Trigger Selection Strategies to Stabilize Program Verifiers

K. Rustan M. Leino[0] and Clément Pit-Claudel[1]*

[0] Microsoft Research, Redmond, USA
leino@microsoft.com
[1] MIT CSAIL, Cambridge, USA
cpitcla@mit.edu

Abstract. SMT-based program verifiers often suffer from the so-called butterfly effect, in which minor modifications to the program source cause significant instabilities in verification times, which in turn may lead to spurious verification failures and a degraded user experience. This paper identifies matching loops (ill-behaved quantifiers causing an SMT solver to repeatedly instantiate a small set of quantified formulas) as a significant contributor to these instabilities, and describes some techniques to detect and prevent them. At their core, the contributed techniques move the trigger selection logic away from the SMT solver and into the high-level verifier: this move allows authors of verifiers to annotate, rewrite, and analyze user-written quantifiers to improve the solver's performance, using information that is easily available at the source level but would be hard to extract from the heavily encoded terms that the solver works with. The paper demonstrates three core techniques (quantifier splitting, trigger sharing, and matching loop detection) by extending the Dafny verifier with its own trigger selection routine, and demonstrates significant predictability and performance gains on both Dafny's test suite and large verification efforts using Dafny.

0. Introduction

Automated program verifiers like Frama-C [16], AutoProof [24], VeriFast [15], SPARK 2014 [13], and Dafny [17] provide usable environments in which to write provably correct programs. By employing efficient (semi-)decision procedures (found in satisfiability-modulo-theories (SMT) solvers [10, 8, 3, 2]) and aggressive caching [4, 19], these verifiers provide users with generally responsive feedback, and by shielding the user from all direct interaction with the decision procedures, the program verifiers offer a gentle learning curve. While SMT solvers are often pretty darn fast, their efficiency ultimately involves various heuristics, which leads to a problem in SMT-based program verifiers: we call it the butterfly effect.

* Authors are listed alphabetically.

DOI: 10.1007/978-3-319-41528-4_20. Published in CAV 2016, Part I, LNCS 9779, pp. 361–381, 2016. The final publication is available at http://link.springer.com/chapter/10.1007/978-3-319-41528-4_20

The *butterfly effect* describes the phenomenon that a minor modification in one part of the program source causes changes in the outcome of the verification in other, unchanged and unrelated parts of the program. When this change in outcome causes the verifier to hit a time limit or other resource limit, previously succeeding verifications turn into spurious verification failures. The butterfly effect thus leads to verification instability, user frustration, and overall a degraded user experience.

By profiling the behavior of an SMT solver (Z3 in the context of the Dafny program verifier), we have found many spurious verification failures to contain *matching loops*—ill-behaved quantifiers causing the SMT solver to repeatedly instantiate a small set of quantified formulas [10]. Such matching loops are bad news, but with some luck, the heuristic proof search in the SMT solver may happen to find a proof without falling into the trap of the matching loop. Evidently, such "luck" occurs often enough that when an unrelated change in the program source tickles the heuristics differently, latent matching loops are perceived as verification instability, whereas the real culprit was the presence of the matching loop in the first place.

In this paper, we contribute strategies for making quantifiers better behaved. The major part of the novelty of our strategies lies in a willingness to rewrite user-defined quantifiers. Our technique automatically selects *matching triggers*, which give a way to control how the SMT solver processes quantifiers. Because our technique finds candidate matching triggers *before* it rewrites quantifiers, we achieve better trigger selection than if the user had rewritten the quantifiers manually. Part of our strategies is also to select the triggers at the program-source level, rather than at the level of formulas in the SMT input. This is a good idea, because it lets our technique avoid some liberal triggers that consist only of functions added during the generation of verification conditions. Moreover, source-level trigger selection gives a clear way to explain to users which triggers were selected and how matching loops were averted, a simple but important feature for which we have received praise from users. We have implemented our strategies in the Dafny program verifier. Our paper also contributes experimental data that shows that our strategies significantly improve both predictability and performance of the verifier.

1. Background

In this section, we give the necessary background on how matching triggers are used by the SMT solver to handle quantifiers and on the architecture of the program verifier.

1.0. Matching Triggers

We assume the SMT solver deals with quantifiers along the lines proposed by Greg Nelson [22], which has been implemented, for example, in Simplify [10] and Z3 [8, 7]. The idea can be described as follows.

At any time during its proof search, the state of the SMT solver includes the set of formulas from which it is attempting to discharge the proof goal. These formulas are represented by various cooperating decision procedures. The decision procedure for uninterpreted functions is of special importance, as it not only keeps tracks of equivalence classes of terms (typically in a data structure called an *E-graph* [11, 23]), but also serves as a mediator between the theories. When an existentially quantified formula is equated with `true`, it is Skolemized and the resulting formula is equated with `true` in the E-graph. When a universally quantified formula is equated with `true`, the strategy of the SMT solver is to instantiate the quantifier and equate the resulting formulas with `true`.

Logically, it is sound to instantiate a universal quantifier with anything at all. However, arbitrary instantiations are not likely to help the proof search. To make more informed instantiation decisions, the SMT solver relies on *matching patterns*, also known as *matching triggers* or just *triggers*. Given some triggers for a quantifier, the SMT solver looks for terms in the E-graph that match any of the triggers and then instantiates the quantifier accordingly. The process is a bit like in term rewriting, except that the new instantiations are added to the E-graph, rather than somehow replacing the matching terms.

Let us illustrate with an example. A possible trigger for the quantifier

```
forall x: int · f(x) == 3 * g(x) + 5
```

is `f(x)`, meaning that the presence of any term `f(E)` in the E-graph gives rise to the instantiation `x := E` (yielding the formula `f(E) == 3 * g(E) + 5`). From now on, we will surround the terms of a trigger with curly braces. In this example, we may choose to understand the trigger as saying "wherever there is an interest in `f`, instantiate the quantifier". Another possible trigger is `{g(x)}`, which would have the effect of producing information about `g` in terms of `f`. It is possible to associate both triggers with the quantifier, which says to instantiate the quantifier if either an `f` term or a `g` term is found in the E-graph: `{f(x)} {g(x)}`. Yet another possibility is to use the trigger `{f(x), g(x)}`, which says to instantiate the quantifier only with those terms that appear in the E-graph as arguments to both `f` and `g`.

Here is more subtle example. The quantifier

```
forall x: int · 0 < x ⟹ f(x) == f(x-1) + f(g(x))
```

may tempt us to consider any of `{f(x)}`, `{f(x-1)}`, or `{f(g(x))}` as candidate triggers. However, `{f(x)}` is problematic. If the E-graph contains a term `f(E)`, then the instantiation `x := E` will produce a term `f(E-1)`, which gives rise to another possible instantiation, `x := E-1`. This is known as a *matching loop* [10] and should be avoided. The fact that the term `f(E-1)` in the instantiation is guarded by the antecedent `0 < E` does not help, because `E` may be term whose distance from `0` cannot be determined from the proof context.

Candidate trigger `{f(x-1)}` is also problematic, but for another (or rather, additional) reason: it contains the symbol `-`, which is interpreted by the decision procedure for arithmetic. When a symbol is interpreted, one cannot rely on it appearing in this form in the E-graph. For example, the E-graph may contain

a term `f(y+2)` but this may still not cause the instantiation `x := y+3`, because the term `y+3`, let alone the equality `y+2 == (y+3)-1`, may be unknown to the E-graph.

Candidate `{f(g(x))}` does not suffer from the problems we just described. It is rather discriminating—it will cause an instantiation `x := E` only if the E-graph contains a term that applies `f` to a term in the equivalence class where `g` is applied to `E`. Of course, depending on the application, this is possibly too discriminating to give rise to the instantiations that are needed to reach the proof goal.

Triggers can be specified as part of the SMT-LIB 2 input to the SMT solver. In effect, this provides a way to program the SMT solver [20]. If the input does not specify a trigger for a quantifier, the SMT solver attempts to select triggers from the terms in the quantifier's body. As we argue in this paper, leaving trigger selection to the SMT solver can contribute to verification instabilities. Instead, we show a strategy for selecting triggers at a level closer to the problem domain. A formal semantics of triggers, as well as ways to define decision procedures using quantifiers and triggers, has been studied by Dross et al. [12].

1.1. Architecture of the Program Verifier

The verification conditions generated for a Dafny program contain quantifiers from three major sources. One source is the axiomatization of standard Dafny operators and types, like the axiomatization of finite sequences. These quantifiers have hand-written triggers. A second source is the encoding of constructs defined in the Dafny program, like user-defined recursive functions. The triggers for these quantifiers come from hand-crafted schemas (see, e.g., [0]). The third form is user-written quantifiers. Previously, these were translated into SMT input without any attempts at computing triggers. Consequently, it had been left to the SMT solver to select triggers. To better understand why this can cause problems, let us say a few words about the architecture of the verifier and about what we will call parasitic terms.

The architecture of the Dafny verifier is the standard one of translating the source language into an *intermediate verification language* (Dafny uses Boogie [1, 18]) and then generating (using the Boogie tool) verification conditions from it. Each of these two steps does a translation into more coarse-grained types and more concrete encodings.

For example, consider the Dafny expression

```
forall x · 0 ≤ x < a.Length ⟹ a[x] == 31
```

where `a` denotes a reference to an integer array. In the translation of this expression into Boogie, the heap dereference is made explicit and the offset into the array is computed from the integer `x`. Here is (a slight simplification of) the Boogie encoding:

```
forall x: int ·  0 ≤ x ∧ x < _System.array.Length(a) ⟹
                   Unbox(read($Heap, a, IndexField(x))) == 31
```

As one can glean from this expression, the logical encoding of Dafny uniformly stores array elements as being of a type `Box`, so the read from the heap is followed by an `Unbox` operation. Furthermore, when Boogie translates this expression into SMT input, the formula becomes:

```
forall x: int · 0 ≤ x ∧ x < _System.array.Length(a) ⟹
    U_2_int(Unbox(intType, MapType1Select($Heap, a, IndexField(x)))) = 31
```

Here, we see yet another translation between types, where most of Boogie's types are collected into a type called `U` and the formula includes a mapping from `U` to `int` [18].

As can be seen in this example, what at the level of Dafny seems like a good trigger—the term `a[x]`, which would express "whenever there is an interest in an element of `a`, instantiate the quantifier"—is not easily identifiable in the SMT formula. In fact, at the SMT level, the term `IndexField(x)` may look like a good trigger, but that is quite a liberal trigger and is likely to lead to far too many irrelevant instantiations. We call the terms involving these additional functions *parasitic*. In other words, parasitic terms are terms introduced by Dafny's translation solely for encoding purposes; examples include `Box`, `Unbox`, and `IndexField`.

The Dafny front-end and verifier already infer from the given program various pieces of information, like omitted types and rank functions for termination. Dafny IDEs make this information available as hover text. In line with this tradition of informing users about inferred elements, our attitude is that Dafny users should not need to write triggers themselves, but may need to understand triggers in order to diagnose poor verification performance. It would thus be nice to communicate selected triggers to the user. Unfortunately, triggers selected in the SMT solver are difficult to obtain, and their inclusion of parasitic terms would not make sense to the Dafny user. Moreover, in many cases, translating back from a trigger picked by the SMT solver to a Dafny expression is hard: the translation from Dafny to the SMT solver's language is not a bijection.

These two aspects (avoiding parasitic terms and informing the user about trigger selections) lead us quite naturally to argue that it makes sense to select triggers at the source level rather than leaving this task to the SMT solver. From the use of good triggers, one can hope for better behaved instantiations and thus more stable verification. Moreover, information from the trigger-selection process can more easily be explained to the user through hover text. Of course, selecting triggers at the source level would be easy to accomplish by applying the SMT solver's algorithms to the nodes of a different abstract syntax tree (AST). However, our strategy goes beyond merely selecting triggers at the source level, as we explain in the next section.

2. Trigger Selection

When tasked with adding triggers to a quantifier, our code proceeds in a series of small steps. At a high level, it first walks down the AST of the body of the

quantifier, collecting terms that could be part of a trigger. Then, it enumerates subsets of these terms, thus generating a large collection of trigger candidates (each candidate trigger is a set of terms). It then rejects candidates that fail to mention all quantified variables, and filters the set of candidates to remove redundancy and improve the performance of the SMT solver. Finally, it uses heuristics to select relevant triggers, attempting to predict and prevent matching loops, and issuing warnings if matching loops seem unavoidable.

Each individual step is detailed below. Though previous literature has not given them extensive treatment, and in particular not with a focus on efficiency of implementation, some of these steps have appeared in one form or another in previous work. Apart from a rigorous description, our contribution lies beyond these steps: in addition to annotating single quantifiers, we introduce two new techniques, quantifier splitting and trigger sharing. These techniques are the key to preserving as much expressiveness as possible despite matching loop suppression. We describe them in detail at the end of this section.

2.0. Annotating the AST

Our extension first annotates subterms of each quantifier's body by labeling some of them as trigger heads, and others as trigger killers. *Trigger killers* are terms that are not permitted to appear in triggers: typically, these have forms that do not reduce to uninterpreted functions, such as arithmetic operations (`a + b`) or logical connectives (`a` $\implies$ `b`). Conversely, *Trigger heads* are terms that may appear in triggers, such as applied functions (`P(x)`), array accesses (`a[x]`), member accesses (`this.someField`), or set membership tests (`a in S`). More precisely, trigger heads are nodes of the AST of a quantifier's body whose children include at least one of the quantified variables, and do not include trigger killers. After annotating each subterm, our code collects all trigger heads, and attaches them to the quantifier.

As an example, in `forall x · x in S` $\iff$ `f(x) > f(x+1)`, our prototype annotates `x in S` and `f(x)` as trigger heads, and `x+1` and `f(x) > f(x+1)` (and thus all of their ancestors, like `f(x+1)` and the whole body of the quantifier) as trigger killers.

This phase takes time linear in the cumulative size of the ASTs all quantifiers in the source program, which is bounded by the size of the source program itself.

2.1. Generating Candidates

After collecting terms suitable for inclusion in a trigger, our code generates candidate triggers by enumerating combinations of these suitable terms. Since the powerset of all collected terms can quickly grow large, this enumeration is restricted by ensuring that each generated candidate has two properties: *adequacy* as a trigger (each candidate mentions all variables, as required by the SMT solver), and *parsimony* (removing a term from any candidate causes it to become inadequate). This parsimony property is highly desirable: since it puts more constraints on quantifier instantiations, any non-parsimonious candidate matches less often than its parsimonious counterparts.

As an example of the effect of the parsimony requirement, consider a collection of three suitable terms `P(x, y)`, `Q(y, z)`, and `R(x, z)`. From this collection, our code constructs three candidate triggers: `{P(x, y), Q(y, z)}`, `{P(x, y), R(x, z)}`, and `{Q(y, z), R(x, z)}`. `{P(x, y), Q(y, z), R(x, z)}` is eliminated because it is redundant (we call a candidate trigger *redundant* when it is strictly more specific than another candidate trigger; this happens when any match against the more specific trigger induces a match against the less specific one).

The parsimony requirement is particularly useful when the body of a quantifier mentions many predicates (in the extreme case of `forall x · P1(x) ... Pn(x)`, it allows our code to generate only n candidates, instead of the naïve 2^n), but implementing it efficiently is non-trivial. Indeed, it is not enough to check as subsets are enumerated that each added term mentions a previously unmentioned variable: the addition of a new term can make a previous term redundant, as for example when adding `R(x, y, z)` to a candidate containing `P(x)` and `Q(y)`. To track parsimony efficiently, our code keeps track of ownership relations between terms and variables. When recursively constructing subsets of a given set of terms, our code first ensures that the newly added term does mention a previously unmentioned variable; if so, the term is added, and it gains ownership of all variables that it mentions. After this operation, if any term is left without ownership of any variable, the whole subset is marked as redundant, and that branch of the subset generation recursion is cut. For performance, as it recursively constructs subsets, our code equips each partially constructed set of terms with two hashmaps: one hashmap from each term to the set of variables owned by that term, and the other hashmap from each variable to its (single) owner, if any. By incrementally constructing these hashsets of variables owned by each term and hashmaps associating each variable to its owner term, our code can efficiently (in time linear in the number of quantified variables in the context) determine whether adding a term to a partially constructed set makes it redundant (with regard to variables mentioned)[0].

Continuing on the previous example `forall x · x in S` $\iff$ `f(x) > f(x+1)`, our code generates only two triggers: `{f(x)}` and `{x in S}`. The candidate trigger `{f(x), x in S}` is redundant, and thus our code excludes it.

Without the parsimony requirement, this step would have for each quantifier a worst-case time complexity exponential in the number of previously collected trigger heads. Thanks to that requirement, however, this step has complexity $k \cdot n \cdot n_1 \cdot \ldots \cdot n_k$ where k is the number of quantified variables and $n = \sum_i n_i$ is the number of trigger heads in the quantifier's body (each n_i counts how many of these terms mention the i^{th} quantified variable). This yields an upper bound of $k \cdot n^{k+1}$; in practice, this upper bound is seldom reached: each trigger head often mentions a single quantified variable.

[0] Curious readers are directed to the `CopyWithAdd` method of the `SetOfTerms` class implemented in the `Triggers/TriggerUtils.cs` part of our implementation.

2.2. Picking Triggers and Preventing Matching Loops

In its last phase, our code uses a (necessarily incomplete) heuristic to evaluate whether each candidate trigger may cause a matching loop. Roughly, this heuristic flags a trigger as potentially looping if instantiating the quantifier may lead to a ground term that again matches the trigger. In more details, our code proceeds as follows for each candidate:

0. For each term `t` of the candidate, our code collects all terms of the quantifier's body that match the term `t`. For example, it may pick `f(x+1)`, `f(if a then b else c)`, and `f(0)` for the candidate term `f(x)` (a match occurs with a term `t'` when `t'` can be unified with `t`; that is, when there exists an instantiation of the variables of `t` that yields `t'`).
1. For each matching term, our code decides whether the match should be deemed a false positive, and if so removes it. False positives are terms that
   - are equal to the trigger (a term `f(x)` does not cause loops if the trigger is `f(x)`),
   - also appear in the trigger (a term `f(g(x))` does not cause loops if the trigger is `{f(y), f(g(x))}`, despite being a match for `f(y)`),
   - differ from the trigger only by variables (a term `f(y, x)` is not deemed to cause a loop if the trigger is `f(x, y)`; indeed, `forall x,y · f(x, y) == f(y, x)` is a quantifier that harmlessly can use either term as a trigger), or
   - differ by terms that do not contain bound variables (a term `f(0)` does not cause loops if the trigger is `f(x)`).
2. If any terms are left that could cause loops, our code marks the trigger as risky, recording the terms with which it may loop, and excludes it from the pool of candidate triggers.

With matching loops mostly eliminated, our code then proceeds to pick triggers: it analyzes the set of generated trigger candidates, ordering them according to a trigger specificity relation (wherein a trigger is less specific than another one if every match of the latter is also a match of the former), and excluding all non-minimal candidates. Indeed, just like non-parsimonious candidates, non-minimal candidates are redundant.

Note that, crucially, this selection phase happens after the matching loop suppression phase. As an example of minimality, and of the importance of this ordering, consider a collection of two terms `f(x)` and `f(f(x))`. In this case, our minimality heuristic would retain only one trigger, `{f(x)}`. But since its matching loop detection logic preemptively removes `{f(x)}` from the candidate pool, `{f(f(x))}` is selected instead, indeed preventing a matching loop.

This phase has, for each quantifier, a time complexity bounded by $k \cdot n^2$, where k is the number of candidates, and n the size of the body of the quantifier (the quadratic factor is obtained by bounding the cost of comparing two terms for equality by the size of the quantifier's entire body).

2.3. Splitting Quantifiers and Sharing Triggers

The strategy presented above suffers from one crucial weakness: in practice, users tend to collect related conditions under a single quantifiers. Thus, expressions of the form `forall x · P(x) ∧ (Q(x) ⟹ P(x+1))` are quite common. The algorithm presented above finds `{P(x)}` and `{Q(x)}` as reasonable trigger candidates, but proceeds to eliminate `{P(x)}` from the candidates pool, noticing its potential to loop with `{P(x+1)}`. Unfortunately, this means that with the naïvely auto-generated trigger `{Q(x)}`, this quantifier is not enough to prove a proposition such as `P(0)`, which should follow trivially.

To offset this over-specificity, our code implements *quantifier splitting*, a Dafny-to-Dafny rewriting technique that splits user-written quantifiers to reduce the chances of large quantifier bodies causing self-loops. For the example above, it thus produces two quantifiers, not one, and proceeds to annotate them separately. `forall x · P(x)` gets a trigger `{P(x)}`, and `forall x · Q(x) ⟹ P(x+1)` gets a trigger `{Q(x)}`.

Exactly how to perform trigger splitting is an interesting design concern: on one extreme, one could simply split the body of the quantifier around conjunctions. On the other extreme, one could rewrite the body into conjunctive normal form before splitting. The former approach is too weak: it fails to split quantifier bodies in the common case where a collection of properties is predicated by a common condition, such as `forall i · 0 ≤ i < |a| ⟹ (... ∧ ... ∧ ...)`. The latter approach correctly handles this pattern, producing three predicated quantifiers, but can introduce exponential increases in formula size. Our approach is therefore a compromise between these two extremes, where only one level of normalization is allowed, around ⟹. This properly handles the example above, without leading to significant increases in formula size. It is also in line with Dafny's predicated quantifier notation: the example above can be written `forall i | 0 ≤ i < |a| · ... ∧ ... ∧ ...`, which gets split into three quantifiers of the same shape `forall i | 0 ≤ i < |a| · ...`.

Splitting a quantifier before performing matching loop detection, however, is still not enough to recover its original expressiveness: after splitting the example above into `forall x · P(x)` and `forall x · Q(x) ⟹ P(x+1)`, and assigning `{P(x)}` as the trigger for the first split quantifier and `{Q(x)}` for the second, adding a term `Q(n)` to the collection of ground terms of the SMT solver does not immediately entail learning `P(n)` (it did for the original quantifier with `{Q(x)}` as the trigger). To recover more of the lost expressiveness, our code enriches the pool of terms to be considered for trigger elaboration at the beginning of the trigger generation process by sharing candidates between all quantifiers that derive from the same split quantifier. This strategy, which we call *trigger sharing*, yields two triggers for the first split quantifier `forall x · P(x)`, namely `{P(x)}` and `{Q(x)}`. The second quantifier still only gets one trigger, since `{P(x)}` would loop with `P(x+1)`. Interestingly, this strategy leads to quantifiers whose triggers are composed of terms that do not necessarily appear in the quantifier's body; an otherwise uncommon, but not undesirable, situation: in a sense, the shared trigger captures

insight gathered from the programmer about weak connections between relatively independent propositions.

This phase has a time complexity linear in the size of each quantifier.

3. Evaluation

Our new trigger selection strategy brings three main benefits:

- **Verification is more predictable**: adding auto-generated triggers to split quantifiers significantly reduce verification instabilities. We demonstrate this effect on Dafny's test suite by comparing the standard deviations of test running times across multiple runs with different seeds, with and without Dafny-generated triggers.
- **Verification is faster**: Dafny-generated triggers prevent certain matching loops from occurring, improving verification times. We demonstrate this effect on a large verification effort from the IronFleet project [14].
- **Debugging is easier**: Dafny-generated triggers are built from terms found in the bodies of user-written quantifiers, allowing Dafny to print detailed warnings and error messages. We discuss our experience using these messages as a debugging aid while adjusting examples that relied on promiscuous trigger instantiations to verify, and while verifying new programs.

We discuss these three aspects in the following subsections, before presenting experimental results supporting these conclusions.

3.0. Improved Predictability

Measuring the impact of Dafny-level trigger generation on predictability is particularly difficult. Indeed, triggers selected at the Dafny level are often much more specific triggers than those which the SMT solver would have picked from encoded terms. This is mostly desirable, but it still causes a number of problems: certain examples for which the SMT solver was lucky and could find a proof of thanks to absurdly liberal triggers stop working; other examples that (again, due to luck) had a short proof using promiscuous triggers take much longer to verify. In most cases, the problems are easily fixable: tightening the proof by adding a few assertions is often enough to drive the solver down the right proof path, without using unreasonable triggers. To ensure fairness, our experiments separate the contribution of these modifications to improvements in verification performance or stability; yet we still view these added annotations as net gains: it is preferable to invest slightly more work in ensuring that a verification is stable than to rely on luck and liberal trigger choices.

Looking more closely at instabilities, we see two ways in which unpredictable performance manifests itself: one derives from variations in solver behavior due to choices of random seeds, and the other derives from the specific way in which the verification problem is stated by the user. Ideally, we wish our verification tool to be robust to both.

In the first case, we expect a robustly annotated collection of quantifiers to be agnostic to the random choices that inform the solver's behavior, and thus verification times to be mostly independent from the choice of random seed used by the solver, or from the version of the solver used. We call this static predictability, or robustness to prover changes.

In the second case, we expect equivalent formulations of verification conditions to lead to roughly similar verification performance, and equal verification results. In particular, we do not expect users to adjust their writing to minimize unpredictable verification performance, or to make insignificant source changes to finding "the" right formulation that causes Dafny to succeed (unfortunately, the current verification process encourages users to do precisely this: since triggers are hard to debug, it is often simpler to experiment with various formulations of the problem, until one is found that does not seem to send the prover into a loop). We call this dynamic predictability, or robustness to source changes.

3.1. Improved Verification Speeds

Beyond the improved predictability, Dafny-generated triggers offer a significant performance boost in large verification projects. Analysis of traces produced by the SMT solver shows that in many cases, lax triggers are the cause of numerous useless instantiations, slowing down the proof search process. By ensuring that we never pick generic, uninformative triggers, we can perceptibly reduce the verification times of many complex developments.

3.2. Easier Debugging and Interaction

Beyond performance and predictability, Dafny-generated triggers provide a meaningful improvement in user experience:

- Triggers are Dafny terms, and can therefore easily be displayed to the user, in the form of informative tooltips in Dafny IDEs. In contrast, triggers collected by the SMT solver are parts of Dafny's and Boogie's combined encodings, and thus hardly meaningful to the user.
- Quantifier splitting happens at the Dafny level, and therefore is also amenable to presentation to the user, in the form of subtle mouseover messages.
- Potential matching loops can be detected early, and reported to the user. Instead of being an arcane part of SMT performance debugging, triggers become a discreet part of the usual verification landscape, which the user is reminded of only when it may start causing issues.
- Common patterns introducing matching loop (such as `forall x · a[x] ≤ a[x+1]`, or `forall x · f(x) = f(f(x))`) can be the object of specific advice, beyond the "potential matching loop" warnings. Our implementation does not offer such specialized advice, but it would be a reasonable extension of our current efforts.

In practice, we have received very enthusiastic feedback from users of the new system, and praise for the integrated, single-language experience that triggers at the Dafny level allow.

3.3. Experimental Results

We evaluate the impact of Dafny-generated triggers on Dafny programming through three experiments:[1]

- In the first one, we run most of the Dafny test suite (about 350 test files of varying complexity and running times) with varying random seeds, and measure per-test completion times and standard deviations across ten runs. We run the original Dafny code on the original test suite, followed by our own version of Dafny with and without triggers, on a version of the test suite updated to verify successfully with Dafny-generated triggers. Running these three tests allows us to evaluate performance and static predictability gains derived both from adding triggers to user-written quantifiers, and from editing the test suite to correct the warnings issued by our trigger-generating code.
- In the second experiment, we use our implementation to process a large code base after enabling Dafny-generated triggers, and show significantly improved verification times.
- In the third, informal experiment, we fully verify a version of the classic Union-Find algorithm, and discuss how auto-triggers improve the verification experience as the code and its proofs are being written. Contrary to our performance and static predictability tests, this experiment shows the dynamic predictability benefits of using Dafny-generated triggers throughout the process of developing and verifying new code.

3.3.0. Performance and Stability Evaluation Across Multiple Runs of the Test Suite Most of Dafny's test suite (perhaps 70%) is unaffected by the addition of Dafny-generated triggers: verification is and remains stable, with newly added triggers often slightly reducing variance across multiple runs, and performance remaining mostly unchanged. This is expected (and fortunate!), for two reasons: first, most of the test suite is made up of small programs, whose complexity is too low to cause significant verification issues. In that case, more precise triggers can help direct the search faster, but not by very much. Second, we expect matching loops to be relatively uncommon (especially given that the more complex examples in Dafny's test suite were written by experts). Figure 0 shows a high-level summary of these results, by comparing the distribution of standard deviations of verification times across the entire updated test suite, with and without Dafny-level triggers. In general, the addition of triggers slightly improves stability, diminishing variance across the entire test suite.
Beyond this general improvement in stability, measuring performance and instabilities across prover runs for the remaining 30% of the test suite shows many interesting patterns. Figure 1 shows detailed measurements for a few such examples. To produce it, we repeatedly ran Dafny in three distinct configurations:

[1] All experiments in this section were run on two cores of an Intel Core i7-4810MQ CPU (2.80GHz), on a machine equipped with 32GB of RAM and running Z3 version 4.4.1.

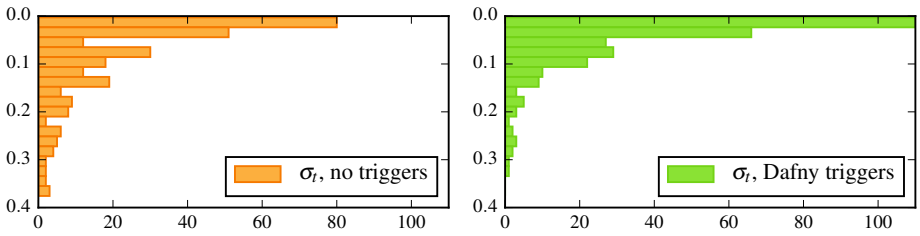

Figure 0. Histogram of standard deviations of verification times in seconds across five runs of Dafny's entire test suite, with and without Dafny-generated triggers. The right figure shows that using Dafny-generated triggers yields a marked shift towards lower deviations (i.e. towards the top of the figure).

- With Dafny-generated triggers, on a copy of the test suite modified to work with these new triggers (these modifications are described at the end of this section)
- Without Dafny-generated triggers, on the same modified test-suite
- Without Dafny-generated triggers, on the original test suite

In all three cases we ran Dafny 10 times, passing a new random seed to Z3's arithmetic, SMT and SAT modules on each run[2].

Figure 1 shows the performance and predictability consequences of automatically selecting triggers for a corpus of six example programs taken from Dafny's test suite. These programs are a mix of algorithms and submissions to various verification competitions, with about 100 to 300 lines of Dafny code each.

We conclude this section with a quick review of the changes that adapting the Dafny test suite to use our trigger generation strategy required. With automatically generated triggers, but no loop detection nor quantifier splitting, about 55 tests (out of 350) initially failed to verify. Adding matching loop detection fixed about 10. Adding quantifier splitting with trigger sharing fixed 10 more. For the remaining 35 tests, the causes were distributed as follows:

- About 10 tests were using explicitly recursive constructs, where matching loops were expected, and needed no changes beyond silencing warnings
- About 10 tests were implicitly relying on excessively liberal triggering to prove complex correspondences between expressions involving sequences (looking at a Z3 trace for the offending quantifier would show that Z3 was picking a very unspecific term to trigger on, and triggering a lot). Adding stricter triggering annotations caused these sequence equivalences to become unprovable; it was easy to fix these issues by adding extra annotations.
- About 5 tests had unnecessary matching loops, which can be fixed by rephrasing quantifiers, thus acting upon the warning issued by our implementation.

[2] The seeds used were 32901, 52510, 15712, 371, 65410, 21223, 38836, 27584, 7013, and 11502.

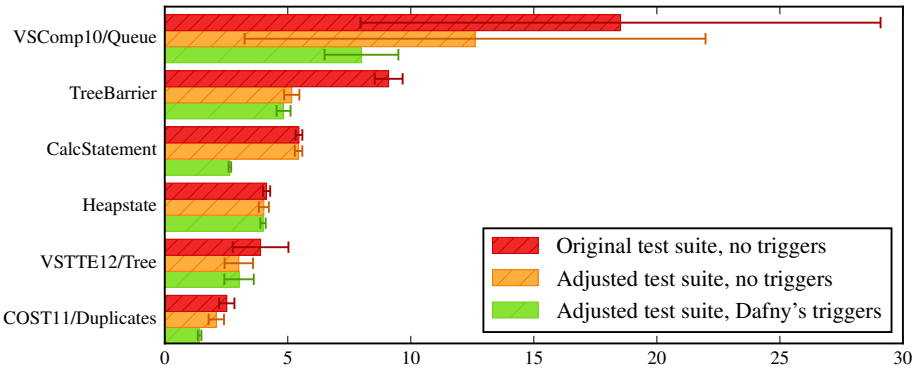

Figure 1. Verification times in seconds for six example programs taken from Dafny's test suite, running the original Dafny on its test suite, our own version of Dafny with trigger generation turned off on an updated copy of the test suite, and our own version of Dafny with trigger generation turned on that same copy of the test suite. Error bars show the standard deviation of verification times across ten runs with distinct random seeds.

– The rest had various issues, where specific properties were not being proved due to stricter triggering. In most cases, adding extra assertions was enough to lead Z3 to a proof, in a much more principled way that the haphazard matching that was occurring in the original Dafny.

3.3.1. Performance Results on Large Verification Efforts Focusing on verification performance, our second experiment pitted our implementation against the original Dafny to check the proofs of the implementation layer of IronFleet's IronRSL, a Paxos-based replicated-state-machine library. The focus of our attention was thus a collection of 48 source files totaling 13916 lines of Dafny source code. Figure 2 compares the running times of each of the 48 files with and without Dafny-generated triggers, sorted by descending relative improvement. Across the full corpus, our implementation achieves an overall speedup factor of 1.6, reducing the total running time from 1 hour and 4 minutes to 39 minutes. The average speedup across the test suite is 1.15, and the average speedup on tests that the change significantly affects ($> 20\%$ speedup or slowdown) is 1.45. These results are even more encouraging that these programs were written by experienced Dafny hackers.

3.3.2. Experience Report on Verifying a Simple Algorithm With and Without Dafny-Generated Triggers As a final experiment, we informally assessed the dynamic robustness of our new implementation. We verified a number of small programs using Dafny with auto-generated triggers, including a Union-Find implementation. As we worked on it, we did not notice significant differences: everything was running smoothly, and Dafny was not reporting specific warnings

Figure 2. Verification times in seconds for the 48 programs composing the implementation layer of IronRSL, with trigger generation at the Dafny level turned on and off. The experiment shows a significant speedup across the entire library, with programs that suffer from the change being only slowed down by a small proportion.

about our quantifiers. Switching to a different environment, however, revealed how much Dafny-level triggers were doing for us: at multiple points we tried to verify the program without these auto-selected triggers; in most cases, verification simply timed out.

In total, it took about 10 hours to write 330 lines of verified Dafny code. The program includes 25 universal quantifiers; checking its proofs takes 18 seconds with Dafny-generated triggers, and 31 seconds without. Most of the performance difference results from Z3 exploring fruitless paths due to overly permissive triggers.

Figure 3 shows an example of a quantifier annotation, displayed as an in-editor mouseover tooltip.

4. Related Work

The idea of using matching triggers to instantiate quantifiers in SMT solvers stems from Nelson's PhD thesis [22] and they were first implemented in the SMT solver Simplify [10]. When a quantifier is not given an explicit trigger, Simplify attempts to select one. It first looks for single, minimal terms that can be used as triggers, and selects all of these, except ones that would give rise to a matching loop for the quantifier according to a simple heuristic. If no such triggers exist, then Simplify attempts to find a trigger with multiple terms. It only picks one multiple-term trigger in order to avoid having to consider exponentially many. In contrast, we may consider polynomially many. This gives us extra flexibility in trigger choices.

Z3 has a similar trigger-selection mechanism, but due to its more efficient matching technique [7], it does not make a hard distinction between single-term

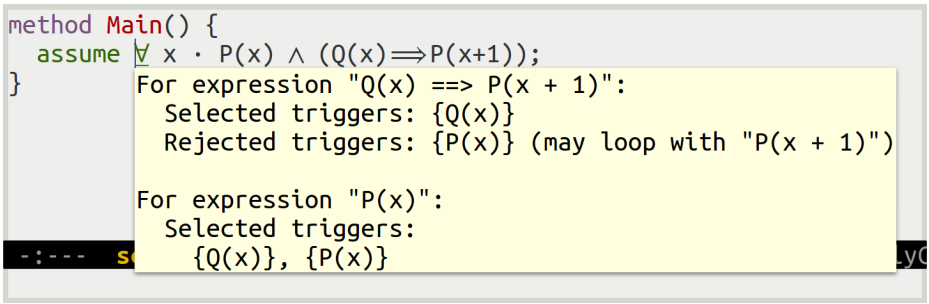

Figure 3. Emacs' *dafny-mode* (part of the *boogie-friends* package) showing a trigger-related message. Our code has correctly split the quantifier, adding two triggers (`{P(x)}` and `{Q(x)}`) to the first half, and a single one (`{Q(x)}`) to the second half, thus avoiding a matching loop. This type of information was useful for our Union-Find experiment.

triggers and multiple-term triggers. On the downside, Z3 does not check for matching loops.

SMT solvers CVC4 [2] and Alt-Ergo [3] also support quantifiers and triggers. We have tried running Dafny with CVC4 version 1.4, which supports SMT-LIBv2 input. Unfortunately, this version of CVC4 fails to verify most of Dafny's test suite, except in tiny examples where the verification conditions do not involve any significant quantifiers. Some preliminary experiments with the upcoming version 1.5 show promise of being a viable alternative to Z3 for Dafny.

The program verifier VCC [6] also computes its own triggers for user-written quantifiers, rather than leaving this to the SMT solver. The selection criteria is aware of the VCC style of specifications and gives priority to certain terms, for example those that mention the special "ownership" field `\owns` or user-defined functions. The quantifiers are not rewritten for the purpose of finding better triggers, but some form of loop prevention is used.

There have been other attempts to rewrite verification conditions in order to make them perform better with SMT solvers. Böhme and Moskal measured the performance impact of different heap encodings [5]. In the context of Viper [21], Uri Juhasz has implemented Boogie-to-Boogie transformations that summarize in join nodes information common to all branches, which can reduce the need for case splits in the SMT solver and thus increase performance [@Uri-Juhasz:PersonalCommunication].

5. Future Work

While our work addresses the most prevalent source of verification instability we have found, there are other sources.

One other source that involves quantifiers involves a quantifier that many users write: expressing with a quantifier that an array is sorted

```
forall i · 0 ≤ i < a.Length - 1 ⟹ a[i] ≤ a[i+1]
```

For this quantifier, our technique reports that it cannot find a valid trigger without introducing a possible matching loop. This provides a noticeable improvement over Dafny's previous behavior of silently accepting the quantifier, because it calls to the user's attention the fact that the quantifier may cause problems for the verifier. We could, however, go one step further and rewrite the quantifier:

```
forall i,j · 0 ≤ i < a.Length - 1 ∧ j == i+1 ⟹ a[i] ≤ a[j]
```

The trigger `{a[i], a[j]}` for this quantifier has the nice property that the instantiations it causes do not introduce any more array-dereference terms. We would like to introduce automatic rewrites of this form, but have not yet implemented them.[3]

Such automatic rewrites of problematic quantifiers could be investigated systematically, and could be distributed as an auto-fix IDE feature: in cases where obtaining a confirmation from the user before doing a rewrite is desirable, we would display a tooltip offering the rewrite. Such an investigation would make for good future work.

Another source of verification instability is the use of non-linear arithmetic. To keep such issues manageable, the Ironclad Apps project chose to mostly turn off Z3's support of non-linear arithmetic and instead rely on manually crafted lemmas about the needed properties. Providing better automated and stable support for non-linear arithmetic remains fertile research ground.

Finally, we would like to comment on the fact that we have implemented our matching-loop detection inside Dafny on a per-quantifier basis. Our infrastructure has the basic building blocks for doing the matching-loop detection given a larger collection of quantifiers. Within Dafny, a possible extension of our work would be to look for possible matching loops within some cluster of declarations, for example among all loop invariants that a user has supplied for a loop; a significant difficulty would be to deal with the exponential number of combinations that arise from matching sequences involving triggers composed of multiple terms in multiple quantifiers.

Outside Dafny, matching-loop detection could do well at the Boogie or Z3 level. This would allow non-Dafny tools to benefit from this functionality. One could also imagine an automatic postmortem analysis of an SMT-solver run to detect loops that caused bad performance. Trying to prove the absence of matching loops in a given verification condition would be wonderful. This seems related to termination issues in rewriting systems (see, e.g., [9]), but we are unaware of any work that specifically addresses this problem for triggers where congruence closure is involved.

[3] For this particular property, the quantifier `forall i,j · 0 ≤ i < j < a.Length ⟹ a[i] ≤ a[j]` is often even better, because it makes it more readily usable without having to appeal to transitivity of ≤ and induction.

6. Conclusion

We have presented effective strategies for selecting matching triggers for quantifiers that make the proof search of SMT solvers better behaved, thus improving the experience of users of automated program verifiers. Our implementation of these techniques in the Dafny program verifier demonstrates significant improvements in verification stability, verification performance, and proof elaboration and debugging experiences. We have received extremely positive feedback from early users of our implementation, on large verification efforts. By rewriting some quantifiers, our technique is able to select suitable triggers for quantifiers that otherwise would be ill-behaved or rejected for fear of matching loops. By applying our technique at the level of source expressions, we avoid triggering on parasitic terms introduced in the translation to first-order formulas, and obtain better-behaved triggers that we can directly report to users, thereby giving them meaningful feedback about the automatic trigger selection process.

We have tightened up a major source of verification instability. While other sources remain, we argue that our strategies are ready to be used in program verifiers and look forward to further stability improvements.

Acknowledgments We are grateful to Chris Hawblitzel and Bryan Parno for productive discussions and feedback during the development of the auto-generated triggers and for help in setting up the IronFleets experiments, to Michał Moskal for his help in understanding how VCC generates triggers, and to Claire Dross and the anonymous reviewers for their comments on drafts of this paper. We also thank Andrew Reynolds for discussions about quantifiers and Dafny support in CVC4, Jay Lorch for his help testing the Dafny mode for Emacs, and Daan Leijen for typesetting assistance in Madoko. A special thanks goes to Jonathan Protzenko for connecting us authors and thus kickstarting this collaboration.

A. Pseudo-Code for the Main Algorithm

This annex offers high-level pseudo-code for the main algorithm introduced in this paper; it simplifies the types of many of the relevant functions for clarity, and glosses over most performance optimizations discussed in the body of the paper.

```
def AnnotateAndSplit(quantifier):
  AnnotateSubtree(quantifier.body, quantifier.variables)
  candidates = TriggerCandidates(quantifier)
  for split_q in SplitQuantifier(quantifier):
    safe_candidates = RemoveLoops(candidates, split_q)
    split_q.triggers = PickTriggers(safe_candidates)

def AnnotateSubtree(node, variables):
  for c in node.children:
```

```
    AnnotateSubtree(c, variables)
  if (node.type in KILLER_TYPES or
      any(c.annot == TriggerKiller for c in node.children)):
    node.annot = TriggerKiller
  elif any(node.mentions(v) or any(c.mentions(v) for c in node.children)
        for v in variables):
    node.annot = TriggerHead

def TriggerCandidates(quantifier):
  for subset in Subsets(quantifier.trigger_heads):
    # Adequacy: All quantified variables are mentionned
    if is_adequate(subset, quantifier.variables):
      # Parsimony: No term can be removed without breaking adequacy
      if not is_redundant(subset):
        yield TriggerCandidate(subset)

def SplitQuantifier(quantifier):
  if quantifier.type == ForallNode:
    if quantifier.body.type == AndNode:
      for c in quantifier.body.children:
        yield ForallNode(c, quantifier.variables)
    else: yield quantifier
    # (... Similar case of existential quantifiers omitted)

def RemoveLoops(candidates, split_quantifier):
  for candidate in candidates:
    matches = []
    for term in candidate.terms:
      for desc in split_quantifier.descendants:
        if desc.can_unify_with(term, split_quantifier.variables):
          if not FalsePositive(desc, term, candidate):
            matches.append(desc)
    if not any(matches): yield candidate

def FalsePositive(desc, term, candidate):
  return (desc == term or desc in candidate.terms or
          all(is_var(t) or is_const(t) for t in term.disjoint_union(desc)))

def PickTriggers(candidates):
  for candidate in candidates:
    if not any(candidate.more_specific_than(other) for other in candidates):
      yield candidate
```

References

0. Nada Amin, K. Rustan M. Leino, and Tiark Rompf. Computing with an SMT solver. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs — 8th International Conference, TAP 2014*, volume 8570 of *Lecture Notes in Computer Science*, pages 20–35. Springer, July 2014.
1. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.
2. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification — 23rd International Conference, CAV 2011*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011.
3. François Bobot, Sylvain Conchon, Évelyne Contejean, and Stéphane Lescuyer. Implementing polymorphism in SMT solvers. In Clark Barrett and Leonardo de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo Theories*, pages 1–5, 2008.
4. François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Preserving user proofs across specification changes. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments — 5th International Conference, VSTTE 2013, Revised Selected Papers*, volume 8164 of *Lecture Notes in Computer Science*, pages 191–201. Springer, 2014.
5. Sascha Böhme and Michał Moskal. Heaps and data structures: A challenge for automated provers. In Nikolaj Bjand Viorica Sofronie-Stokkermans, editors, *Automated Deduction — CADE-23 — 23rd International Conference on Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 177–191. Springer, July–August 2011.
6. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, August 2009.
7. Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *Automated Deduction — CADE-21, 21st International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, July 2007.
8. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, March–April 2008.
9. Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1/2):69–116, February/April 1987.
10. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
11. Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, October 1980.

12. Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. In Pascal Fontaine and Amit Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012*, volume 20 of *EPiC*, pages 22–31. EasyChair, June–July 2013.
13. Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré, and Yannick Moy. Rail, space, security: Three case studies for SPARK 2014. In *7th Europen Congress on Embedded Real Time Software and Systems (ERTS^2 2014)*, 2014.
14. Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM – Association for Computing Machinery, October 2015.
15. Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.
16. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
17. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, April 2010.
18. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *LNCS*, pages 312–327. Springer, March 2010.
19. K. Rustan M. Leino and Valentin Wüstholz. Fine-grained caching of verification results. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification — 27th International Conference, CAV 2015, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 380–397. Springer, July 2015.
20. Michał Moskal. Programming with triggers. In Bruno Dutertre and Ofer Strichman, editors, *SMT 2009, 7th International Workshop on Satisfiability Modulo Theories*, August 2009.
21. Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation — 17th International Conference, VMCAI 2016*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, January 2016.
22. Charles Gregory Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC, June 1981. The author's PhD thesis.
23. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
24. Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems — 21st International Conference, TACAS 2015*, volume 9035 of *LNCS*, pages 566–580. Springer, April.