

# Programming with Triggers

Michał Moskal  
European Microsoft Innovation Center  
Aachen, Germany  
University of Wrocław  
Wrocław, Poland  
michal.moskal@microsoft.com

## ABSTRACT

We give a case study for a Satisfiability Modulo Theories (SMT) solver usage in functional verification of a real world operating system. In particular, we present a view of the E-matching pattern annotations on quantified formulas as a kind of logic programming language, used to encode semantics of the programming language undergoing verification. We postulate a few encoding patterns to be benchmark problems for a possible E-matching alternative. We also describe features required from the SMT solver in deductive software verification scenarios.

## Categories and Subject Descriptors

I.2.3 [Artificial intelligence]: Deduction and Theorem Proving—*Logic programming*; F.3.1 [Logics and meanings of programs]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

## Keywords

SMT, axiomatizations, E-matching, triggers, program verification

## 1. E-MATCHING FOR THEORY BUILDING

Satisfiability Modulo Theories (SMT) solvers decide satisfiability of first-order formulas in the presence of background theories (like integer or bit-vector arithmetic, arrays, etc.). Verification of hardware and software is a major application area of SMT solvers. This paper focuses on usage patterns of SMT solvers in software verification scenarios. In particular, we present a case study of applying VCC [5]<sup>1</sup>, a deductive verifier for C, powered by the Z3 [9] SMT solver,

<sup>1</sup>VCC, including SMT-support tools described later in the paper, is available for academic research, with source code, at <http://vcc.codeplex.com/>.

in a large operating system verification project (more in Section 1.2). VCC takes annotated C functions as input, and turns them, with the help of Boogie [1], into verification conditions (VCs). Validity of a VC implies (partial, as we do not check for termination) correctness of a program. Therefore, if a model for a negation of a VC can be constructed, it points to a possible problem in the function, while unsatisfiability of the negation of a VC implies correctness of the verified function.

Each verification tool depends on a *verification methodology*, dictating the specification language and commonly used specification idioms, as well as the particular modelling of the programming language semantics to be used. Therefore, from a SMT point of view, verification conditions should be evaluated modulo a theory describing the verification methodology. Clearly, no SMT solver supports such arbitrary theory out of the box. Moreover, given the complexity of such a theory and the pace at which it tends to evolve during development of a verification tool, it seems highly impractical to implement such theory inside of an SMT solver. This is why usually [11, 13, 2, 16, 12] in deductive verification a first-order axiomatization is developed, using theories available in the SMT solver (like uninterpreted functions, integer arithmetic, bit-vector arithmetic, and arrays). The formulas presented as axioms to the SMT solver should be understood as theorems in the model of the programming language semantics and verification methodology.

A verification methodology axiomatization consists mostly of universally quantified formulas. SMT solvers usually [10, 9, 14, 20] use instantiation techniques, controlled by user-supplied or heuristically chosen *triggers*, which are most often subterms of the quantified formula. The triggers are used to guide an *E-matching* procedure, which looks for ground terms matching the triggers modulo the equality relation considered in the current model (more in Section 2).

Quantifier instantiation with triggers is often viewed, especially in the SMT community, as an unreliable heuristic, with no completeness guarantees, developed for a legacy system (the SMT solver Simplify [10]) for solving first-order problems. On the other hand, the deductive verification community is generally not concerned with general first-order problems, and instead wants a way of encoding the semantics of the verified programming language. The views of trigger/axiomatization engineering vary from “we need even more control” to “let us pick some triggers and hope the magical SMT solver will get it right”. This chapter strongly supports the former camp: with unrestricted quantifier instantiation verification problems very quickly become

intractable for Z3, and the experience with Z3’s superposition calculi was similar.

## 1.1 Related Work and Contributions

With the exception of Spec#’s treatment of comprehensions [19], there has been not much publications about particulars of triggering. On the other hand several tools, including ESC/Modula-3 [11], ESC/Java [13], Spec# [2], Havoc [16], and Why [12] use these kinds of patterns. Only the encoding of ESC/Java’s logic is described in some more detail [23]. Overall it seems that there is not enough knowledge exchange between the SMT and deductive verification communities regarding these topics. We hope that this chapter will partially bridge that gap, and help develop alternatives to E-matching, by clarifying its present usage patterns.

We view the first-order logic, together with trigger annotations, as a logic programming language used to encode the semantics of the code being verified. This operational view is illustrated by a number of encoding patterns: frame clauses (Section 2.2) being source of large fraction of quantified formulas in a typical VC; versioning (Section 2.3), demonstrating the automatic trigger selection employed in Simplify and Z3 to be too restrictive; stratified triggering (Section 2.4), showing the opposite situation, with novel existential activation used to be again more liberal; and finally rather surprising behavior of a set theory axiom (Section 2.5). We also describe typical use cases of SMT in verification (Section 3), including the particular timing and output requirements placed on the SMT solver. Finally, we tackle the topic of debugging and profiling axiomatizations (Section 4), using tools we have built for this purpose.

The encoding patterns presented here are the most complex among ones we have used in VCC. We thus postulate them to be benchmark problems for a possible E-matching alternative.

Several axiomatization patterns we present are heavily influenced by the Spec# program verifier, due to similarities in treatment of ownership and framing. We note in the text when this is the case.

## 1.2 Background: The Hypervisor Verification and VCC

The Hypervisor verification project<sup>2</sup> aims at full functional verification of the kernel of Hyper-V, an industrial virtualization platform, currently shipped with Microsoft Windows Server 2008. It is essentially a small operating system, with memory management, a scheduler, and essential device drivers. It consists of about 100 000 lines of C code (excluding comments) and about 5 000 lines of assembly.

The ultimate goal of the project is a formal proof that Hyper-V simulates the virtualized hardware for each of the guest operating systems. There are however multiple intermediate goals, the first one being verification of memory safety in concurrent context. Even this first step relies on establishing, e.g., functional correctness of red-black trees and complex concurrency synchronization protocols.

The goal of the project is to verify the code that is shipped, not to change it just to facilitate verification. This requires handling C in its full “glory”, a restriction to a “safe subset”

is out of question. Moreover, the entire code-base should be verified, including concurrency control primitives (e.g., spin locks), which are usually taken for granted by verification methodologies. Finally, annotations are supposed to be maintained by the regular development team once the verification is complete. Because an average programmer is usually not an expert in interactive theorem proving, automatic methods should be used as much as practically possible. The project involves up to 20 people working, mostly on specification of the Hyper-V, for three years, making it one of the largest formal verification efforts ever attempted.

These conditions make for a fairly good case-study for verification in the “real world”.

VCC [5] is a tool used for Hypervisor verification. It was developed with the needs of Hypervisor verification project in mind, but given the scope of that project we expect it to be usable on wide spectrum of C programs. In particular, the verification methodology [7], seems applicable to a wide class of various concurrent algorithms.

VCC extends the C language with contracts in style of JML [17] and Spec# [2]. Functions are equipped with pre- and post-conditions while types (i.e., structures and unions) are equipped with two-state invariants, which describe valid states and possible changes of objects of those types. Contracts are specified in a variant of the C programming language consisting of side-effect free expressions, first-order quantification, and lambda expressions.

The annotated C programs are translated, with help of the Boogie verification condition generator [1] to formulas understood by the Z3 [9] SMT solver. Even though Boogie has multiple theorem prover back-ends (not even restricted to first-order logic, e.g., there exists a back-end [4] for Isabelle/HOL), VCC currently focuses on the SMT back-end and Z3 in particular.

Verification in VCC is function- and thread-modular: each function is verified separately, as if executed by a single thread, where actions of other threads are simulated at certain points.

**First Order Manifesto.** Verification of complex, functional properties of programs has been, to date, mostly done using interactive, higher-order provers. To leverage automation offered by modern SMT solvers, VCC restricts the specification language not to use any higher order or specialized logics. The specifications are expressed using first-order predicates, possibly operating on *ghost state*, i.e., fields and objects introduced only for the purpose of specification. Ghost fields are used, e.g., to store a map-abstraction representing all nodes of a red-black tree in the tree object, or to capture concurrent protocols.

We have been able to specify and verify multiple recursive data structures, as found in the Hyper-V code, some complex synchronization primitives (spin locks, reader-writer locks, rundowns, custom algorithms for message passing) and specify a good deal of data structure invariants. We currently do not face expressiveness problems with the first-order specification language.

**Annotation Language Flexibility.** To facilitate the specification of complex functional properties, VCC supports manipulation of ghost data types, including maps (from pointers and integers to arbitrary types) as well as entire states of execution, which can be captured and used to evaluate expressions in them. Additionally, new user-defined

<sup>2</sup>It is part of the Verisoft XT verification project, supported by BMBF under grant 01IS07008. The Verisoft’s Aviation subproject, focusing on PikeOS embedded operating system verification [3] also uses VCC.

ghost data types can be specified at the level of C, using function symbols and axioms.

Foremost, however, VCC supports explicit triggers in user-supplied quantified formulas. We found this ability invaluable in specification of recursive data structures (Section 2.4), and helpful in a number of other situations. We intend to survey common triggering styles in specifications, toward the end of the project, to see if and how the trigger selection can be mechanized. Currently, however, we focus on a handful of “specification idioms”, which are “recipes” describing how to specify a particular implementation artifact, including triggers.

## 2. ENCODING PATTERNS

SMT solvers incrementally build a sequence of partial models for the ground (i.e., quantifier-free) part of the input formula. A term is *active* iff the current partial model gives it an interpretation. Based on *triggers* attached to the quantified formulas and the set of currently active terms, the quantified formulas are instantiated. The resulting ground instances are conjoined to the input formula, possibly leading to refinements of the model.

A *trigger* is a set of non-ground terms. The terms in a trigger need to mention all variables that are quantified over. A trigger is said to *match* in the partial model  $M$ , if there exists a substitution  $\sigma$  such that for each term  $t$  in the trigger,  $\sigma(t)$  is active in  $M$ . For example, a trigger  $\{f(g(x)), h(y)\}$  will match in a model where the term  $f(c)$  and  $h(e)$  are active and  $c = g(d)$  for  $\sigma = [x := d, y := e]$ . Note that there can be multiple triggers for a given quantified formula, each consisting of one or more terms. For each trigger, for each matching substitution, the formula will be instantiated (i.e., terms in a trigger are treated as conjunction, while different triggers are treated as a disjunction). The exact time at which the instance will be generated is determined heuristically. Experience with Z3, VCC and Spec# suggests eager instantiation to be the most efficient. Quantifier instantiation leads to refinements in the model, which give rise to new matches, new instantiations, and so on. Following Boogie, we list triggers in curly braces after the quantified variables of a formula.

### 2.1 The Simple: Tuples and Inverse Functions

This simple example shows how triggering can make the behavior of an SMT solver rather unpredictable. Let us consider a typical axiomatization of a pair constructor and selector functions:

$$\forall x, y. \{\text{pair}(x, y)\} \text{fst}(\text{pair}(x, y)) = x \wedge \text{snd}(\text{pair}(x, y)) = y$$

In other words, whenever the term  $\text{pair}(t, s)$  becomes active, the axiom will also activate (and give value to) the selector functions. Thus, if an assumption like  $\text{pair}(0, a) = \text{pair}(1, a)$  is present, the axiom will, through congruence closure, cause  $0 = 1$  to be assumed. On the other hand, should we select  $\{\text{fst}(\text{pair}(x, y))\}$  as the trigger, which would be natural if we thought of the axiom being the definition for the  $\text{fst}(\dots)$  function, an assumption like the above alone would not trigger the axiom. Only if the terms  $\text{fst}(\text{pair}(0, a))$  and  $\text{fst}(\text{pair}(1, a))$  would happen to be active, possibly because of some other proof obligations, would the axiom trigger and cause inconsistency to be detected. This would generally cause unpredictable behavior of the SMT solver: a proof of a particular assertion could be dependent on some

unrelated previous proofs. Therefore, the author of the axiomatization needs to identify the cases where the existence of an “interface” function like  $\text{fst}(\dots)$  is also used to derive some properties of the objects it is applied to, in particular distinguishing between different instances of such objects.

#### 2.1.1 Extensible Records

Consider the tuple example again, but one where we do not define the constructor function (or the definition axiom) at all. Instead, whenever we need to construct a tuple object, let us say  $\langle 1, 2 \rangle$ , we would introduce a new constant  $c$ , and assume  $\text{fst}(c) = 1 \wedge \text{snd}(c) = 2$ . Because there is no mention of the constructor function, new fields can be added freely, assuming the cardinality of the type of  $c$  is big enough. For example, VCC background axiomatization defines several selector function on program states, including one for memory values ( $\text{state}_{\text{mem}}$ ) and one for status (ownership etc.,  $\text{state}_{\text{st}}$ ). We then define helper functions to access different “dimensions” of state. Finally, we subdivide the information about ownership of a particular pointer even further, using  $\text{status}_{\text{closed}}$  and  $\text{status}_{\text{closed}}$  selector functions:

$$\begin{aligned} \text{memory}(S, p) &\equiv \text{rd}(\text{state}_{\text{mem}}(S), p) \\ \text{status}(S, p) &\equiv \text{rd}(\text{state}_{\text{st}}(S), p) \\ \text{owner}(S, p) &\equiv \text{status}_{\text{owner}}(\text{status}(S, p)) \\ \text{closed}(S, p) &\equiv \text{status}_{\text{closed}}(\text{status}(S, p)) \end{aligned}$$

The reason for such a two-stage encoding is performance. For example, ordinary memory write putting value  $v$  at pointer  $p$  is going to turn a state  $S_0$  into  $S_1$ , where only  $\text{state}_{\text{mem}}$  is updated, while  $\text{state}_{\text{st}}$  stays unchanged:

$$\begin{aligned} \text{state}_{\text{mem}}(S_1) &= \text{wr}(\text{state}_{\text{mem}}(S_0), p, v) \wedge \\ \text{state}_{\text{st}}(S_1) &= \text{state}_{\text{st}}(S_0) \end{aligned}$$

Subsequent reads from  $\text{state}_{\text{st}}(S_1)$  do not need to go through any quantifier instantiation to be transformed into reads on  $\text{state}_{\text{st}}(S_0)$ . On the other hand, the ownership-related information tends to be updated all at once, and therefore there is no reason for separation of heaps. For example, closing an object  $p$  and setting its owner to  $o$  is done with the following assumption<sup>3</sup>:

$$\begin{aligned} \text{state}_{\text{mem}}(S_1) &= \text{state}_{\text{mem}}(S_0) \wedge \\ (\exists s. \text{state}_{\text{st}}(S_1) &= \text{wr}(\text{state}_{\text{st}}(S_0), p, s)) \\ \wedge \text{owner}(S_1, p) &= o \wedge \text{closed}(S_1, p) \wedge \dots \end{aligned}$$

We postulate existence of a status of  $p$  such that the owner is  $o$  is  $p$  and  $p$  is closed. Alternatively, instead of the existential quantifier, one could say that the new state is after update is what it is:

$$\begin{aligned} \text{state}_{\text{mem}}(S_1) &= \text{state}_{\text{mem}}(S_0) \\ \wedge (\text{state}_{\text{st}}(S_1) &= \text{wr}(\text{state}_{\text{st}}(S_0), p, \text{status}(S_1, p))) \\ \wedge \text{owner}(S_1, p) &= o \wedge \text{closed}(S_1, p) \wedge \dots \end{aligned}$$

which might be trickier to understand, but is otherwise very similar. Another example is pointers to ghost state, which we can draw freely from the set of integers, and thus they can encode arbitrary amounts of information. In particular, pointers to certain objects encode versions of ownership domains.

<sup>3</sup> The ownership information also includes time stamps, reference counts, and so on, which tend to be updated all at once, even if closedness and ownership do not.

## 2.2 The Common: Framing in the Heap

Basically any reasoning in deductive verification builds on top of heap updates and accesses. This suggests the heap encoding to be crucial for performance. In fact the time of reasoning about the heap is dominant in VCC problems. This section gives an overview of the heap encoding, as used in VCC and Spec#, with some references to other systems. The VCC heap<sup>4</sup> is axiomatized using standard select-of-store axioms:

$$\begin{aligned} \forall H, p, v. \text{rd}(\text{wr}(H, p, v), p) &= v \\ \forall H, p, q, v. p \neq q &\Rightarrow \text{rd}(\text{wr}(H, p, v), q) = \text{rd}(H, q) \end{aligned}$$

The function  $\text{wr}(\dots)$  is used when a single heap location is updated. On the other hand, upon procedure call several locations, let us say  $a$  and  $b$ , need to be updated. This is expressed by introducing a fresh variable  $H_1$  and connecting it with the current heap, say  $H_0$ , using a *frame clause*, like:

$$\forall q. \text{rd}(H_0, q) = \text{rd}(H_1, q) \vee q = a \vee q = b$$

Spec# and VCC use *ownership* to organize objects in the heap, in particular with respect to framing. Each object has a distinguished field which stores the reference to the current owner of the object. The *ownership domain* of an object  $o$  is the set of objects from which  $o$  can be reached by following zero or more ownership links. If a procedure is allowed to write  $p$ , it can also write everything in the ownership domain of  $p$ . Because the reachability relation, used in definition of the ownership domain, is not expressible in first-order logic, we over-approximate the set of written locations to include all objects not directly owned by the current thread (denoted  $\text{me}$ ). Therefore, a frame clause for a procedure writing  $a$  and  $b$  in VCC would be:

$$\forall q, f. H_0[q, f] = H_1[q, f] \vee q = a \vee q = b \vee H_0[q, \text{owner}] \neq \text{me}$$

where  $H[p, f] \equiv \text{rd}(H, \text{field}(p, f))$ <sup>5</sup>, and the function  $\text{field}(p, f)$  gives the address of a field  $f$  in the object pointed to by  $p$ . Consequentially, for almost every  $H[p, f]$  access we will see, the term  $H[p, \text{owner}]$  being generated. Depending on the methodology, there might be more such artifacts, which together contribute a fair amount of complexity to heap reasoning.

**Chaining.** VCC uses backward chaining on frame clauses, i.e., they trigger on  $H_1[q, f]$ . Any heap access at  $H_k$  will be back-propagated to  $H_{k-1}$ ,  $H_{k-2}$  and so on. Alternatively triggering on  $H_0[q, f]$  would lead to forward chaining: accesses at the beginning of the function will be propagated toward the end. VCC requires backward chaining. For example let us consider a simplified version, of a verification condition, saying that writing 7 to a field  $\text{cnt}$  of some object preserves the invariant that all  $\text{cnt}$  fields are positive. Let  $I(H) \equiv (\forall q. H[q, \text{cnt}] > 0)$ .

$$\begin{aligned} (I(H_0) \wedge (\forall q, f. H_0[q, f] = H_1[q, f] \vee q = a) \wedge \\ H_1[a, \text{cnt}] = 7) \Rightarrow I(H_1) \end{aligned}$$

<sup>4</sup>The memory model designed for VCC [6] imposes a typed object model on top of C flat memory. Thus, the heap axiomatization is very similar to the one used for type-safe languages.

<sup>5</sup>We use the symbol  $\equiv$  to define a syntactic shortcut, which is expanded before the SMT solver sees it. This has triggering behavior different from introducing a function symbol and defining equivalence through an axiom. Boogie allows for easy switching between those two styles of function definitions on per-function basis.

To prove validity of that formula, the SMT solver will skolemize the universal quantifier from  $I(H_1)$ , generating an assumption  $\neg(H_1[q_0, \text{cnt}] > 0)$ .  $I(H_0)$  will only be applied on  $q_0$ , when the term  $H_0[q_0, \text{cnt}]$  is activated, which cannot happen, if the frame clause triggers only on  $H_0[q, f]$ .

**Multiple Heaps.** Some tools use multiple logical constants to encode the heap. For example in ESC/Java the split is done on per-field basis [23], while in Frama-C [21] the heap is further split based on syntactic aliasing analysis. This is clearly beneficial for the SMT solver, as no reasoning is necessary to infer that updates on different heaps commute. However, in case of VCC or Spec#, the benefits would be minimized because the frame clause of a procedure potentially needs to simulate write effects on all the partial heaps, because one does not know where objects from ownership domains might be stored.

### 2.2.1 The Good Heap

The verification methodology usually involves some protocols on accessing the heap. For example, one might model heap locations holding machine integers as mathematical, unbounded integers, but make sure a value outside the appropriate machine integer range is never stored in such a location. However, an axiom like:

$$\forall H, p. 0 \leq \text{rd}(H, p) \leq 2^{32} - 1$$

introduces an inconsistency, as in principle one could instantiate it with  $[H := \text{wr}(H_0, q, 2^{32}), p := q]$ . However, the point is that we are never going to create a heap like that. Moreover, because of triggering, the SMT solver is never going to instantiate the axiom with such a heap, which makes the unsoundness of such an axiom hard to detect. On the other hand, we do not want to rely on triggering for soundness, and therefore heaps obeying the verification methodology protocols are distinguished from other heaps with help of a predicate, let us call it  $\text{good\_heap}(H)$ . It is assumed for every “approved” heap, i.e., one introduced by the verification tool, and used as a precondition in axioms like the range axiom above. We never supply a definition for such a predicate, only axiomatize its consequences.

One can use several layers of such predicates (and e.g., Spec# and VCC do), depending on which of the system invariants hold. For example, after a heap update we know about integer ranges, but we do not know that invariants of all objects are preserved, before we check the invariant of the object just being updated.

## 2.3 The Liberal: Versioning

This section gives an example where the usual automatic trigger selection heuristics (used in Simplify, Z3 and CVC3) are too restrictive, and we need to introduce explicit triggers to make it more liberal.

In VCC, in a particular heap, an object can be either *open* or *closed*. In closed objects, only fields marked with the volatile modifier can change. The set of objects owned by an object is stored in a non-volatile field<sup>6</sup>, and consequently the set of object in an ownership domain, as well as their non-volatile fields, cannot change, as long as the root object is closed. Therefore, if an object  $o$  is closed in each of a sequence of consecutive heaps, we can infer that a value of

<sup>6</sup>This can be overridden by an explicit annotation, but for brevity we skip that possibility.

some field in the domain of  $o$  is the same in all of them. To avoid quantifying over sequences of states we use versioning: each object is equipped with a field carrying the version, and when the object is closed, this field is assigned a value encoding the values of all elements of the ownership domain. We do not specify the encoding explicitly, just axiomatize some of its properties.

Let  $\text{domain}(H, r) \equiv \text{ver\_domain}(H[r, \text{version}])$ <sup>7</sup>. The function  $\text{ver\_domain}(\dots)$  is uninterpreted, without any axioms attached to it. Its mere existence guarantees that the domain is part of the version encoding (cf. Section 2.1.1). The following axiom states that the encoding of the version also includes all non-volatile fields of objects in the domain:

$$\begin{aligned} \forall H, p, q, f. \{p \in \text{domain}(H, q), \text{rd}(H, \text{field}(p, f))\} \\ \neg \text{volatile}(f) \wedge p \in \text{domain}(H, q) \Rightarrow \\ \text{rd}(H, \text{field}(p, f)) = \text{fetch}(H[q, \text{version}], \text{field}(p, f)) \end{aligned}$$

Any value read from an object  $p$  known to reside in domain of  $q$  is considered to be a function of version of  $q$  ( $\text{fetch}(\dots)$  is similar to  $\text{ver\_domain}(\dots)$  in that sense), and hence if the version of  $q$  does not change, the value also does not change<sup>8</sup>. The explicit triggers on the axiom above, will cause it to be applied whenever a field of an object, known to reside in a domain is accessed. However, should we allow Simplify or Z3 to automatically choose the trigger, it would go for  $\text{fetch}(\text{rd}(H, q), \text{field}(p, f))$ , as this is the only single-term trigger possible. This is however fairly useless, because the only way to activate applications of  $\text{fetch}(\dots)$  is to apply this very axiom. This is thus an example where automatic trigger selection has not only performance implications, but also makes the axiom outright useless.

### 2.3.1 Filtering Trigger

The  $\text{fetch}(\dots)$  axiom above, with the explicit trigger, will be instantiated for volatile and non-volatile fields, and then the instances for volatile fields will be discarded per the implication precondition. To limit its applicability already at the instantiation level, we introduce a function symbol  $\text{non\_volatile}(\dots)$ , assume it for non-volatile fields and then add  $\text{non\_volatile}(f)$  to the trigger. If we use the function symbol  $\text{non\_volatile}(f)$  in the bodies of quantified formulas only when it is already placed in the trigger, no new instances of  $\text{non\_volatile}(\dots)$  will be created. Thus, the formulas will trigger only for non-volatile fields, for which we explicitly assumed the predicate. A similar pattern is used to supply different definitions of certain functions for primitive and non-primitive pointers.

If for some reason we want to avoid multi-triggers (for example because Simplify does not handle them very efficiently), one can use “the *as* trick” [23]: in addition to  $P(x) = \text{true}$ , we would assume  $\text{as\_}P(x) = x$ , thus allowing triggering on  $\{h(\text{as\_}P(x))\}$  instead of the multi-trigger  $\{h(x), P(x)\}$ . We have not used it in VCC in this particular place, but Section 2.4 uses similar trick in the definition of the  $\in'$  predicate.

<sup>7</sup>VCC stores the version of the object at the address of the object itself (i.e., we have  $\text{field}(p, \text{version}) = p$ ; because pointers in VCC include type information, no real field is actually stored there), so listing an object in the frame clause really means listing its version. This plays well with encoding of frame clauses.

<sup>8</sup>Following Spec# we use a similar trick for frame axioms of pure methods [8].

## 2.4 The Restrictive: Stratified Triggering

The following section demonstrates a case, where the automatic trigger selection will cause too many instantiations, i.e., we will need to restrict triggering.

Consider the following formula, being part of a simplified invariant of a doubly-linked list:

$$\begin{aligned} \text{inv}(H, l) \Leftrightarrow \dots \\ \wedge (\forall p. p \in \text{owns}(H, l) \Rightarrow H[p, \text{next}] \in \text{owns}(H, l) \wedge \\ H[p, \text{prev}] \in \text{owns}(H, l) \wedge \\ H[p, \text{data}] \neq \text{null}) \end{aligned}$$

The expression  $\text{owns}(H, l)$  refers to the set of objects owned by  $l$  in the heap  $H$ . If the trigger would be  $p \in \text{owns}(H, l)$  we would cause a *matching loop*: a term  $p_0 \in \text{owns}(H, l)$  will possibly activate the terms  $H[p_0, \text{prev}] \in \text{owns}(H, l)$  and  $H[p_0, \text{next}] \in \text{owns}(H, l)$ , each of which will in turn activate two more terms, and so on. Even if we limit instantiation depth to  $n$ , we still get  $2^n$  quantifier instances, and severe restrictions on  $n$  are unrealistic (see Section 3). Instead we split the formula into two recursive parts, triggering on the consequent of the implication, and a non-recursive part describing properties of a single list node:

$$\begin{aligned} \text{inv}(H, l) \Leftrightarrow \dots \\ \wedge (\forall p. \{H[p, \text{next}] \in \text{owns}(H, l)\} \\ p \in \text{owns}(H, l) \Rightarrow H[p, \text{next}] \in \text{owns}(H, l)) \\ \wedge (\forall p. \{H[p, \text{prev}] \in \text{owns}(H, l)\} \\ p \in \text{owns}(H, l) \Rightarrow H[p, \text{prev}] \in \text{owns}(H, l)) \\ \wedge (\forall p. \{p \in \text{owns}(H, l)\} \\ p \in \text{owns}(H, l) \Rightarrow \psi(H, l, p)) \end{aligned}$$

where  $\psi(H, l, p) \equiv (H[p, \text{data}] \neq \text{null})$ . This way we have removed the matching loop, but another problem remains.

For real trees and lists  $\psi$  is more complicated, and therefore we want to avoid  $\psi$  being instantiated too often. However, terms of the form  $p \in \text{owns}(H, l)$ , occur commonly and may have nothing to do with lists, e.g., might become active due to instantiation of definition of set operations or frame clauses. To address this problem, we introduce a function  $\text{as\_node}(\dots)$ , along with an axiom:

$$\forall p. \{\text{as\_node}(p)\} \text{as\_node}(p) = p$$

making it an identity and define  $p \in' S \equiv \text{as\_node}(p) \in S$ . So, by wrapping  $\text{as\_node}(\dots)$  around a term, we are essentially putting a special marker on it that can be later used in triggers. If we replace all occurrences of  $\in$  with  $\in'$  in the invariant<sup>9</sup>, both in triggers and formula bodies, we end up with much more restricted triggering behavior. The formula will trigger only for pointers  $p$  for which  $\text{as\_node}(p)$  was activated.

For example, the typical verification condition might look like  $\text{inv}(H_0, l) \wedge \Delta(H_0, H_1) \Rightarrow \text{inv}(H_1, l)$ , stating that the invariant of  $l$  is preserved by the state transition between heaps  $H_0$  and  $H_1$  (there might be intermediate heaps between them, but this is irrelevant here). When proving the last conjunct of the invariant, the SMT solver tests satisfiability of a formula  $\text{inv}(H_0, l) \wedge \Delta(H_0, H_1) \wedge \neg(\forall p. p \in \text{owns}(H, l) \Rightarrow \psi(H, l, p))$ . Assuming the bound variable  $p$  to be skolemized into  $p_0$ , the solver assumes  $p_0 \in' \text{owns}(H_1, l)$

<sup>9</sup>VCC provides a definition of the  $\in'$  predicate, so the user can choose to use  $\in'$  in invariants, and live with the consequences.

and  $\neg\psi(H_1, l, p_0)$ . In particular, the term `as_node(p0)` is activated. Thus, when the solver infers  $p_0 \in \text{owns}(H_0, l)$ , then  $\psi(H_0, l, p_0)$  follows, hopefully conflicting with  $\neg\psi(H_1, l, p_0)$ .

We have now limited the instantiations. In cases where the limitations are over-restrictive, e.g., the user needs a lemma  $\psi(H_0, l, n)$  for a specific list node  $n$ , then the user needs to introduce the marker `as_node(n)`, usually by adding an explicit assertion of the form  $n \in' \text{owns}(H_0, l)$ .

**Existential Activation.** In the previous example, it is possible that one needs to look one element forward in the list, to prove that the invariant of an arbitrary element is preserved, e.g., we might need  $\psi(H_0, l, H_1[p_0, \text{next}])$  in addition to  $\psi(H_0, l, p_0)$ . However, because  $p_0$  is a fresh constant, introduced by the SMT solver, the user cannot explicitly assert  $H_1[p_0, \text{next}] \in' \text{owns}(H_0, l)$ , which would be required to trigger it. Instead, the user can supply an annotation which states what terms should be activated when the formula undergoes skolemization:

$$(\forall p. \{p \in \text{owns}(H, l)\} \{\text{ex\_act}: H[p, \text{next}] \in' \text{owns}(H, l)\} \\ p \in \text{owns}(H, l) \Rightarrow \psi(H, l, p))$$

This pattern was crucial in verification of recursive data structures in VCC.

## 2.5 The Weird: Distributivity, Neutral Elements and Friends

This section talks about rather surprising behavior of a common set theory axiom. Similar axioms, causing similar problems, include pointer arithmetic normalization (for example,  $\&(\&p[i])[j] = \&p[i+j])$  and various distributivity axioms (for integer arithmetic, bit-vector arithmetic or set theory). The set theory example we are going to use is the following axiom describing the relation between set union and difference:

$$\forall A, B, C. \{(A \setminus B) \setminus C\} (A \setminus B) \setminus C = A \setminus (B \cup C)$$

Such an axiom may seem benign, but the number of applications resulting from a ground term  $(\dots((c \setminus d_0) \setminus d_1) \dots \setminus d_n)$  is exponential with  $n$  (we will get all possible parentheizations of the expression  $d_0 \cup \dots \cup d_n$ , and also some of its subexpressions). The number of instances can be reduced to quadratic by introducing another function symbol  $\hat{\setminus}$ :

$$\forall A, B. \{A \setminus B\} A \setminus B = A \hat{\setminus} B \\ \forall A, B, C. \{(A \hat{\setminus} B) \setminus C\} (A \hat{\setminus} B) \setminus C = A \hat{\setminus} (B \cup C)$$

Alternatively, we could trigger the original axiom on  $A \setminus (B \cup C)$ . However, should at some point the term  $c \setminus (\emptyset \cup d)$  arise, a matching loop would occur, provided the SMT solver would know  $d = \emptyset \cup d$ , but not  $c = c \setminus \emptyset$ , for example, because of axiom instantiation order or a missing axiom. The matching loop would involve instantiations where  $B = \emptyset, C = d$  and  $A$  is  $c, c \setminus \emptyset, (c \setminus \emptyset) \setminus \emptyset$  and so on.

The morale here, is that one needs to be careful when supplying such axioms that can be recursively applied, and make sure they do not loop or trigger excessively often. Luckily, such cases can be usually easily spotted when profiling the axiomatization (i.e., examining SMT solver log files containing list of instances produced during solver's run on a particular problem).

## 3. PERFORMANCE REQUIREMENTS ON THE SMT SOLVER

An important aspect of verification tools that is often overlooked is that they fail most of the time. This is inherent in the process of developing specifications: one tries different version of annotations (and possibly code) until the program finally goes through the verifier. This usually involves running the verifier every minute or so, after small changes or additions in annotations. Thus, usually we have a large number of unsuccessful runs of the verifier, and one successful run at the end. Therefore, in terms of SMT, the time to find a (probable) model is much more important than the time to prove unsatisfiability. This is particularly interesting, as it seems SMT with quantifiers currently lacks good stop conditions, short of waiting for all the matching possibilities to be exhausted. This is more of practical, rather than theoretical, problem because even if we were able to express the axiomatization in a fragment of logic with finite model property the size of formulas involved would likely make the theoretically finite models gigantic.

From the interactive standpoint, it would be ideal to have responsiveness in the range of a regular compiler, i.e., a couple of seconds. Experience shows that response times of over a minute are discouraging (or worse), and response times of over an hour definitely stop the development of annotations. Incrementality could be possibly exploited: the verifier is run several times with only slightly different versions of the VC. VCC currently does it manually: the user specifies which assertion they are interested in proving, and once they have proven them one-by-one, they can run the full verification, possibly on a build server (or multiple build servers).

As for the general scale of problems, the VCC background axiomatization includes about 300 quantified formulas, almost all with explicit triggers, including 50 with multi-triggers. While Hyper-V is structured into layers, most of its types are visible in most of the functions. There are about 300 types with 1500 fields, which after translation yields about 13000 axioms, half of them ground, consuming about 5 megabytes of SMT-format [22] file. Just the number of triggers involved exposed a couple of problems in Z3 E-matching indices. Because vast majority of Hyper-V functions are small, the background description of types and their invariants dwarfs the size of the translation of the function body itself. On the other hand, the function body is where the complexity lies: verification times vary between few seconds and hours, despite the fact that the background types are the same.

The number of quantifier instances when verifying a function is usually in the range of tens of thousands. Moreover, most of the axioms are never instantiated. However, interactions between the ones that are needed are quite complex. For example, we examined a proof of validity of a simple function inserting an element into a singly-linked list. While the example took less than 10 seconds to verify, the maximal required matching depth, i.e., the depth of the causal DAG for instances needed in the proof, was already 17. This function involved about 10 heap updates (most of which were ghost updates of the map abstracting the list and methodology bookkeeping). Thus, for every heap location accessed at the end of the function, one would need to apply 10 axioms to learn about the value of that location at the beginning of the function. 10 out of 17 instances in the chain were actu-

ally application of frame clauses. On the other hand, this chain also involved 6 different user defined formulas, coming from the invariant of the list. We conclude from this that any attempt at putting hard limits at instantiation depth are misguided.

## 4. DEBUGGING AND PROFILING AXIOMATIZATIONS

Analogously to ordinary programs, axiomatizations need to be debugged when the verification tool gives invalid answers and profiled when the tool takes too much time or memory. An invalid answer can be either due to unsoundness, when the tool proclaims a buggy program to be correct, or due to an incompleteness in the opposite case.

This section gives some insight on methods and tools implemented in VCC to aid in debugging and profiling axiomatizations.

### 4.1 Soundness

It is possible to develop axiomatization, where each formula presented to the SMT solver as an axiom is actually a theorem, which is valid in a model being a conservative extension build in a higher order logic proof assistant. In case of VCC there are however some practical reasons because of which such an effort was not undertaken. The two most important reasons are: a verification methodology in flux, developed alongside the axiomatization and the need to meet performance requirements on the behavior of Z3 with the axiomatization, which required frequent updates and a lot of experimentation. Analogous state of affairs persists in other, similar verification/bug-checking efforts [11, 2, 16].

The quality assurance in VCC is thus largely based on testing. As with most compilers, the size of the test suite corpus by far exceeds the size of the source of the compiler. The tests are both positive (i.e., benchmarks that are expected to verify) and negative (where a specific error is expected). The problem with negative benchmarks is that they are usually synthetic: simple code snippets constructed to show a specific verification error. The positive benchmarks can be also synthetic, but additionally larger code snippets are usually available showing somewhat more complicated use cases, combining together several features.

To partially compensate for that, VCC uses a reachability analysis [15], which checks if the SMT solver can find any unreachable program points (i.e., statements in the program, which the SMT solver can prove will never be executed). An unreachable program point, for the SMT solver, amounts to proving false at that point. Therefore, the analysis essentially reduces to testing several versions of the program, with `assert(false)` statements placed just before joints of the control flow graph. If the SMT solver finds such assertion to be valid, the location is unreachable. Such a result might point to code that is indeed unreachable (for example, stemming from a defensive programming technique), however more often than not it points to an inconsistent function preconditions or an inconsistent background axiomatization. Such a test is definitely not a silver bullet<sup>10</sup>, in particular it only detects immediate inconsistencies, not ones requiring non-trivial actions from the user to reproduce, or giving too strong guarantees about the programming language seman-

<sup>10</sup> In Boogie it is referred to as the *smoke test*, from turning a device on and seeing if the smoke goes out of it.

tics. Still, we found this analysis very valuable during VCC development.

Soundness problems are thus usually discovered through a failing test case. After initial efforts to minimize such test case, if the cause of unsoundness is still unknown, it is useful to know a small subset of axioms that are needed for the problem to manifest. Such a subset can be extracted from a full proof, if one can be produced by the SMT solver, but also from the UNSAT core<sup>11</sup>. Additionally, if neither of those options is available, one can just run smaller and smaller subsets of axioms through the SMT solver, using some automated procedure. On the other hand, we have not found the exact proof to be very useful, mainly due to its size and complexity.

### 4.2 Completeness

When analyzing a completeness problem (i.e., one where the verifier returns spurious errors), we try to pinpoint the cause of a failure by minimizing the test case, and possibly adding explicit assertions. If a problem is not apparent, we turn to a model in which the negation of a VC is satisfiable. The general technique is to evaluate the failed assertion in the model and try to trace the reasons why it fails. When doing so, we usually arrive at a point where our conceptual model of what should happen disagrees with the Z3 model. If the annotations are correct, the disagreement is because of a missing axiom or a wrong trigger. Such problems need to be solved by both the verification tool author, but also by the users, in case they use complex specifications.

#### 4.2.1 The Model Viewers

As the models can get quite large, VCC includes two tools for inspecting them. The VCC specific model viewer offers a debugger-like view of the counterexample, interlinked with the source code. One can trace pointers and inspect values of fields of data structures in different states. It thus hides a lot of axiomatization detail and is meant for the user of the verification tool as an aid in debugging their specifications.

The other tool is generic and can be used with different axiomatizations, mainly by the authors of the axiomatization or the SMT solver. It displays all active terms from the model and allows for inspecting equalities between them. For every term one can also see its immediate sub- and super-terms. For example, when looking at term  $rd(H_7, p)$ , one can see it is currently equal to 17, go to its subterm  $p$ , and then look at all other terms using  $p$ , e.g.,  $rd(H_6, p)$ ,  $rd(H_8, p)$  and  $wr(H_8, p, 3)$ , revealing value of pointer  $p$  in different states.

The two encoding patterns described below do not alter the logical value of the VC, and not even (significantly) the search tree of the SMT solver, but are used only to make the reason for failure more explicit in the model.

#### 4.2.2 Instantiation Traces in Models

To make sure a quantified formula was triggered, one can use *marker functions*. First, we axiomatize the marker function to be always true:

$$\forall x, y. \{mark_1(x, y)\} mark_1(x, y)$$

<sup>11</sup> If we treat an SMT problem as a big conjunction, then the UNSAT core is a (hopefully small) subset of the conjuncts, which are unsatisfiable.

Then, if we want to check if the following formula triggers:

$$\forall x, y. \psi(x, y)$$

we change it into the following (leaving trigger intact and preserving its truth value):

$$\forall x, y. \psi(x, y) \wedge \text{mark}_1(x, y)$$

This forces the model to include  $\text{mark}_1(t, s)$  for every  $t, s$ , for which the quantified formula is instantiated. Note that  $\forall x, y. \psi(x, y)$  might be used positively and negatively, for example as part of a type invariant, which is why we need to axiomatize the marker function to be always true.

Such marker functions can be used by the verifier author, but are also available for the end-user, if the language allows for specification function definitions, as it is the case for VCC.

### 4.2.3 Assignment Traces in Models

When looking at the model, it is often useful to inspect values of local variables at different points. Each source level assignment to a variable introduces, during VC generation, a new *incarnation* of that variable. To help keep track of different incarnations through the tool chain (e.g., to protect from copy-propagation), after the assignment we introduce an assumption, using the uninterpreted function symbol `local_is(...)`, for example the following source code:

```
void foo()
{
  int x;
  x = 10;
  x = x + 1;
  assert(x < 11);
}
```

will be transformed to:

```
void foo()
{
  assume(x1 == 10);
  assume(local_is(the_x, 4, 2, x1));
  assume(x2 == x1 + 1);
  assume(local_is(the_x, 5, 2, x2));
  assert(x2 < 11);
}
```

The parameters to `local_is(...)` are a unique symbolic constant, unused elsewhere, and thus immune from copy-propagation, the line and column where the assignment took place and the current value of the variable (i.e., a reference to the current incarnation). The model for this program will include `local_is(the_x, 4, 2, 10) = true` and `local_is(the_x, 5, 2, 11) = true`, revealing the value of `x` at different program points, and allowing the user to spot why the assertion fails. We use a similar technique to tie different heaps to source locations.

## 4.3 Performance Problems

Usually there is a tension between expressiveness and ease of annotation on one side and performance on the other side. Generally, proper triggering allows for mitigating some such tensions. A feature, bringing new expressive power, usually consists of new function symbols and axioms. If the axioms are triggered only by the new function symbols, parts of

the code not using the feature do not suffer performance problems. Using a build server infrastructure to track the time of executions of different parts of the test suite is a good way of making sure, that introduction of a new feature does not interfere with existing use cases.

Performance problems tend to manifest themselves in larger benchmarks, especially ones coming from users of the verification tool. An SMT solver usually allows one to gather some simple statistical data at the end of its run. In case of Z3 such statistics include the number of quantifier instances, conflicts, as well as various statistics from the arithmetic module (which is usually the only non-core theory of interest for VCC). Most of the performance problems we had were due to excessive quantifier instantiation, but some were due to inefficiencies in Z3, particularly in the arithmetic module. The statistics help to distinguish between those two cases: for example low number of instances per second (in case of Z3 and VCC less than about 10000) points to a problem different than quantifier instantiation.

### 4.3.1 The Causal DAG

In case of quantifier problems, one can instruct Z3 to save data concerning all quantifier instances made during the search to a log file. For example, given the formula:

$$\psi \equiv (\forall x. \{f(g(x))\} f(g(x)) \Rightarrow h(x, x))$$

if in the current model the terms  $f(d)$  and  $g(c)$  are active and  $d = g(c)$ , then Z3 creates an instantiation tautology  $\psi \Rightarrow \psi_c$ , where:

$$\psi_c \equiv f(g(c)) \Rightarrow h(c, c)$$

After that, the following tuple is added to the log file:

$$L_c \equiv \langle \psi, [x := c], \{f(d), g(c)\}, \{h(c, c)\} \rangle$$

The tuple lists the reference to the quantified formula, the substitution, the active terms that caused the match, and the outcome of the instantiation (i.e., terms that were created in course of instantiation; the particular log entry above assumes this is the first time in the current search branch where  $h(c, c)$  was needed).

Later, in the same search branch, if an instantiation tautology  $\phi \Rightarrow \phi_t$  is generated, where its log entry is:

$$L_t \equiv \langle \phi, \dots, \{\dots, h(c, c), \dots\}, \dots \rangle$$

we call  $L_c$  a *cause* of  $L_t$ . The directed graph formed by instances with the causal relation (where the edge goes from  $L_c$  to  $L_t$ ) is acyclic (instances can only cause other instances later in time). We refer to it as the *causal DAG* (directed acyclic graph). This information might be imprecise (e.g., the instance where a term first appeared does not necessarily have to be the one that caused the term to be activated; the matching algorithm will not log terms from proofs of required equivalence class merges and so on). Still, we have found it invaluable in tracing performance problems.

By the nature of performance problems, the number of instances involved tends to be rather large (up to millions). We have thus implemented a tool for analyzing instantiation log files, called the axiom profiler. It allows one to trace a particular instance through the causal DAG, while also inspecting various summaries. One summary is the cost of an instance, which is the estimated number of instances caused by it. More precisely, the cost of an instance node  $n$  in the

causal DAG  $E$  (where  $(n, m) \in E$  means  $n$  causes  $m$ ) is given by  $c(n)$ :

$$c(n) = 1 + \sum_{(n,m) \in E} \frac{c(m)}{\text{indeg}(m)}$$

where  $\text{indeg}(n) = |\{m \mid (m, n) \in E\}|$ , is the number of incoming edges of  $n$ . In other words, the cost of an instance is shared equally by all instances that caused it. The other summary is the depth of the instance (which is also computed by Z3 during runtime), which is a maximal sum of weights of nodes on the path from a DAG root to the instance. The weight is user-defined and defaults to 1. We view weights as a promising future direction in restricting solver search space, but do not have much experience with them yet.

### 4.3.2 The Conflict Tree

Another piece of information that the Z3 log file provides is the *conflict tree*. Consider the assignment stack in the SMT solver. First a number of literals is pushed, and a conflict is found. We can think of this chain of literals as one long branch of a tree. Then backtracking occurs, taking us somewhere up the chain, and, starting from there, new sequence of assignments happens, creating another branch. Backtracking always ends just below decision literals, thus by analysing the sequence of literal assignments, decisions and backtrack operations, we can reconstruct a tree, where internal nodes are labelled by literals, the tree is branching at decisions, and the leaves are labelled by conflicts.

Because the information about literal assignments and quantifier instantiation is interleaved in the log file, the costly places in the search (ones where lots of quantifier instances are generated) can be identified. This information can be later attributed to conflicts, i.e., one can see how many quantifier instances and literal assignments it took to discover a particular conflict. Therefore, one can pick a costly conflict at random and try to see if it “makes sense”, i.e., if one expects such conflict to contribute to the final outcome and if it really should be that hard to find it. Similarly, one can find an instantiation at random and see if it “makes sense”. This is the most effective way of finding performance problems we have found. Clearly, this requires an intimate understanding of the inner workings of axiomatization, as well as some understanding of the SMT solver search algorithms, which is why we view the axiom profiler as a tool for the author of a verification tool, and not for the end-users.

### 4.3.3 Z3 Inspector

In Section 3 we have argued how important it is for the SMT solver to return the “satisfiable” answer fast. However, the ideal time constraints on such responses (in range of few seconds) are very hard to meet with the current combination of hardware speed, SMT solver performance and VC complexity. As a partial remedy, Boogie offers its users a way of inspecting the progress of Z3 on the current VC. Unlike in other implementations [12] this is achieved without splitting the VC into subformulas before sending them one-by-one to the SMT solver. Instead, we instruct Z3 to simulate a failure at pre-defined time intervals, and use the standard error-reporting mechanism. This standard mechanism relies on “labels” [18], which describe the part of formula, which is satisfied by the current monome, and thus also the assertion, negation of which is satisfiable in the model of the monome.

The labels also encode the execution path that lead to the failing assertion.

Boogie, which is normally responsible for translation of labels to error messages, captures continuous label output of Z3 and passes it, along with error translation information, to the Z3 Inspector tool. The Inspector displays the source code of the program, from which the VC was generated, where each line is annotated with possible errors that could be reported for this line, should one of the obligations in the VC fail to prove. As the SMT solver works through the formula, the error message corresponding to the current monome is highlighted.

This works particularly well for the case split strategy in Z3, which tries to satisfy the formula left-to-right. This means Z3 will try to prove assertions top-down and, which is more important, will only move to the next assertion, after the previous one is proven. Translated to Z3 Inspector terms the assertion will blink, one by one, for certain amount of time. If one assertion blinks for a long time, it means the prover is trying to hard to refine a model for the case of this assertion failing. The user of the verification tool can then decide, based on time it took to prove this or similar assertion before, to consider this particular assertion to be faulty and kill the verification process.

The cumulative number of samples, pointing to the particular error message, is also displayed, allowing for analysis of which proof obligation was particularly costly, also after the search is done. One could imagine the conflict clauses being communicated out of Z3, so Z3 Inspector could mark the errors that are already guaranteed not to happen (i.e., we know that there is not model that would satisfy the negation of their assertions). This might be useful with a more random case split strategies.

In general the Z3 Inspector provides progress information about the verification process, which is accessible and meaningful for the end-user. It was helpful in reducing user frustration with the verification process by providing some basic control over it.

## 5. CONCLUSION

We have presented some of the triggering patterns used in software verification, as well as development practices for axiomatizations, in hope of establishing the requirements for a possible alternative to E-matching based quantifier instantiation methods. Following the presented operational view, where triggering is a way of programming the SMT solver, we envision such an alternative to admit high degree of control, e.g., a domain specific language based on term rewriting systems. More automatic techniques, e.g., superposition, could be useful in subproblems involving formulas from the program specification, as such formulas are less likely to contain useful guidance to the SMT solver, than the ones from the background axiomatization.

Given some help from the SMT solver side, some of the presented encoding patterns might be expressed more directly, allowing the SMT solver to treat them more efficiently. For example, the “*as* trick”, of assuming  $f(c) = c$ , just to be able to use  $f(x)$  in a trigger, is a method for controlling when a certain axiom should be applied, and the function  $f(\dots)$  does not have much logical meaning, just polluting the data-structures of an SMT solver. If a native SMT construct for labelling terms and axioms would be provided, one would not need to do so. Clearly each

such feature would need to have its possible performance benefits weighted against complications in the SMT solver implementation and input language.

**Benchmarks.** SMT benchmarks exercising techniques described in this chapter are available in the UFNIA (and AUFLIA with older benchmarks) division of the SMT LIB. Current VCC benchmarks will soon be submitted to the UFNIA division, pending resolution of some known soundness issues.

**Acknowledgements.** The author wishes to thank Rustan Leino, Nikolaj Bjørner and Stephan Tobies for their insightful comments and suggestions regarding this paper, Leonardo de Moura for his help in getting the ideas to work with Z3, and Lieven Desmet for his contributions to the early versions of the axiom profiler.

## 6. REFERENCES

- [1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, Sept. 2006.
- [2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.
- [3] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Better avionics software reliability by code verification – A glance at code verification methodology in the Verisoft XT project. In *Embedded World 2009 Conference*, Nuremberg, Germany, Mar. 2009. Franzis Verlag. To appear.
- [4] S. Böhme, M. Moskal, W. Schulte, and B. Wolff. HOL-Boogie: An interactive prover-backend for the Verifying C Compiler. *Journal of Automated Reasoning*, 2009. To appear.
- [5] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer. Invited paper, to appear.
- [6] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A Precise Yet Efficient Memory Model For C. In *Proceedings of Systems Software Verification Workshop (SSV 2009)*, 2009. To appear.
- [7] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, Feb. 2009. Available from <http://research.microsoft.com/pubs>.
- [8] Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In M. B. Dwyer and A. Lopes, editors, *FASE*, volume 4422 of *Lecture Notes in Computer Science*, pages 336–351. Springer, 2007.
- [9] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.
- [10] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
- [11] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec. 1998.
- [12] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003.
- [13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, May 2002.
- [14] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In F. Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 167–182. Springer, 2007.
- [15] M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 23–30. ACM, Sept. 2007.
- [16] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In G. C. Necula and P. Wadler, editors, *POPL*, pages 171–182. ACM, 2008.
- [17] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
- [18] K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3):209–226, 2005.
- [19] K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 615–622. ACM, 2009.
- [20] M. Moskal, J. Lopuszański, and J. R. Kiniry. E-matching for fun and profit. *Electr. Notes Theor. Comput. Sci.*, 198(2):19–35, 2008.
- [21] Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, Jan. 2009.
- [22] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at [www.SMT-LIB.org](http://www.SMT-LIB.org).
- [23] J. Saxe and K. R. M. Leino. ESC/Java design note 8a: The logic of ESC/Java, 1999. Available at <http://secure.ucd.ie/products/opensource/ESCJava2/ESCtools/docs/design-notes/escj08a.html>.