vanHelsing: A Fast Theorem Prover for Debuggable Compiler Verification

Roland Lezuo*, Ioan Dragan†, Gerg Barany*, Andreas Krall*
*Institute of Computer Languages
Vienna University of Technology
{rlezuo,gergo,andi}@complang.tuwien.ac.at
†Institute e-Austria
Timisoara, Romania
idragan@ieat.ro

***Abstract*—In this paper we present vanHelsing, a fully automatic first-order theorem prover aimed especially at the class of problems that arise in compiler verification. vanHelsing accepts input problems formulated in a subset of the TPTP language and is specially tailored to efficiently solve expression equivalence problems formulated in first-order logic. Besides solving these problems, vanHelsing provides also graphical debugging help which makes the visualization of problems and localization of failed proofs much easier. From our experiments we noticed that using this specialized tool one can gain up to factor 3 in performance when compared to the non-specific theorem provers.**

I. Introduction

Correct compilers are essential tools for the development of safety-critical systems. However, real-world compilers can contain hundreds of bugs that lead to the generation of wrong code [1], [2]. The formal verification of compilers is therefore an important research area.

Typical problems that arise in compiler verification can be effectively formulated in first-order logic. The problem of automated or interactive theorem proving in first-order logic is a well studied field. For solving problems in first-order logic several mature automatic theorem provers are available. However, we have found that these are surprisingly inefficient for solving compiler verification problems.

Additionally, off-the-shelf provers provide no feedback describing the reason why a particular proof attempt failed. This means that a failing proof attempt only shows that there is some bug in the compiler or in the problem formulation, but there is no indication how to identify the underlying issue. This paper describes vanHelsing, a theorem prover that is particularly efficient at solving compiler verification problems. The prover includes a debug mode which minimizes failing inputs to the bare essentials and makes it easy to recognize the reason for the failing proof.

We have implemented translation validation [3] tools for various back-end passes of our research compiler. The (untrusted) compiler emits *translation facts* and our (trusted) tools generate *proof obligations*. *Translation facts* are encoded as first-order predicates describing the semantics of the program before and after a compiler pass. The *proof obligations* are properties that must hold (depending on program and compiler pass) so that the translation is correct. *Proof obligations* are formulated as *conjectures*. A library of axioms is provided for each compiler pass defining which transformations are allowed. Combination of the *translation facts*, the axioms and the *proof obligations* result in first-order problems which are handed off to a theorem prover.

A majority of problems in the context of translation validation are proving equivalence of expressions [4]. A compiler commonly splits programs into basic blocks (instruction sequences without branches). Basic blocks have live-in and live-out values and apply arithmetic operations to those values. A correct translation of a program preserves the *semantics* of those expressions (and the control flow and memory side-effects, but that is beyond the scope of this paper). We call them *data flows* of a basic block.

In the first attempt we decided to use for the verification purpose the best performing theorem provers available. In order to choose the best performing first-order theorem provers we turned to the CASC competition [5]. And from there we picked the best performing ones, Vampire [6], [7] and E [8]. Because both provers are based on various resolution and superposition calculi, for more details see [9], we encountered the following problem. The expected outcome of each problem is that the prover finds a refutation. In this case the compilation was performed correctly and the refutation is an *evidence*. In case the prover does not find a refutation proof for a problem two results are possible. Either that the problem is satisfiable, in this case we get no further information from the prover, or the prover times out. Such results are not useful for our purposes as they provide no clue for the cause of the problem (i.e. the erroneous compilation).

Another problem that arises in using an off-the-shelf prover is that even in case refutation is found, there is still no guarantee that the found contradiction proves the intended goal. This can happen if a contradiction can be found directly in the axioms or clauses that are passed to the prover, in this case *spurious* proof of refutation is found.

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) [4]. The motivation to develop the vanHelsing prover was to exploit the special structure of this proofs to i) improve the performance and ii) overcome the described

shortcomings.

The remainder of this paper is structured as follows. At 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 first start by giving an example. In Listing 1 we can observe the structure of problems that are emitted by our research compiler. The output is in TPTP [10] format and encodes three independent data flows. Written as expressions, these are:

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

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 similarity between the expressions for sym_5 and sym_9. If the symbols in corresponding operand positions stand for equivalent expressions, a compiler's constant folding pass may have converted the first of these expressions into the other. 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.

With superposition [11] based provers like Vampire [6], [7] and E-prover [8] there are two problems that arise in our application domain: i) if the *translation facts* produced by the (untrusted) compiler contain a contradiction a *spurious* refutation will be found and ii) if the translation was erroneous the prover states *satisfiability*, but gives no hint about the nature of the problem. In our experience, the first problem can occur in the early stages of implementing translation validation in an existing compiler. Bugs can occur in the generation of translation facts but masquerade as wrong-code bugs. This problem can be solved by running the prover on the *translation facts* and axiom set without the conjecture; they must be satisfiable.

For the second problem, we are not aware of a good previously proposed solution. For repairing such problems a domain expert has to manually check the proof to locate the reason why the proof failed (and locate the compiler bug). This is a tedious and time-consuming task. Ideally, it should be guided by a minimal model (an example of wrong code) produced by the prover that suggests to the compiler writer where to look for the bug.

Satisfiability modulo theories (SMT) [12] 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. From our experiments we noticed a big degradation in performance of the Z3 solver, more details about this are presented in Section VI. Another issue is the fact that in case the SMT solver finds the problem to be unsatisfiable then no further information about reason of unsatisfiability is given.

In the case of superposition based theorem provers *evidence* is created. The problem can be formulated in a different fashion so a model is found as *evidence*, losing the models for erroneous translations, though. A missing evidence weakens the strength of the argument that the compilation is proven to be correct (proof can not be retraced independently).

In our application, generated problems are commonly larger than 10.000 lines of TPTP. Therefore our tool should also provide a debug mode to analyze failing proofs. A representation of the encoded data-flow represented as a graph helps to catch the interrelation of predicates and values, but also creates the connection to the problem domain. vanHelsing can print a graph representation of the problem utilizing the widely used DOT language [13]. More details about how vanHelsing can be used in order to find failing proofs can be found in Section V.

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 prover uses a subset of TPTP v6.0 [10].

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 be 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 prover 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.

```
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

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 by 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. In the case of vanHelsing prover we decided that it is interesting to also handle conjunction of functor patterns.

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 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.

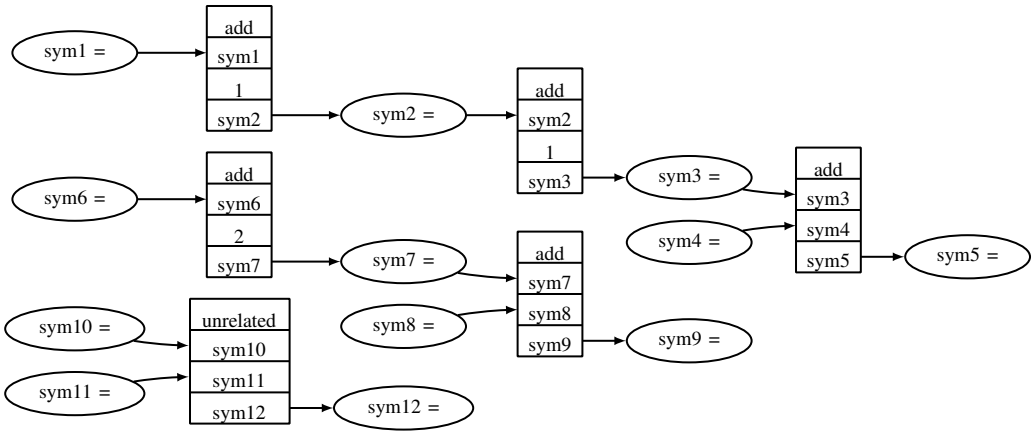

Fig. 1: Initial Data Flow Trees

In order to better visualize how the input problem looks like, we have added to vanHelsing an option that prints the internal representation of the problem, proof graph, using DOT language [13]. 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 we have verified vanHelsing's soundness by using Vampire. We did this by running both tools on the same set of problems and both tools provided the same outcome. 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 an evidence file summarizing all performed unifications together with the matching patterns. vanHelsing therefore never finds *spurious* proofs and always provides an evidence.

In order to further improve performance of vanHelsing we have also implemented a number of 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 first optimization we make sure that vanHelsing does not try to match a function application pattern if there are no terms it could match, e.g. in case of `add(A,2,B)` no matching should be done for `add` predicate name. This optimization becomes effective if conjecture patterns inherit this property from their clauses. A generic set of axioms can then be used for each proof as they don't impact the execution time.

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

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

We match the first predicate and get concrete values for the free variables *A* and *B*. Matching the second predicate can now be accelerated if *A* or *B* are well-defined, meaning that their values are known. 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 we consider the first three arguments. Term indexing proves to be a key feature and can be considered as the most important among optimisations added to 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 pattern may be skipped during the pattern matching phase as they can not

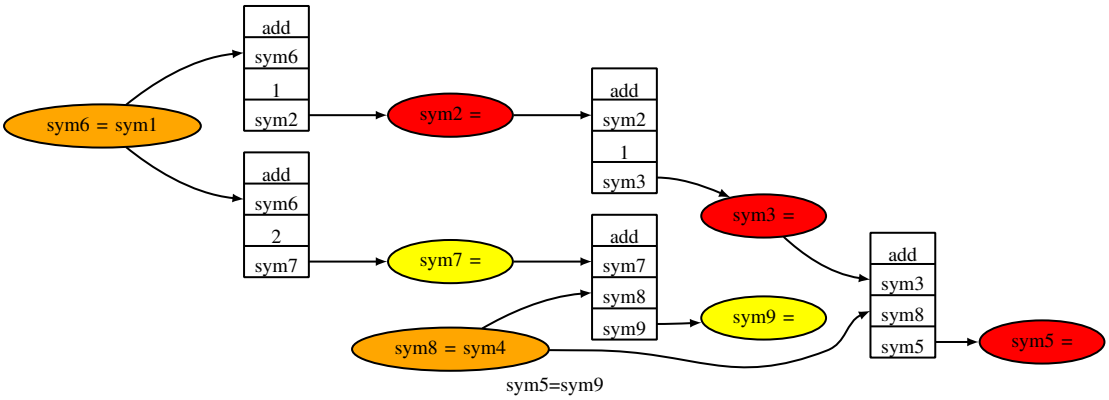

Fig. 2: A failing proof

produce any new unifications. A functor sort must be unfrozen when a new fact involving this sort is added to the proof.

V. Debugging a failing proof

The final problem graph is also a very good indicator as to why the proof failed (i.e. locate the compiler bug). By exploiting the regular structure of the 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) assuming all predicates encode functions. The direction of the edges are therefore incoming for all but the one with the highest index at each functor node.

```
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 often is 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. This is an instance of a more general fact of arithmetic. Adding the needed axiom (Listing 4) results in a succeeding proof depicted in Figure 3. Such axioms are produced by the compiler as part of their translation facts, documenting each transformation that was applied to the program.

```
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 the problems into *smt* format 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. While Z3 would profit from its built-in arithmetics instead of the axiomatization provided by us. But since the problems are automatically created by the verification engine of the research compiler it is unreasonable to freely change the way problems

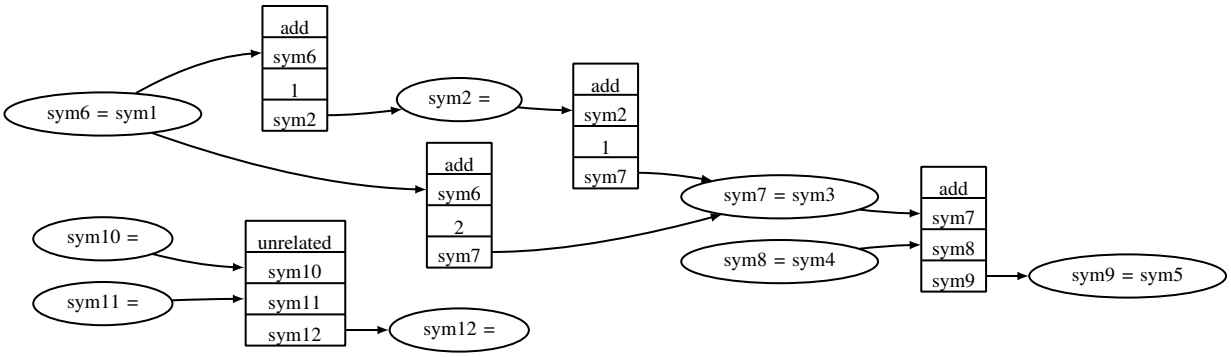

Fig. 3: A succeeding proof

TABLE I: Benchmark Set and Performance

Benchmark Set	# 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

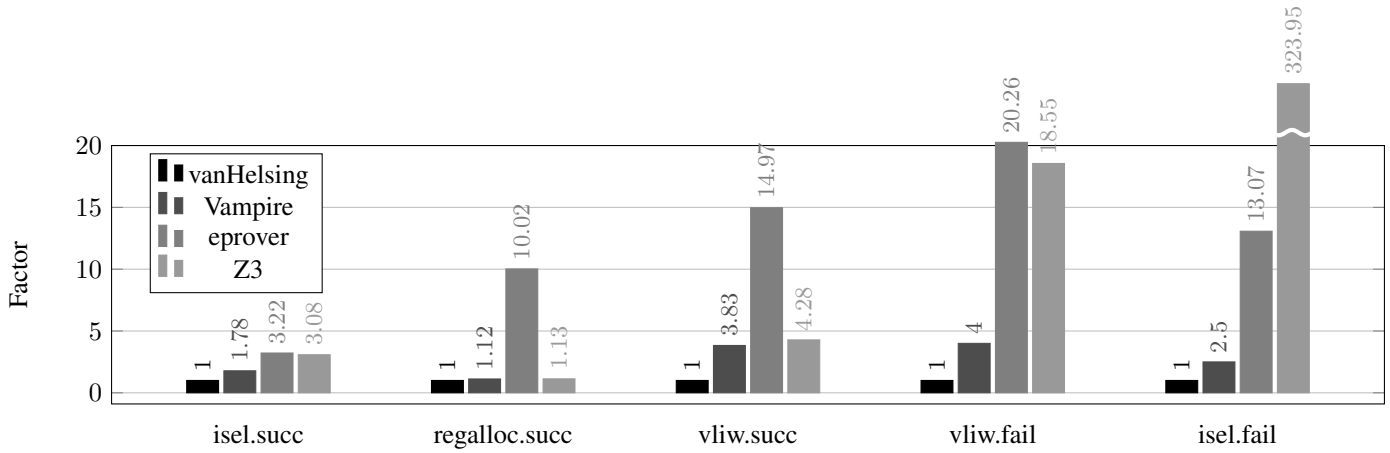

Fig. 4: Relative performance (factor), smaller is better

are formulated, although probably this would highly influence performance of Vampire, E and Z3.

Table I contains the number of problems in the set, average file size (mean) as an indicator of the complexity of each problem, the total size of the set and the 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 [5] 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 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. Using Z3, proves in general to almost match the performance of Vampire, except for the case of *sat* problems. For evaluating the performance used hard timeouts (*-T:3 -t:3*) of 3 seconds (vanHelsing needs less than 3 seconds for all 343 problems in the set). 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 the vanHelsing 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 DOT format. If the proof for a problem can not be found the *relevant* sub graph can be displayed. In order to better highlight the problematic parts of failing equivalence proofs, vanHelsing uses different colors.

Using this approach in order to analyze failing proofs reduces the time spent from hours to minutes. Beside its debugging capabilities, vanHelsing's performance on the kind of 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] 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.

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

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

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

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

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

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

[10] 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.

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

[12] 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

[13] 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.

Availability and Reproducibility[1]

An evaluation package containing binary 64 bit Linux versions is available at http://www.complang.tuwien.ac.at/tbfg/vanHelsing-evaluation.tar.gz. It also contains the benchmark presented in the evaluation section of this paper. Time was measured using the *time* command, we reported the *real* value. Measuring of the total runtime of a benchmark set was done using the following command line: `time for i in *.tptp; do ${PROVER} $i; done`. The script *tptp_2_etptp.sh* converts TPTP files to E-prover input format and can be found in the *perf_test* directory.

At this time no source code is public available, we hope to change this in the near future.

The command line options mentioned in this paper are: *-p* to produce the evidence file, *-i* dumps the initial problem graph to a DOT file, *-F* dumps the relevant sub graphs of failed conjectured equalities.

Please note that the message `Failure` does not indicate an internal problem of vanHelsing. It means that no proof for the conjecture has been derived.

[1] This section will be removed in the final version of the paper