10} and sym_{11}, while sym_5, sym_9 and sym_{12} are live-out. (sym_2, sym_3 and sym_7 are intermediate values and neither live-in nor live-out).

```
1  fof(id0,hypothesis,add(sym1,1,sym2)).
2  fof(id1,hypothesis,add(sym2,1,sym3)).
3  fof(id2,hypothesis,add(sym3,sym4,sym5)).
4
5  fof(id3,hypothesis,add(sym6,2,sym7)).
6  fof(id4,hypothesis,add(sym7,sym8,sym9)).
7
8  fof(id5,hypothesis,unrelated(sym10, sym11, sym12)).
```

Listing 1: Translation facts with three data flows

Note the relation between the expressions for sym_5 and sym_9. The corresponding translation validation problem could then be formulated as a first-order formula expressing the following question: Assuming $sym_1 = sym_6$ and $sym_4 = sym_8$, does $sym_5 = sym_9$ hold? If it does, then this particular instance of this program transformation is proved correct.

To generate a complete problem which can be handed off to a theorem prover or proof checker two *translation facts* (of the same basic block, before and after a pass) are combined with so called *proof obligations*. *Proof obligations* are additional formulas and the *conjecture*. The additional formulas describe the transformations performed by the pass and the conjecture is the correctness criterion for the pass. In our framework the compiler guides the proof system by providing additional formulas as *witness-information* (which is still untrusted information though). The generation of the *conjecture* (based on the witness-information) is part of the trusted code. A majority of the *proof obligations* are about showing that two *data flows* are equivalent, i.e. they are instances of the expressions equality problem. From a compiler verification point of view it is interesting to have *evidence* that the compilation was indeed performed correctly. This *evidence* should contain a traceable, constructive argument.

As our translation validation framework is designed to be used in an industrial context with medium to large applications, performance is an important issue. The theorem prover is key to achieve a good performance of the whole translation validation framework. In order to choose the best performing first-order theorem provers we turned to the CASC competition [9]. And from there we picked the best performing ones, Vampire [10], [11] and E [12]. Both provers are using resolution and superposition [13] calculi to perform the proof. More details can be found in the Handbook of Automated Reasoning [14].

For our verification application, based on the sketched problem formulation, the most likely outcome is thus that a refutation will be found. This means the compilation has been performed correctly and the refutation is the proof trace (*evidence* for our application). There is an annoying uncertainty,

because a refutation merely means that a contradiction has been derived. If the derived contradiction is unrelated to the conjecture the proof proves nothing for our application and we call it a *spurious* proof. As the compiler is untrusted it may easily emit *translation facts* or *witness-information* containing a logical contradiction. *Spurious* proofs are thus a real issue for a translation validation application. It is of course possible to execute the theorem prover on the *translation facts* without a conjecture to validate that it is free of contradictions. The major drawback of this approach is that it roughly doubles the verification time.

On the other hand, if no refutation can be found, the result is either that the problem is satisfiable, or a timeout occurs. For our application this means that the compilation went wrong, i.e. it can't be proven correct. The main issue with superposition based provers is that there is no indication of the cause of the error. For acceptance in an industrial context the tool must provide good error reporting facilities and help to identify the problem with the proof. Manually finding the problems in proofs consisting of thousands of formulas is a tedious and time-consuming task. It needs quite some experience and knowledge with theorem provers to pinpoint the problem and translate it back into the original problem domain, i.e. to answer the question what went wrong in the compilation. vanHelsing offers graphical debugging aid to investigate the cause of failing proofs. In our experience this allows compiler domain experts to quickly identify the issues in the original problem domain with a deep understanding of the tool. More detail will be presented in section V.

Satisfiability modulo theories (SMT) [15] is a major branch in automated theorem proving that could theoretically help with this issue. Using the same problem formulation SMT produces a model for erroneous translation (they are satisfiable) which helps in identifying the bug in the compiler. For more complex problems finding a model seems to be a very time consuming task. In our experiments we noticed a big degradation in performance of the Z3 solver, more details about this are presented in Section VI. On the other hand no information is given when the problem is found to be unsatisfiable. For our application this means that no *evidence* is created.

After studying these problems, the observation we made is that many proofs in our problem domain have a very similar tree-like structure. As they are derived from the data-flow of the compiled programs we call them data-flow equivalence problems (DFE) [6]. The motivation to develop the vanHelsing checker was to exploit the special structure of these problems in order to i) improve the performance and ii) have a tool that reliably creates *evidence* and iii) provide users the possibility to debug why a proof failed.

III. The Problem Class

In the following we are going to briefly present the supported input language, features and restrictions imposed by vanHelsing. As input language the vanHelsing checker uses a subset of TPTP v6.0 [8].

As top-level elements Typed First-Order Formulas (TFF), containing type information for predicates and variables and First-Order Formulas (FOF) are accepted. Due to the way vanHelsing is designed, the TFF formulas are accepted but it does not keep track of the type specified in the formula but rather assumes integer types for all values. Nonetheless correct type information should be added to achieve compatibility with other provers (i.e. Vampire). All TPTP formulas have the generic form `language(id, role, formula).` with language being `fof` or `tff`. The role of a FOF formula is one of *axiom*, *hypothesis* or *conjecture* (formula to be proven). The subset of accepted FOF formulas is tailored to model data-flow equivalence (DFE) problems. We first list the supported subset of TPTP and establish some terminology before defining the DFE problem itself (section III-B).

A. Input Language

- *Values / Variables* – `$true, 1, -4, sym2`
  The values $true and $false encode the boolean constants *true* and *false*. Integer constants represent the corresponding integer value. Boolean and Integer constants are called *well-defined* values (that is: their semantic value is known). All other values (e.g. *sym2* are supposed to be (unknown) integer values. Variables starting with an upper case letter are universally quantified free variables used in patterns.
- *Functor Application* – `pred(x,y,z)`
  In the context of vanHelsing, all functions must be treated as predicates. Although the formalization exclusively uses predicates the problem domain exclusively uses functions. In order to address these issues, by convention we map a n-ary function f of the problem domain to a $n+1$-ary predicate f in the DFE. The addition $z = add(x, y)$ in the problem domain would be mapped to the predicate $add(x, y, z)$. If a functor application is also part of the conjecture, the corresponding fact must eventually be derived for the proof to succeed.
- *Equality* – `x = y`
  Assuming *x* and *y* are both *well-defined* but have different values this implies that the problem contains a contradiction. If equality is used in the conjecture the values *x* and *y* must eventually be unified for the proof to succeed.
- *Inequality* – `x != y`
  In case *x* and *y* are both *well-defined* but have the same value it implies that the problem contains a contradiction. If inequality is used in the conjecture the values *x* and *y* must not be unified for the proof to succeed.
- *Implication* – `lhs => rhs`
  If *lhs* (the pattern) evaluates to true *rhs* (the action) will be performed. An action may either be a function application, in that case a new fact will be added to the proof or an equality, which triggers an unification. No unbound free variables must occur in the action. An example usage is the implication `(add(A,B,X) & add(A,B,Y)) => X=Y)`. In the context of our work, implications drive the unification used to solve the DFE.
- *Conjunction* - `formula1 & formula2`
  Informally introduced in above example. Let us define

conjunction to be similar to the notion used in first-order logic, as a remark we note that conjunctions can be also used in the context of terms. Important applications of the conjunction is a conjecture consisting of multiple clauses and of course in complex patterns of implications.

- *Equivalence* – `lhs <=> rhs`
  The equivalence pattern will be translated into two implications (`lhs => rhs` and `rhs => lhs`).

B. Data-flow Equivalence Problems

The vanHelsing checker is designed and optimized to solve a specific problem class very efficiently. In this section we define the data-flow equivalence problems and give an example.

Given a set of functions $F = \{f_i : i \in 0 \dots n\}$ (each with a fixed arity) and a set of variables $V = \{v_j : j \in 0 \dots m\}$ a **data-flow (DF)** is a set of function applications $v_j = f_i(a_0, \dots) : v_j \in V, f_i \in F, a_0, \dots \in V$. A variable v_j which is the result of applying a function f_i ($v_j = f_i(a_0, \dots)$) is called to be *defined* by this function application. A variable a_j appearing as an argument in a function application is called to be *used* by this function application. The DF is free of cycles meaning that a variable defined by a function application is never used in that function or any other function defining the arguments (recursive).

There are two distinct sets of variables in a DF. The set of all variables which are only defined but not used is called the *live-out* set, the set of variables only used but never defined is called the *live-in* set.

Given a defined variable v_j its data-flow tree (expression) can be constructed by recursively replacing all variables not in the live-in set with their defining function applications.

A DFE consists of two DF (DF_0 and DF_1) and two mappings M_I and M_F. M_I is a bijective function associating each variable v_i^0 of the live-in set of DF_0 with a variable v_i^1 of live-in set of DF_1. M_F also is a bijective function associating the variables in the live-out sets.

The *syntactic* DFE problem can now be formulated as follows. Let DF_0 and DF_1 be data-flows, M_I a live-in mapping and M_F a live-out mapping. Is the data-flow tree of each live-out variable v_j^0 of DF_0 equal to the data-flow tree of $M_F(v_j^0)$ (respecting the equality of live-in variables defined by M_I)?

The *semantic* DFE does not ask for syntactic equality of the data-flow trees but semantic equality. A set of semantic equivalent transformations must be given then. In this paper we implicitly mean semantic DFE problems unless stated otherwise. The key observation here is that the conjecture of a DFE is a conjunction of equalities. To prove those equalities a prover must not derive any new clauses, unification is sufficient.

We encode semantic DFE in first-order logic. All n-ary function applications are mapped to $n + 1$-ary predicates (with the function result being the last argument). Variables of the DFE are mapped to (TPTP) values. The (syntactic and semantic) equivalence of data-flow trees needs to be encoded by axioms. Listing 2 shows an example. The two data-flows are

```
fof(id0,hypothesis,add(sym1,1,sym2)).
fof(id1,hypothesis,add(sym2,1,sym3)).
fof(id2,hypothesis,add(sym3,sym4,sym5)).

fof(id3,hypothesis,add(sym6,2,sym7)).
fof(id4,hypothesis,add(sym7,sym8,sym9)).

fof(id5,hypothesis,unrelated(sym10, sym11, sym12)).

fof(ax1,axiom,(add(A,B,X) & add(A,B,Y)) => X=Y).
```

Listing 2: A TPTP file containing three data-flow trees

formed by the predicates *id0, id1, id2* and *id3, id4*. (Actually there is a third data-flow formed by predicate *id5*). Axiom *ax1* encodes syntactic equivalence. The mappings I and F are missing in this listing and will be added later.

In order to better visualize how the input problem looks, vanHelsing offers an option to print the internal representation of the problem, the proof graph, using GraphViz [16]. Figure 1 shows the initial graph built from the example given in Listing 2. Function applications are printed as structured rectangular boxes. The first field contains the predicate's name while the following fields contain the predicate arguments. And we use, as a convention, the last argument to represent the result. All arguments are linked to the referenced value nodes and are printed in ellipses. In case of the unified values, values that are equal, are printed as a list after the equal sign, in our example there are none.

IV. Implementation

vanHelsing is a command line tool for POSIX systems written in C++. It is used as part of our translation validation tools and we consider the implementation as stable and mature. During the development of the validation tools vanHelsing's implementation has been thoroughly tested by executing Vampire in parallel and assure that both tools produce the same results. As input language it accepts a subset of the FOF language specified in TPTP (including all-quantified variables, negation, conjunction, implication and equality).

Internally the problem is represented as a graph with two basic node types. Value nodes represent constants, variables and symbols while functor nodes represent basic boolean operators and user defined predicates. Implications of the form $P(X) \wedge P(Y) \implies X = Y$ drive the *unification* engine. The antecedent is treated as a *pattern* and matched against the graph. If a match is found the equality described by the consequent is used to rewrite the graph, i.e. to unify value nodes. The value node with the smaller degree is removed from the graph and edges are inserted from all its adjacent nodes to the unified node.

This matching is repeated in a round-wise manner until a fixed point is found. Thus vanHelsing implements a forward-chaining strategy.

Whether two data flows (expressions) are *semantically* equivalent is reduced to the question whether the value nodes representing their result have been unified or not. In case they have been unified vanHelsing can create evidence file summarizing all performed unifications together with the matching patterns. vanHelsing therefore never finds *spurious* proofs and always provides evidence.

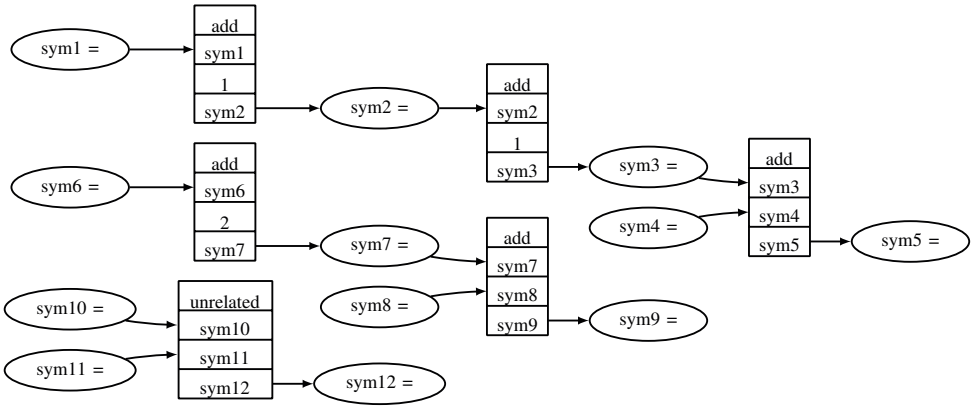

Fig. 1: Initial Data Flow Trees

In order to further improve performance of vanHelsing we have also implemented a number of optimisations inside vanHelsing. These optimizations in conjunction with the way we represent the problem proves to perform best in the context of compiler back-end verification.

1) Dead patterns: As a first optimization it is ensured that vanHelsing does not try to match a function application pattern if there are no terms it could match, e.g. in cases where only `add` functions are used no matching needs to be performed for a function named `sub` (subtraction). This optimization becomes very effective because conjunction patterns inherit this property from their clauses. This enables the use of a generic library of axioms which describe all optimizations performed by the compiler and reuse of that library for other verification projects, because axioms not used by a specific pass don't impact the execution time of the prover.

2) Term Indexing: The unification process is driven by implications. Many of the axioms describe the syntactic equivalence of the data-flow trees. They all have the generic form:

$$pred(A, B, X) \wedge pred(A, B, Y) \implies X = Y$$

The first predicate is matched and concrete values are assigned to its free variables *A* and *B*. Matching the second predicate can now be accelerated if *A* or *B* are (already) *well-defined*, i.e. their values are constants. vanHelsing stores all functors of a specific type in a hash-map (for fast look-up) and maintains hash-maps for functors with *well-defined* arguments. In the current implementation the first three arguments are considered. Term indexing proved to be a key feature and is considered the single most important optimization implemented in vanHelsing.

3) Functor freezing: We call a sort of functors frozen if no functor of their name has been modified in the current round. A pattern is called frozen if it matches a functor which is frozen itself. The conjecture pattern inherits its frozen status from its clauses. Initially there are no frozen functors, assuring that each pattern is matched at least once. Frozen patterns may be skipped during the pattern matching phase as they can not produce any new unifications. A sort of functors must be unfrozen when a new fact involving this sort is added to the proof (i.e. a new fact was stated in a consequent).

V. Debugging a failing proof

The final problem graph is also a very good indicator of the reason for the failed proof (i.e. the compiler bug). By exploiting the regular structure of data flow equivalence problems it is possible to identify the *relevant* parts of the graph. vanHelsing computes the reachability (from pairs of value nodes which were not unified, but were conjectured to) under the assumption that all predicates are encoding functions. The direction of the edges are therefore incoming for all arguments but the last one (the result).

```
1 fof(ax1,axiom,(add(A,B,X) &
2                add(A,B,Y)) => X=Y).
3 fof(op1,hypothesis, sym1=sym6).
4 fof(op2,hypothesis, sym4=sym8).
5
6 fof(cj1,conjecture, sym5=sym9).
```

Listing 3: Missing an axiom

Listing 3 shows an example for a failing proof. Lines 3 and 4 encode (application provided) knowledge about initially equal live-in variables of two data flows (from Listing 1). The conjecture (in line 6) states that the given symbols must be computed by *semantically* equivalent data flows. Without knowing the arithmetic identity $(A+1)+1+C = A+2+C$, the prover is unable to unify sym_3 and sym_7 and thus the proof fails. Each conjectured equivalence which was not unified is dumped into a separate graph (all irrelevant nodes have been removed). These small, isolated graphs can be manually inspected to identify the reason for the failing proof. Value nodes have one of three possible colors in the failure dumps: red and yellow nodes are only part of one data flow, while orange nodes are part of both (i.e. were unified). The reason for the failure is often near the first value nodes which were not unified (i.e. red or yellow ones). Figure 2 shows the resulting graph (irrelevant nodes have been removed, i.e. the *unrelated* functor and its values are not printed).

Visual inspection quickly reveals that the first different colors begin to appear with sym_2 and sym_7. It is clear that sym_7, representing the expression $A + 2$, needs to be unified with the expression $A + 1 + 1$, represented by sym_3. A developer can thus quite easily conclude the cause of the problem. In this case the proof system is not aware of this

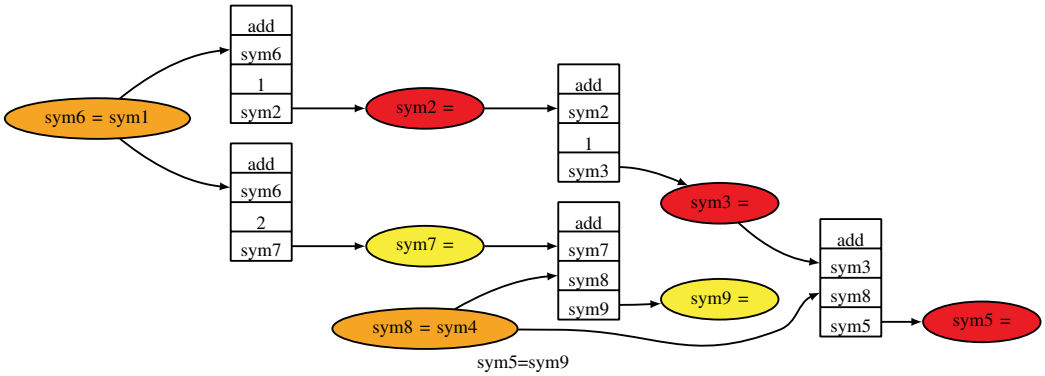

Fig. 2: A failing proof

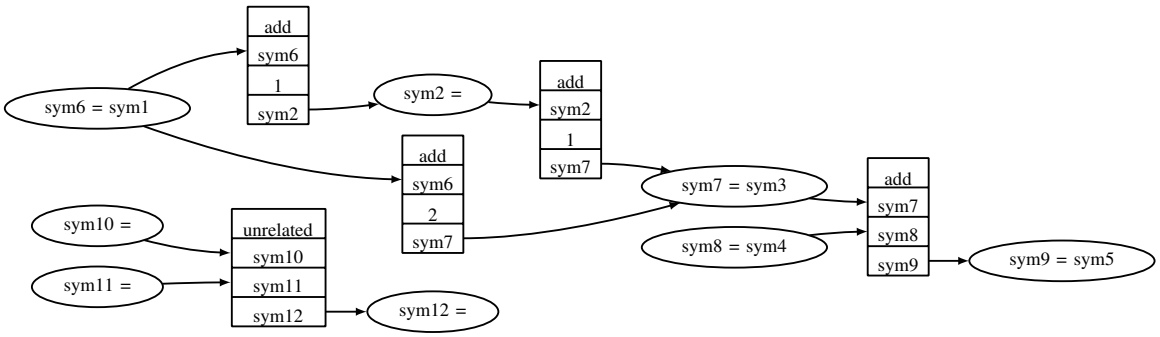

Fig. 3: A succeeding proof

optimization. Adding the needed axiom (Listing 4) results in a succeeding proof depicted in Figure 3.

```
fof(ax2,axiom,(add(A,1,B) & add(B,1,C)
                & add(A,2,D)) => C=D).
```

Listing 4: The missing axiom

VI. Performance Evaluation

We have compiled 5 sets of benchmarks from three different back-end passes of our compiler. The problems within all sets have a common structure, but the structures are different between the sets. Instruction selection (isel) problems are the most complex, because the transformation has the largest impact on the data flow. Register allocation (regalloc) problems are of modest complexity, depending on the amount of spill code inserted. Without spilling the data flow does not change at all, but if registers were spilled the changes are intrusive. VLIW scheduling (vliw) problems are the simplest: Instructions are reordered, the data flow will not be changed at all. Normally the problems emitted by our compiler can be proven, i.e. *isel.succ*, *regalloc.succ* and *vliw.succ*. During development we also collected a set of problems which can not be proven (compiler bugs, missing axioms), i.e. *isel.fail* and *vliw.fail*. Interestingly we noticed that Z3 does not scale well with respect to performance on the failing problem sets.

The problems emitted by our compiler are directly used by Vampire and vanHelsing. Because E-prover does not support types, we have to apply a preprocessing step and remove them. In order to also experiment with Z3, we had to first convert the problems into SMTlib format. For doing so we used the *tptp2x* program that is part of TPTP library and allows us to convert problems into *smt* format which can be used by Z3. The current formulation of the problems proves to be not optimal for Vampire nor for Z3. Vampire would profit from using the built-in equality instead of using axioms. Z3 would profit from its built-in arithmetics instead of the axiomatization provided by us. But as mentioned in the introduction the problem formulation can not be chosen freely. It is given by the design and constraints of the symbolic execution engine for ASM models and by the design goal to keep the trusted code as small and simple as possible.

Table I contains the number of problems in the set, average file size (mean) as an indicator of the complexity of each problem, total size of the set and time each prover needs to process the whole problem set. We report on the best of 3 runs of vanHelsing (version aa115e4), Vampire (1.8 rev. 1362), E-prover (E 1.8-001 Gopaldhara) and Z3 (4.3.1). The tests were performed on a Core i7 @ 1.73 GHz using a 64 bit Ubuntu 12.10. We decided to run all the solvers three times in order to eliminate any effect from the operating system that might influence the results while the experiments are run.

Vampire is a very fast prover and has won the FOF section of the CASC [9] competition for many years now. Although that is the case, for our problem formulation vanHelsing always performs better then Vampire (roughly factor 2). Vampire proves to always be second fastest prover tested for these

TABLE I: Benchmark Set and Performance

Benchmark Set	Size # files	Size mean	Size total	Runtime (seconds) vanHelsing	Vampire	E	Z3
isel.succ	1705	25 kiB	49 MiB	**13.76**	24.54	44.31	42.41
regalloc.succ	454	412 kiB	239 MiB	**49.11**	54.79	491.98	55.34
vliw.succ	401	484 kiB	259 MiB	**54.55**	209.13	816.41	233.74
vliw.fail	27	905 kiB	22 MiB	**4.38**	17.54	88.72	81.21
isel.fail	343	29 kiB	12 MiB	**2.97**	7.45	38.82	961.81

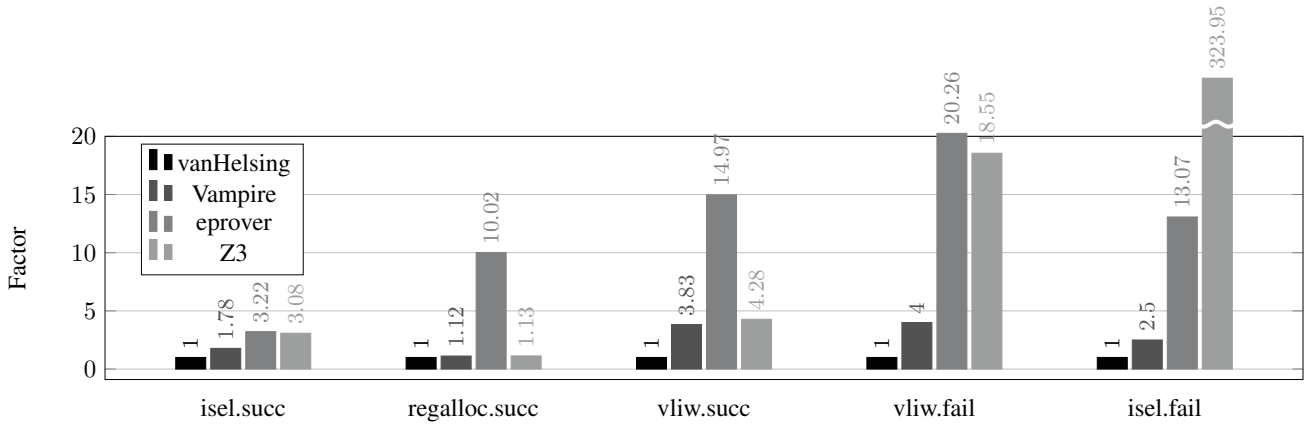

Fig. 4: Relative performance (factor), smaller is better

problem sets. E-prover has been executed in silent mode and failed to prove one problem of the *isel.succ* set. In the case of E-prover we have noticed that the performance is generally worse than vanHelsing and Vampire's performance. In general Z3's performance almost matches Vampire, with one major exception, the class of satisfiable problems. For this evaluation hard timeouts (*-T:3 -t:3*) of 3 seconds (vanHelsing needs less than 3 seconds for all 343 problems in the set) were neccesary. We noticed that using this small timeout Z3 fails to find a solution on all problems of the *vliw.fail* set and only found the solution for 28 problems of *isel.fail*. By increasing the timeout to 240 second, Z3 finds 309 models for the *isel.fail* set, but still no model is generated for any of the problems in *vliw.fail*. Figure 4 shows the relative performance for all provers on each of the problem sets, normalized to vanHelsing's total runtime. Despite the good performance in our application we do not expect vanHelsing to be competitive on general first-order problems.

VII. Conclusions

In this paper we presented vanHelsing, a tool that is tailored toward the problem of proving expressions to be semantically equivalent. These kind of problems frequently, but not exclusively, occur in compiler verification. An outstanding feature of vanHelsing is its ability to produce graphical representations of the problem in GraphViz format. If the proof for a problem can't be found, the *relevant* sub graph can be displayed. vanHelsing uses colors to highlight the problematic parts of failing equivalence proofs. In combination this reduces the time spent to isolate to cause for failing proofs from hours to minutes.

Beside its debugging capabilities, vanHelsing's performance on the problems that occur in practical compiler verification is much better than state-of-the-art theorem provers (up to a factor of 3).

References

[1] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532

[2] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 216–226. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594334

[3] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, pp. 107–115, 2009.

[4] A. Pnueli, M. Siegel, and F. Singerman, "Translation validation," in *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '98. Springer, 1998, pp. 151–166.

[5] M. Blum, S. Kannan, C. Sci, and C. Sci, "Designing programs that check their work," 1989.

[6] R. Lezuo, "Scalable Translation Validation," Ph.D. dissertation, Vienna University of Technology, 2014.

[7] Y. Gurevich, *Evolving algebras 1993: Lipari guide*. New York, NY, USA: Oxford University Press, Inc., 1995, pp. 9–36.

[8] G. Sutcliffe, "The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 337–362, 2009.

[9] G. Sutcliffe and C. Suttner, "The State of CASC," *AI Communications*, vol. 19, no. 1, pp. 35–48, 2006.

[10] A. Riazanov and A. Voronkov, "The design and implementation of VAMPIRE," *AI Commun.*, vol. 15, pp. 91–110, Aug. 2002.

[11] L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire," in *CAV*, 2013, pp. 1–35.

[12] S. Schulz, "E - a brainiac theorem prover," *AI Commun.*, vol. 15, no. 2,3, pp. 111–126, Aug. 2002.

[13] L. Bachmair and H. Ganzinger, "Resolution theorem proving," in *Handbook of Automated Reasoning*, 2001, pp. 19–99.

[14] J. A. Robinson and A. Voronkov, Eds., *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[15] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[16] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *SOFTWARE - PRACTICE AND EXPERIENCE*, vol. 30, no. 11, pp. 1203–1233, 2000.