Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2

Thomas Ball[1], Ella Bounimova[1], Vladimir Levin[2], and Leonardo de Moura[1]

[1] Microsoft Research Redmond, USA
[2] Microsoft Redmond, USA

March 2010
MSR-TR-2010-24
Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Abstract. Static Driver Verifier (SDV) is a verification tool included in the Windows 7 Driver Kit (WDK). SDV uses SLAM as the program analysis engine. SDV 2.0 released with Windows 7 uses a re-designed SLAM2 engine. SLAM2 improves the precision and performance of predicate evaluation by using Z3 SMT solver. To handle predicates with pointers in SLAM2, we propose a novel set of axioms that defines a logical memory model, which is one of the underlying concepts and limitations of SLAM. We also designed an algorithm of encoding predicates passed to Z3 with uninterpreted functions over integers. In this paper, we present the axioms and the encoding. We also show how the axioms can be modified to achieve a better precision by refining the memory model. Our profiling of SDV runs on real device drivers confirms that the axioms and the encoding allowed SLAM2 to achieve a good balance between the precision required by the logical memory model, and Z3 performance on complex predicates. Our presentation of the axioms and the encoding in this paper is decoupled from SLAM2, such that they could be utilized by other static analysis tools when dealing with pointer predicates - often a bottleneck in such tools.

1 Introduction

Static Driver Verifier [BBC+06] (SDV) is a verification tool included in the Windows 7 Driver Kit (WDK). SDV uses SLAM [BR01] as the program analysis engine. SDV 2.0 released with Windows 7 uses a re-designed SLAM2 engine. SLAM2 improves the precision and performance of predicate evaluation by using Z3 SMT solver [MB08]. In particular, we take advantage of the Z3 feature to provide a pre-defined set of axioms as a context for evaluation. Predicates that are given to Z3 by SLAM2 in runs on Windows Device Drivers can be long and

may contain disjunctions, which could be a problem for SMT solvers. Because of that, care has been taken to optimize calls to Z3 as much as possible. In particular, for predicates involving pointers, we propose a novel set of axioms that defines a *logical* memory model, which is one of the underlying concepts and limitations of SLAM. For example, in SLAM, all array elements are merged into one (first) element, pointer arithmetic is abstracted away, etc. Our axioms postulate basic relations between pointers and locations in the SLAM memory model. We also designed an algorithm of encoding predicates passed to Z3 with uninterpreted functions over integers. For comparison, other static analysis tools - for example, HAVOC [LQ08] - use a *physical* memory model, based on select-update axioms, which may significantly degrade SMT solver performance.

Our profiling of SLAM2 runs on real Windows Device Drivers confirms efficiency of the axioms and the encoding: Z3 was never found to be a bottleneck in those runs when the tool run out of resources (time or memory). On the other hand, we have evidence that precision of SLAM2 predicate evaluation is better than that of SLAM1.

2 Definitions

In the presence of pointers, predicate evaluation should take into account relations between pointer variables and memory objects, or *locations*. In a predicate on program variables that we pass to Z3, locations are represented as simple variables, (for example, x, y, z), and terms. A term can be a direct and indirect field access (for example, $x.f, y{->}g$), array element access (for example, $x[0], x[i]$), or dereference (for example, $*x, *(x.f)$).

We define locations as elements of an abstract domain L. Each location X has a unique address represented by a function from locations to integers: $A : L{->} INT$. We represent a value stored in a location as a function $V : L{->}V$, where V is an abstract domain for *values*. The location domain L is partitioned into two disjoint sets: *normal* locations L_1 and *abnormal* locations L_2. The domain of normal locations is partitioned into three disjoint sets: *basic* locations L_B, *field* locations L_F and *implicit* locations L_I.

A variable from a predicate denotes a basic location. An access term (field access or an array element access) denotes a field location. A dereference could denote a location which is also denoted by another variable or an access term: for example, in the context of predicate $x == *y$, variable x and dereference $*y$ denote the same basic location. In such case, we say that this dereference is *aliased* with the corresponding variable or access term. A dereference that cannot be aliased with a variable or an access term is thought of as denoting an implicit location - in other words, implicit location can only be referenced through dereferences.

A predicate may contain dereference terms that cannot yield any normal location. For example, dereference $*p$ in the predicate $(p == 0 \wedge *p == N)$ cannot denote any normal location that would make this predicate satisfiable, for any value of N. To deal with such predicates, we introduce abnormal locations.

Note that detection of an abnormal location could signal a bug in the program. For example, the predicate above could represent an attempt of an execution of the following incorrect C code: $\{p = 0; *p = N; \}$.

We combine structure and array locations into a concept of an *aggregate* location, by replacing field names with integers from an interval $[0\,..\,N-1]$ [3], where N is the number of fields in the structure, preserving the order of the fields. Access terms of the form $x[f]$ are replaced with access terms of the form $x.f$, where f is a non-negative integer, and both field names and constant indices of arrays are referred to as *field indices*.

In the first set of axioms below, we do not consider predicates with access terms $x[n]$, where n is an expression other than an integer constant. In section 6.4, we show how to handle a more general case, where n is an arbitrary expression.

Given a program, we consider a finite interval $[0\,..\,N-1]$ of integers that represent all field indices in this program. Here, N is the number of fields (including array elements) in the largest (with respect to the total number of fields) aggregate defined in the program. Intuitively, we consider aggregate locations as graph structures, with parent locations (for example, a whole aggregate location) connected to their child locations (for example, top-level fields) via one of the two possible links: field indices or dereference links. Formally, we introduce a location constructor function $S(X, C)$, with X being a (parent) location and C being a connection link from X to the (child) location $S(X, C)$. A connection link C is either a field index (an integer from interval $[0\,..\,N-1]$), or dereference link D. The dereference link D is introduced as a unique abstract object to represent a pointer dereference location $*X$ as $S(X, D)$. Thus, constructor $S(X, C)$ is polymorphic on its second argument: the entire domain of links is the union of two sets: $[0\,..\,N-1] \cup D$; we require that $D \notin [0\,..\,N-1]$.

All non-basic locations, both normal and abnormal, are generated by applying location constructor S to basic locations from the set L_B. The set of the basic axioms $I.a$ below defines valid terms generated by this constructor, at the same time defining dependencies between this constructor and the address and value functions (A and V, introduced earlier).

Below is the core *abstract* version of the axioms. We call it "abstract" to distinguish it from the *implementation* version in I.b. In the axioms below we use capital letters F, G, X, Y to refer to semantic objects: field indices F, G and locations X, Y.

3 Axioms I.a: the core version

In the axioms below, X and Y are abstract locations from the domain L, D stands for integer -1 (dereference link encoding), the integer interval $[0\,..\,N-1]$

[3] We follow here C convention for array indices; for languages with a different convention, indices would have to be converted into the C-style indices.

encodes field indices.

1. $\forall X \in L_N : \quad A(X) > 0$
2. $\forall X, Y : \quad A(X) = A(Y) \implies X = Y$
3. $\forall X, F \in [0 \,..\, N-1] : \quad S(X, F) \notin L_B$
4. $\forall X : \quad A(S(X, D)) = V(X)$
5. $\forall X, Y, F \in [0 \,..\, N-1], G \in [0 \,..\, N-1] :$
   $S(X, F) = S(Y, G) \implies X = Y \wedge F = G$
6. $\forall X, Y, F \in [0 \,..\, N-1] : \quad V(X) = V(Y) \implies V(S(X, F)) = V(S(Y, F))$

Axiom 1 requires that normal locations have meaningful addresses. Axiom 2 states that each location is uniquely identified by location's address. In other words, axiom 2 requires for function A to be injective. Axiom 3 requires that constructor S generates only non-basic locations. Axiom 4 states that pointer value is equal to the address of the location obtained by dereferencing the pointer. From axioms 4 and 1, it follows that if the value of a pointer X is 0, the address of the location $S(X, D)$ is also 0, which means that this is an abnormal location (axiom 1). Axiom 5 states that two field locations are identical only if they share the same parent location and the same link associated with them. Axiom 6 is a counterpart to axiom 5. It states that if two parent aggregate locations have equal values, then the corresponding child locations that have the same link associated with them also have equal values.

Next, we give the implementation version of the abstract axioms. In the implementation version, all semantic objects - locations, links, addresses and values - are encoded with integers and uninterpreted functions on integers. It is the implementation version of the axioms that we give to Z3 SMT solver. When we translate a predicate into the form for Z3, we encode basic locations explicitly. All other locations are implicitly encoded by the constructor $S(X, F)$. All normal locations are encoded with positive integers. Since the dereference link D is encoded with -1, all the links belong to $[-1 \,..\, N-1]$. Below, we define translation rules for the conversion of a predicate into the encoded form.

4 Translation rules

We are using small letters f, v, x to refer to syntactic objects: fields, variables, and locations terms, respectively. Below, x' stands for the result of translation for the term x by applying the translation rules (recursively); f' denotes encoding for the field index f, B is the maximum number of basic locations on the program.

Given a predicate, replace the location terms in this predicate as defined by the following rules:

```
1. Basic location encoding: each variable v is replaced with
  a unique positive integer n, n < B.
2. Dereference *x is replaced with S(x',D).
3. Direct field access x.f is replaced with S(x',f'),
   where f' is a non-negative integer n, n < N.
4. Indirect field access x->f is replaced with S(S(x',D),f').
```

5 Axioms I.b: the core implementation version

In the implementation version of the axioms, X and Y are integers from the interval 1 through B that encode abstract locations in L_B, as defined in the translation rules above. All other normal locations are thought of as implicitly encoded with integers larger than B; abnormal locations are thought of as implicitly encoded with non-positive integers; values are thought of as implicitly encoded with any integers.

The implementation version of axioms I.a with the encoding specified above is as follows:

1. $\forall X > 0 : \quad A(X) > 0$
2. $\forall X, Y : \quad A(X) = A(Y) \implies X = Y$
3. $\forall X, F \geq 0 : \quad S(X, F) > B$
4. $\forall X : \quad A(S(X, D)) = V(X)$
5. $\forall X, Y, F \geq 0, G \geq 0 :$
   $S(X, F) = S(Y, G) \implies X = Y \wedge F = G$
6. $\forall X, Y, F \geq 0 : \quad V(X) = V(Y) \implies V(S(X, F)) = V(S(Y, F))$

6 Modifications of core axioms

In this section, we show how to modify Axioms I.a and I.b. There are two goals that we want to achieve by the modifications: either to optimize Z3 evaluation (section 6.1), or to make the axioms more precise, i.e., to add some elements of the physical memory model (axioms 6.2 - 6.4). For the sake of simplicity, we only give modifications for the implementation version I.b. However, the same modifications can be done for the abstract version I.a.

6.1 Axioms II: optimized version

We can improve Z3 performance by modifying axioms 2 and 5 - which significantly contribute to the evaluation complexity - by introducing inverse functions for functions A and S. A^{-1} is the inverse function for A, and for S, S^{-1} and S^{-2} are the inverse functions over the first and second argument (excluding the dereference link), respectively. Then we can re-write axiom 2 as axiom 2.a, and axiom 5 as two axioms 5.a and 5.b:

$2a. \quad \forall X : \quad A^{-1}(A(X)) = X$
$5a. \quad \forall X, F \geq 0 : \quad S^{-1}(S(X, F)) = X$
$5b. \quad \forall X, F \geq 0 : \quad S^{-2}(S(X, F)) = F$

6.2 Axioms III: merging address of aggregate with the address of its first field

The set of axioms I.a (or I.b) is sound for the evaluation of a predicate that does not take into account the following C language conventions about physical

memory allocation: (i) address of the first field of a structure is the same as the address of the entire structure; (ii) address of the first element of an array is the same as the address of the entire array. Modification in this section accounts for these conventions, thus extending the applicability of the axioms I. Note that modification II is not applied in this case.

Axioms I.b are modified as follows: axiom 2 is replaced with the two axioms: 2a and 2b:

$$2a. \quad \forall X > B: \quad A(X) = A(S(X,0))$$
$$2b. \quad \forall X > B, Y: \quad Y \neq S(X,0) \implies (A(X) = A(Y) \implies X = Y)$$

Axiom 2.b above states that axiom 2 from I.b holds only for those locations where one of the locations is not a first element of another location.

6.3 Axioms IV: physical memory model, no unions

According to the C language, data type of a field (or an array element) can be used to determine address of the next field (array element) in the structure (array). Additionally, memory alignment should be taken into account in some cases. Let's consider a function $T(X, F)$ that defines those rules, where X is an aggregate location, F is a field index, and return value is the address shift between the field F and the next field $F + 1$. If we are given the F function for a particular implementation, we can modify axioms I.b by adding the following axiom:

$$7. \quad \forall F, 0 \leq F < N - 1: \quad A(X, F + 1) = A(X, F) + T(X, F)$$

6.4 Axioms V: locations as field indices

To handle array elements with variable indices, we modify both the translation rules and the axioms. The modification will be shown for the impementation version of the axioms and rules, but could be extended on the abstract version as well.

First, we add one new translation rule to the set defined in section 4. Consider a term $x[k]$, where k is a variable or a location term. If k is a variable, k' is an integer that encodes k as a basic location; if k is a location term, k' is translation of k as defined by the (modified) translation rules. Then, $x[k]$ is translated into $S(x', V(n'))$. Here, we also assume that integers belong to the interval $[0..N–1]$. Second, the axioms are modified simply by adding a new axiom to constrain results of the function V, which we need to distinguish encoding of the dereference link D from the location values (which can now be used as field indices):

$$\forall X > 0: \quad V(X) \neq D$$

7 Conclusion

To evaluate predicates with pointers with Z3 in SLAM2, we designed a novel set of axioms that defines a logical memory model used by SLAM, and an encoding of the predicates passed to Z3 based on uninterpreted functions on integers. We showed how the axioms can be modified to incorporate more details into the memory model and to achieve a better precision. Our profiling of SDV runs on real device drivers confirms that the axioms and the encoding allowed SLAM2 to achieve a good balance between the precision required by the SLAM logical memory model, and Z3 performance on complex predicates. For SDV 2.0, the true bugs/total bugs ratio is 90-98% on Windows 7 Microsoft drivers, depending on the class of driver. The number of non-useful results (timeouts, "don't know" results) has been reduced greatly. In particular, for drivers shipped as WDK samples, it is 3.5% for WDM drivers and 0.02% for KMDF drivers. We believe that the axioms and the encoding presented in this paper could be utilized by other static analysis tools when dealing with pointer predicates - often a bottleneck in such tools.

References

[BBC+06] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 73–85, 2006.

[BR01] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *CAV*, pages 260–264, 2001.

[LQ08] S. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using smt solvers. In *Principles of Programming Languages (POPL '08)*, January 2008.

[MB08] Leonardo De Moura and Nikolaj Bjrner. Z3: An efficient smt solver. In *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.