Inferring Annotations For Device Drivers From Verification Histories

Zvonimir Pavlinovic[1], Akash Lal[2], and Rahul Sharma[3]

[1] New York University, USA, zvonimir@cs.nyu.edu
[2] Microsoft Research, India, akashl@microsoft.com
[3] Stanford, USA, sharmar@cs.stanford.edu

Abstract. Finding invariants is an important step in automated program analysis. Discovery of precise invariants, however, can be very difficult in practice. The problem can be simplified if one has access to a candidate set of predicates (or *annotations*) and the search for invariants is limited over the space defined by these annotations.
We present an approach that infers program annotations automatically by leveraging the history of verifying related programs. Our algorithm extracts high-quality annotations from previous verification attempts, and then applies them for verifying new programs. We present a case study where we applied our techniques to Microsoft's Static Driver Verifier (SDV). SDV currently uses manually-tuned heuristics for obtaining a set of annotations. Our techniques can not only replace the need for this manual effort, they even out-perform these heuristics and improve the performance of SDV overall.

1 Introduction

The performance of program verifiers depends on the discovery of precise invariants or procedure pre/post conditions. The task of finding invariants is often broken down into finding a set of candidate facts, or annotations, and then using these facts to establish inductive invariants. For example, predicate-abstraction-based tools such as SLAM [3] or BLAST [12] rely on predicate discovery. Tools such as UFO [1] and Duality [19] rely on interpolation to generate candidates for procedure summaries. Each of these techniques, however, infer annotations by analyzing only the program to be verified.

We propose a complementary approach of inferring program annotations automatically by exploiting information available from prior verification runs. We build on the insight that annotations useful for verifying a particular program are often already observed earlier during the verification of related programs. For instance, programs that use the same API probably require similar annotations for verifying contracts of that API. We keep track of the verification history by accumulating a set of programs and the annotations required to construct their respective proofs. We leverage this history to generate a *small* set of annotations that are *useful* for subsequent (unseen) programs.

There are two key challenges in making this approach work. First, annotations are formulas over program variables, thus, tied to program-specific variable names. We abstract away from program-specific names by working with *abstract annotations*, which are arbitrary formulas with *holes*. Abstract annotations are *concretized* to a program by filling the holes with the program's variables.

The set of all (abstract) annotations in the verification history has the nice property that it is sufficient to establish the correctness of all programs observed in the history. However, this set is likely to be very large, making the verifier spend a significant amount of time just discarding invalid annotations. Our second challenge is to keep the set of inferred annotations small. We design a *minimization* algorithm that computes a set of abstract annotations such that: (1) it is enough to establish correctness proofs of all programs in the history, and (2) no smaller subset (or *syntactically simpler* set, in a sense that we formalize later) is enough to establish all correctness proofs.

Our primary motivation behind these ideas is to improve the performance of Microsoft's Static Driver Verifier (SDV) [2]. SDV is an industrial-strength tool for formal verification of Windows device drivers. SDV checks that drivers conform to certain properties (called *rules*) that establish correct usage of the kernel API. SDV currently uses manually-tuned heuristics for obtaining a set of annotations that are passed to a program verifier. Using a repository of small in-house drivers, our techniques can not only replace the need for this manual effort, they even out-perform these heuristics and improve the performance of SDV overall.

We summarize our contributions as follows:

- We present an algorithm for inferring an optimal set of abstract annotations from past verification efforts.
- We apply the algorithm to SDV and experimentally show that inferring annotations from past verification efforts can potentially generate better annotations than ones provided by a human expert. The set of abstract annotations inferred by our algorithm can improve the verification times by 22% on average and reduce inconclusive results by 47%.

2 Overview

This section illustrates our framework for learning useful annotations. If `f` is a procedure and ϕ is a formula, we use $[\phi]$@`f` to denote that ϕ is a valid postcondition of `f`. If `f` has a loop starting at location `L` then the notation $[\phi]$@`f`@`L` denotes that ϕ is a valid loop invariant for `L`. We model assertion failures as setting of a special `ok` bit to *false*. For a global variable `x`, let `old(x)` refer to the value of `x` at the beginning of the procedure or loop, depending on the context in which it is used. For instance, $[(\texttt{x} == \texttt{old(x)} + 1) \vee \neg\texttt{ok}]$@`f` means that the execution of `f` either increments the value of `x` or it fails an assertion. In other words, if `f` doesn't fail then it increments `x`. $[\texttt{x} == \texttt{old(x)}]$@`f`@`L` means that the loop at location `L` of procedure `f` preserves the value of `x` across

```
var depth: int;

procedure init() {
  depth := 0;
}

procedure Acquire() {
  depth := depth + 1;
}

procedure Release() {
  depth := depth - 1;
}

procedure d_exit()
{
  assert depth == 0;
}
```

(a)

```
procedure P_n()
{ call init(); call dispatchP_n(); }

procedure dispatchP_n() {
    [call Acquire()]^n;
L1: while(*)
    { call Acquire(); call Release(); }
    [call Release()]^n; call d_exit();
}

procedure Q()
{ call init(); call dispatchQ(); }

procedure dispatchQ() {
L2: while(*)
    { call Acquire(); call Release();
      call dispatchQ();  }
    call d_exit();
}
```

(b)

Fig. 1. (a) An Acquire-Release API and (b) a family of programs exercising the API

arbitrary number of loop iterations. A *proof* of correctness of a program is simply a sequence of mutually-inductive postconditions of procedures or loops in the program that imply ok==true at the end of the program. For simplicity (and without loss of generality) we do not talk about procedure preconditions in this paper.

Figure 1 shows a family of programs that exercise the same API. The API and its contract is shown in Figure 1(a). The API models acquiring and releasing a resource. The variable depth keeps track of the number of resources held. For correct usage of the API, at certain points (when dispatch routines return) the number of resources held is asserted to be zero. Entrypoints of programs that exercise this API are $\mathtt{P}_n$ and Q. $\mathtt{P}_n$ is parameterized by the value of n. It calls the routine $\mathtt{dispatchP}_n$ where we use the notation $[\mathtt{st}]^n$ to denote n occurrences of the statement st.

Annotations and Proofs Figure 2 shows possible proofs for the programs of Figure 1. Note that all postconditions in proof A of $\mathtt{P}_n$ do not depend on the value of n, whereas proof B is specific to the value of n. *Annotations* are simply formulas that serve as candidates for postconditions. Generation of invariants from a given set of annotations (which we call *annotation-based invariant generation*) is much simpler than full-blown verification, often even decidable. One may use, for example, predicate abstraction [4] to construct invariants that are Boolean combinations of the given annotations. In our work, we use the Houdini algorithm [8] to find conjunctive invariants: ones that are conjunctions of

// *definitions*
$\psi_i \equiv \texttt{depth} - \texttt{old(depth)} == i$
$\sigma_n \equiv \texttt{old(depth)} == n \Rightarrow \texttt{depth} == n$
$\eta \equiv (\texttt{old(depth)} == 0 \Rightarrow \texttt{ok})$
// *Proof A of* $\texttt{P}_n$
[depth == 0]@init, $[\psi_1]$@Acquire, $[\psi_{-1}]$@Release, $[\eta]$@d_exit, $[\psi_0]$@dispatchP$_n$@L1, $[\eta]$@dispatchP$_n$
// *Proof B of* $\texttt{P}_n$
[depth == 0]@init, $[\psi_1]$@Acquire, $[\psi_{-1}]$@Release, $[\eta]$@d_exit, $[\sigma_n]$@dispatchP$_n$@L1, $[\eta]$@dispatchP$_n$
// *Proof of Q*
[depth == 0]@init, $[\psi_1]$@Acquire, $[\psi_{-1}]$@Release, $[\eta]$@d_exit, $[\psi_0 \wedge \eta]$@dispatchQ@L2, $[\psi_0 \wedge \eta]$@dispatchQ

Fig. 2. Possible proofs of correctness of the programs in Figure 1

some subset of the given annotations. This problem has a lower complexity than predicate abstraction and is very fast in practice for small to medium number of annotations. For example, given annotations $\{\texttt{depth == 0}, \psi_{-1}, \psi_0, \psi_1, \eta\}$ it is very efficient to re-construct a proof for $\texttt{P}_n$ using Houdini.

Minimal Repositories Our technique requires a repository of programs and their proof of correctness. The proofs may be constructed manually or by using proof-generating verifiers. We do not expect to control the proof-generation process. Suppose we have programs Q and $\texttt{P}_n$ for each $n \in N$, for some large set N, in our repository. Further, suppose we have proof of type B (Figure 2) for $\texttt{P}_n$ for all values of n, except n_0, and $\texttt{P}_{n_0}$ has a proof of type A.

This set of proofs produces a large number of annotations $\mathcal{A} = \{\texttt{depth == 0}, \psi_{-1}, \psi_0, \psi_1, \eta, \psi_0 \wedge \eta\} \cup \{\sigma_n \mid n \in N - \{n_0\}\}$. Retaining a large set of annotations is inefficient, even for annotation-based invariant generation techniques. Moreover, some of annotations are very specific to a program, e.g., σ_n is only useful for proving correctness of $\texttt{P}_n$.

Our technique minimizes $\mathcal{A}$ while retaining its invariant-generation power. The "power" is captured using a *cost* metric based on the ability of a set of annotations to prove a set of programs correct, given a fixed verifier. The cost is ∞ if some program cannot be proved, otherwise, it reflects the running time of the verifier. Our algorithm simplifies $\mathcal{A}$ by dropping annotations or making them syntactically simpler as long as the cost does not increase (or only increases by a *tolerable* amount; the exact formulation can be found in Section 3).

For illustration, assume that the cost becomes ∞ as soon as the annotations cannot establish some loop invariant or postcondition of a recursive procedure (intuitively, because these are the critical parts of a proof), and is unit cost otherwise. Starting with $\mathcal{A}$, our algorithm drops depth == 0 and η from this set because these are not important for the inductive argument; i.e., cost remains unit after dropping them. Next, if it tries to drop ψ_0, the cost becomes ∞ because the loop invariant of $\texttt{P}_{n_0}$ is lost. Thus, ψ_0 is retained in $\mathcal{A}$. Next, each of the σ_n annotations get dropped. Even though these annotations were loop invariants in the original proofs, they can be replaced by the more general annotation ψ_0 that is present in $\mathcal{A}$. In this way, learning from a large set of proofs increases the chances of finding annotations that generalize.

Finally, while $\psi_0 \wedge \eta$ cannot be dropped, our algorithm tries to simplify its Boolean structure. The algorithm simplifies it to η: having annotations $\{\psi_0, \eta\}$ is enough for annotation-based invariant generation to establish $\psi_0 \wedge \eta$ as an invariant. At this point, the algorithm reaches a fixpoint where no annotation can be dropped or simplified and it returns the set: $\{\psi_{-1}, \psi_0, \psi_1, \eta\}$.

The algorithm is non-deterministic; it could have chosen to drop $\psi_0 \wedge \eta$ in its first iteration because η was still present in $\mathcal{A}$. In general, our algorithm only guarantees a locally optimal solution with respect to a given cost metric. Globally-optimal solutions are also possible to compute, but at a higher cost, which was not justified in our experiments.

Although the programs considered here are simple, they are derived from real-world code. The API is based on SDV's `SpinLockRelease` property that keeps track of multiple *spinlocks* held by a driver. The example programs are derived from real drivers. The program `Q` illustrates an uncommon (but not rare) scenario where recursion happens via kernel callbacks (driver code itself typically does not exhibit recursion).

Abstract Annotations and a Shared Vocabulary Annotations are formulas over program variables. In general, different programs have different variables. To abstract away from program-specific variables, we introduce the concept of an *abstract annotation* that is a formula over *generic* and *shared* variables.

We call the set of global variables common to all programs in the repository as the *shared vocabulary*. We assume these variables serve a similar role in all programs, e.g., the `depth` variable in our running example will be present in all programs that exercise the acquire-release API. Shared variables, i.e., variables in the shared vocabulary, can be freely used in annotations because they are present in all programs.

Generic variables are not specific to any program. There are fours kinds of generic variables: $\{\text{Local}, \text{Global}, \text{FormalIn}, \text{FormalOut}\}$. An annotation is converted to an abstract annotation by replacing variables by generic variables of the corresponding type. For example, consider the postcondition [`x == y`]@`f` on a procedure `f` with formal input argument `x` and formal output argument `y`. This will get converted to the abstract annotation (`$fin == $fout`) where `$fin` and `$fout` are generic variables of type FormalIn and FormalOut, respectively.

Abstract annotations are concretized when applied to a program. Let V be the shared vocabulary. Let p be a program and *proc* a procedure in p. We define a concretization function $\gamma_{p,proc,V}$ as follows. For an abstract annotation a, $\gamma_{p,proc,V}(a)$ returns all annotations such that a generic variable of type Global is substituted with some global variable of p, a generic variable of type FormalIn is substituted with some formal-in parameter of *proc*, and similarly for FormalOut and Local. $\gamma_{p,proc,V}$ must return all such annotations.

The shared vocabulary V is used as an optimization to limit the number of concretizations of an abstract annotation. V can be empty, in general, and our technique will still apply.

3 Framework

We now formally present the annotation inference algorithm, but first we introduce the necessary notation and definitions.

Language. We assume an imperative programming language with standard features such as global variables, procedures, `assume` and `assert` statements, assignments, etc. We also assume that programs in this language do not have loops. Loops can be encoded using recursion. This allows our framework to only concentrate on procedure postconditions for establishing proofs of correctness.

Given a program p, we denote the set of procedures in p with $\mathit{procs}(p)$. Each procedure can be *annotated* with any number of first order logic (FOL) formulas. These formulas are defined over procedure parameter and global variables and they don't take part in program execution; they are used by program verifiers as candidate postconditions for establishing program correctness.

Abstract annotations. Let V be a set of variables called the shared vocabulary. All programs must contain V as global variables. Let G be a set of generic variables. None of the programs contain a variable from G. An *abstract annotation* a is a formula over variables in $V \cup G$. Further, for every program p and procedure $\mathit{proc} \in \mathit{procs}(p)$, we assume a function $\gamma_{p,\mathit{proc},V}$ that maps an abstract annotation to a set of concrete annotations. As defined in the previous section, $\gamma_{p,\mathit{proc},V}$ substitutes generic variables with variables in scope of proc.

We call a finite set of abstract annotations $t \in \mathfrak{T}$ a *template*. Given a program p and a template t, $\mathit{annotate}(p,t)$ returns p where each procedure $\mathit{proc} \in \mathit{procs}(p)$ is annotated with the set of program annotations $\bigcup_{a\in t} \gamma_{p,\mathit{proc},V}(a)$.

Verification. Given a fixed verifier, for a program p and a template t, we say that $\mathit{proves}(p,t)$ holds if the verifier can prove the correctness of $\mathit{annotate}(p,t)$. The verifier can use the procedure annotations as potential postconditions during the verification. Also, we require that if $\mathit{proves}(p,t)$ and $t \subseteq t'$, then $\mathit{proves}(p,t')$ must hold as well. In other words, if a set of abstract annotations is sufficient for proving some program correct then all of its supersets are also.

3.1 Problem

Our annotation inference problem is defined using a notion of an objective relation. Such relations are used to encode what templates (and hence annotations) are more desirable for the current application in mind.

Definition 1 (Objective relation for a program). *We say* $\to_p: \mathfrak{T} \times \mathfrak{T}$ *is an objective relation for a program* p *iff (1) it is well-founded and (2) for each* $t \in \mathit{dom}(\to_p)$*, the set* $\{t' \mid t \to_p t'\}$ *is finite.*

The objective relation is hence *finite branching*. One example of such a relation is the proper subset relation, i.e., $t \rightarrow_p t'$ iff $t' \subset t$. Another example would be the relation where t' is a copy of t except that an annotation in t' is a subformula of the corresponding annotation in t. This relation roughly corresponds to the *syntactically simpler* concept mentioned in Section 1. Section 4 presents an objective relation used in our experiments with SDV.

We extend the objective relation to a set P of programs $p_1, \ldots, p_n$:

Definition 2 (Objective relation for a set of programs). *We define* $\rightarrow_P$: $\mathfrak{T} \times \mathfrak{T}$, *an objective relation for a set of programs* $P = \{p_1, \ldots, p_n\}$ *as a well founded and finite branching relation* $t \rightarrow_P t' \Leftrightarrow \oplus_i t \rightarrow_{p_i} t'$.

The operator $\oplus$ is a reduction operator and examples include conjunction $\bigwedge$, disjunction $\bigvee$, etc.

We proceed by defining the notion of a minimal template that intuitively stands for a locally optimal template. The locality is defined as a branching set of a template induced by a given objective relation.

Definition 3 (Minimal program template). *Given a program* p*, a template* t *such that* $proves(p, t)$*, and an objective relation* $\rightarrow_p$*, we say* t' *is a minimal template iff:*

1. $proves(p, t')$
2. *there exists no* t'' *such that* $t' \rightarrow_p t''$ *and* $proves(p, t'')$

The definition states that a minimal template must prove a given program and none of its immediate ($\rightarrow_p$) successors do. We point out that, in the above definition, t only ensures that p is correct and this definition establishes no relationship between a minimal template and t. However, the results computed by the implementations for finding minimal templates can be dependent on t.

Our inference algorithm is built around the notion of a minimal template. We hence define the problem of finding a minimal template for a given program.

Problem 1 (Computing minimal template). Given a program p, a template t such that $proves(p, t)$, and an objective relation $\rightarrow_p$, the problem of computing a minimal template is finding a formula t' that is minimal subject to p and the ordering $\rightarrow_p$.

We define a *program repository* as $R = [(p_1, t_1), ..., (p_n, t_n)]$ where $proves(p_i, t_i)$ for $1 \leq i \leq n$. Repositories capture verification histories. The set of programs in the repository R is denoted by P_R. The actual technique used for proving the correctness of p_i can be arbitrary. However, we envision that in practice, a verifier will be fixed for the whole repository. We now define a locally optimal template for a verification history.

Definition 4 (Minimal repository template). *Given a program repository* $R = [(p_1, t_1), ..., (p_n, t_n)]$ *where* $proves(p_i, t_i)$ *for all* $1 \leq i \leq n$*, and an objective relation* $\rightarrow_{P_R}$*, we say that* T *is a minimal repository template (subject to* $\rightarrow_{P_R}$*) iff the following holds*

1. $proves(p_i, T)$ for all $1 \leq i \leq n$
2. there exists no T' such that $T \rightarrow_{P_R} T'$ and $proves(p_i, T')$ for all $1 \leq i \leq n$

We are now ready to formally state the problem of inferring program annotations from past verification runs.

Problem 2 (Inferring program annotations). Given a program repository $R = [(p_1, t_1), ..., (p_n, t_n)]$ where $proves(p_i, t_i)$ for all $1 \leq i \leq n$, and an objective relation $\rightarrow_{P_R}$, the problem of inferring program annotations is to find a template T that is a minimal repository template subject to $\rightarrow_{P_R}$.

In the sequel, we suppress the subscripts of the $\rightarrow$ relations for brevity. We now show algorithms for solving the problems of computing minimal templates.

3.2 Solution

We start with the solution for Problem 1 shown in Algorithm 1. The `MinTemplate` algorithm assumes that a given template t is sufficient to establish correctness of p and that $\rightarrow$ is an objective relation. We start by considering t as a minimal template candidate (line 2). We continue by enumerating all immediate successors of t by $\rightarrow$ (line 4). Then, the algorithm checks if any of successors can prove p (line 5). If so, then the algorithm sets such a successor as a candidate minimal template and repeats the whole process (lines 6 and 7). Otherwise, the minimal candidate is returned as the solution (line 8).

Theorem 1. *Let p be a program, t a template, and $\rightarrow$ an objective relation. If $proves(p, t)$, then Algorithm 1 computes a minimal template for p, t, and $\rightarrow$.*

The complexity of the algorithm depends on $\rightarrow$ relation and implementation of *proves*. Assuming *proves* has unit complexity, the running time of Alg. 1 is $O(l \cdot m)$, where l is the longest well-founded chain of $\rightarrow$ and m is the maximum size of the branching sets $max\{|\{t' \mid t \rightarrow t'\}| \mid t \in dom(\rightarrow)\}$. However, proving a program correct is undecidable in general and asymptotically takes exponential time even for decidable cases. Further, annotating a program p given a template t can also be expensive if for $proc \in procs(p)$, the concretization function $\gamma_{p,proc,V}$ has a large image; t can then potentially be instantiated with a large number of concretizations. This high complexity of the algorithm can be remedied in practice by choosing $\gamma_{p,proc,V}$ with smaller images and exploiting the structure of $\rightarrow$ if the relation is known beforehand.

Algorithm 2 computes a minimal repository template by building on Algorithm 1. First, we find the minimum template for each program in the repository and store it in *mints* (lines 3- 5). Next, the minimal repository template is set to the empty set of clauses (line 6). The outer loop (lines 7- 14) has the invariant that after the i^{th} iteration C is a minimal repository template for the sub-repository $[(p_1, t_1), \ldots, (p_i, t_i)]$. The inner loop checks if an immediate successor of C can prove the correctness of the programs $p_1, \ldots, p_i$. If so, then C is updated to that successor template.

Algorithm 1 Computing a minimal program template

```
Require: $proves(p, t)$ and $\rightarrow$ is an objective relation
1: procedure MinTemplate($p, t, \rightarrow$)
2:   $mint \leftarrow t$
3:   loop
4:     for all $t' \in \{t' \mid t \rightarrow t'\}$ do
5:       if $proves(p, t')$ then
6:         $mint \leftarrow t'$
7:         goto 3
8:     return $mint$
```

Algorithm 2 Computing a minimal repository template

```
Require: $proves(p_i, t_i)$ for all $(p_i, t_i) \in R$
Require: $\rightarrow$ is an objective relation
1: procedure MinRepoTemplate($R, \rightarrow$)
2:   $mints \leftarrow [\,]$
3:   for all $i \in [1, ..., |R|]$ do
4:     $(p, t) = R[i]$
5:     $mints[i] = MinTemplate(p, t, \rightarrow)$
6:   $C = \varnothing$
7:   for all $i \in [1, \ldots, |R|]$ do
8:     $C = C \cup mints[i]$
9:     for all $t \in \{t' \mid C \rightarrow_{\{p_1, \ldots, p_i\}} t'\}$ do
10:      $b = true$
11:      for all $j \in [1, \ldots, i]$ do
12:        $b = b \wedge proves(p_j, t)$
13:      if $b$ then
14:        $C = t$
15:  return $C$
```

One possible optimization is to cache the clauses under which a program can/cannot be proved. Therefore, if $proves(p, t)$ is in the cache and we later make a query $proves(p, t')$ where $t \subseteq t'$, then, we can return *true*. Similarly, if $\neg proves(p, t)$ is in the cache and we make a query $proves(p, t')$ where $t' \subseteq t$, then, we can return *false*.

Theorem 2. *Let R be a program repository and $\rightarrow$ an objective relation. If $proves(p, t)$ for all $(p, t) \in R$, then Algorithm 2 computes a minimal repository template for R and $\rightarrow$.*

We also point out that loop starting at line 7 is in fact not necessary for optimality. The algorithm can immediately start with C as the union of all minimal program templates stored in *mints*. However, such a C could become impractically large. In Algorithm 2, the size of C is kept moderate. Observe that minimal templates computed using these two versions of the algorithm might not be the same. This is because minimal templates are not unique.

Proofs of theorems 1 and 2 are given in Appendix A.

4 Application: Static Driver Verifier

The Static Driver Verifier (SDV) has been an important success story for verification technology. It has utilized SLAM for over a decade with several upgrades

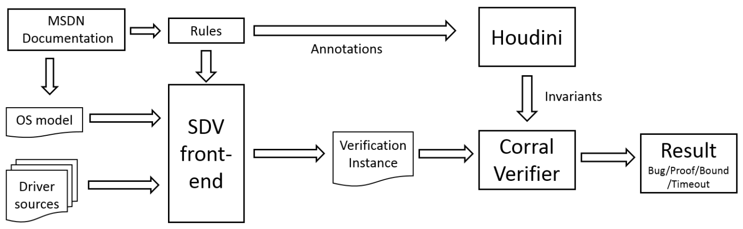

Fig. 3. An overview of SDV.

along the way [3], then further by switching to an SMT-based verifier called Corral [14] and upgrading again [15,16]. Showing further improvements truly builds on the state-of-the-art in the area.

An overview of SDV [20] is shown in Figure 3. SDV accepts the source code of a device driver as input and links it against a model of the kernel (called "OS Model" in the figure). It then checks multiple rules that the driver must satisfy. These rules and kernel contracts (encoded in the OS Model) are made known to driver developers via MSDN[4]. Each driver and rule produces multiple programs with assertion, called *verification instances* that must all verify.

Corral operates by lazily inlining procedures and utilizing an SMT solver to search through a partially-inlined program. To help Corral, SDV uses annotation-based invariant generation. Given a rule and a verification instance, SDV generates annotations and runs Houdini [8] to compute invariants over the annotations. These invariants are injected back to the verification instance as assume statements, which help Corral prune search without compromising soundness.

Much work has been put into fine-tuning the annotation generation inside SDV. One can view the generator as a table lookup; SDV maintains a template for each rule that has been tuned over the years. We refer to such templates as "Manual Templates". We compare the minimal templates inferred by our technique (called the "Inferred Templates") against these manually-provided ones, based on SDV's performance with either set of templates.

Corral has four possible outcomes, as mentioned in Figure 3. Corral uses over-approximations (refined by invariants inferred by Houdini), hence it can prove correctness and return "proof". Corral stops search when it hits an internal coverage bound [17] and returns "bound". Although this is an inconclusive verdict, it is still considered more useful than a Timeout because the latter does not guarantee any coverage.

Training and Test Suites. For internal testing, SDV uses a set of toy drivers, collectively called the *Rule Test Suite* (RTS) for quick "smoke testing" of SDV. These drivers are small, often a few hundred lines of code (with relevant part in tens of lines of code only). We use RTS as the program repository for inferring a minimal repository template for each rule. The use of RTS as the training set is quite natural as it allows us to leverage existing test cases to learn

[4] https://msdn.microsoft.com/en-us/library/windows/hardware/ff552808(v=vs.85).aspx

and improve performance on real drivers. RTS consists of 304 drivers totaling around 100KLOC.

Once we infer a template, we measure the performance of SDV using the template on a set of real device drivers. We use 66 device drivers for this task, totaling around 700KLOC.

SDV has hundreds of rules. For this paper, we concentrate on a collection of 28 rules. Of these, 14 rules were selected because they were known to cause performance issues for SDV. We added, randomly chosen, 14 other rules. These 28 rules on the 66 drivers produced a total of 1420 verification instances.

Obtaining Annotations. The RTS suite consists of fairly small drives that are easy to prove manually. We, however, automated the entire process by running a proof-generating verifier. Our implementation uses Duality [19] but conceptually we could have used tools such as SLAM and Yogi [5] as well.

Our objective is to learn a template specific to each rule. Thus, for each of the 28 rules, we (1) form a repository of programs that assert the rule and their corresponding correctness proofs and (2) define the shared vocabulary V to consist of model variables of the rule as well as OS model variables. Every program in a repository is guaranteed to include the corresponding shared vocabulary as global variables.

We restrict abstract annotations to be *clauses*, i.e., a disjunction of formulas. Conjunctions at the top level are broken down into multiple annotations, one for each conjunct. For convenience, we think of a clause as a set of formulas where disjunction is implied between the elements of the set. The empty set corresponds to `true`. Given two annotations a_1 and a_2, by $a_1 \subseteq a_2$ we therefore designate that the formulas of a_1 are a subset of the formulas of a_2. For a template t, we define $\bar{t}$ to be t consistently indexed by the set $\{1, ..., |t|\}$. In other words, $\bar{t} = \{a_1, ..., a_{|t|}\}$ where $a_i \in t \iff a_i \in \bar{t}$. For convenience, we define $\bar{t}[i] = a_i$. Given two templates t_1 and t_2, we say t_2 is *simplified* (or simpler) than t_1, written $t_2 \trianglelefteq t_1$, iff (1) $|t_1| = |t_2|$ and (2) $\bar{t_2}[i] \subseteq \bar{t_1}[i]$ for all $1 \leq i \leq n$. If $t_2 \trianglelefteq t_1$ and there exists i such that $\bar{t_2}[i] \subset \bar{t_1}[i]$, we say t_2 is *strictly simpler* than t_1, written $t_2 \triangleleft t_1$. Finally, $t_2 \triangleleft_1 t_1$ holds iff $t_2 \triangleleft t_1$ and $|\bar{t_1}/\bar{t_2}| = 1$. In other words, t_2 is strictly simpler than t_1 but only at one abstract annotation; we then say t_2 is 1-simpler than t_1. For example, suppose a template t' is t except that literal l is in $\bar{t}[1]$ but not in $\bar{t'}[1]$. Then we have $t' \trianglelefteq t$, $t' \triangleleft t$, and $t' \triangleleft_1 t$.

Objective Relation. Given a program p and a template t, let $h(p, t)$ be the result of running Houdini on *annotates*(p, t). Further, let $c_{perf}(p, t)$ denote the number of procedures that Corral inlined if it was able to prove correctness of $h(p, t)$, and -1 otherwise. $c_{perf}(p, t)$ measures the performance of Corral. We use it as a proxy for the running time, which is independent of the machine configuration. We now define the *corral* objective relation. Given a program p and two templates t_1 and t_2, $t_1 \rightarrow_p t_2$ holds iff:

- $t_2 \triangleleft_1 t_1$
- $0 \leq c_{perf}(p, t_2) \leq 2 * c_{perf}(p, t_1)$

The above definitions encode our intention to find those templates that are structurally simpler and smaller, if one views making a clause empty as removing it.

The reason why we chose 1-simpler relation instead of the general simplification is that we want to keep branching sets of the objective relation tractable. This way, we are sacrificing optimality for better inference times. Also note that our objective relation allows the performance of Corral to get worse when using t_2. But Corral must still be able to prove correctness. We allow the degrade in performance to allow more opportunities for the templates to get simplified. However, we still restrict the performance to not get out of hand (not more than a factor of 2). This allows us to kill the execution of Corral on $h(p, t_2)$ as soon as it inlines twice as many procedures as on $h(p, t_1)$, without waiting for a verdict.

Lastly, we define a repository objective relation. For a set of programs $P = \{p_1, ..., p_n\}$ we define $t_1 \rightarrow_P t_2 \iff \bigwedge_i t_1 \rightarrow_{p_i} t_2$. In other words, we want to infer templates that are consistently optimal for all repository programs, subject to the objective relation $\rightarrow_p$.

We use Algorithm 2 in our experiments, with some improvements based on the particular objective relation defined above. These improvements are mentioned in Appendix B. We set a timeout of 8 hours for the algorithm. If the execution reaches a timeout, we simply return the current value of C. In practice, for SDV, the template inference needs to be performed just once in a release cycle; hence, one can devote much more time for inference.

5 Evaluation

We implemented our algorithms in a tool called ProofMinimization. It accepts a list of annotated programs written in Boogie [18] as input and computes the minimal repository template. The shared vocabulary is automatically set to the set of global variables common to all input programs[5]

We ran our experiments on a cluster of 6 identical server-class machines. Each of the servers had Intel Xeon CPUs 1.8 GHz, 64 GB RAM and 16 logical processors. The total CPU time of our experiments exceeded well over a month. We relied on parallelism extensively to produce results in a reasonable amount of time.

Training Modes. Our experiments evaluate two different ways of using the inferred templates. In the first mode, called MT, the inferred templates only augment the manual templates. This is achieved by adding the manual templates to each (p, t) pair in our program repository and requiring our algorithm to never throw out a manually-generated annotation. This mode of operation is more controlled: it does not seek to replace existing manual effort, but rather to just augment it.

The second mode of operation, called NT, does not use manual templates at all. This simulates the scenario when no manually-generated templates had been added to SDV. It answers how much of the manual effort behind the design of manual templates can be automated using our techniques.

[5] Implementation is available open-source `https://github.com/boogie-org/corral/tree/master/AddOns/ProofMinimization`

Clearly, the quality of inferred templates will depend on the quality of the training set, i.e., the repository used for inferring a minimal repository template. Because it was never intended to use RTS for inferring templates, the training sets are sometimes inadequate for our approach. For instance, for a few rules, RTS only contains buggy drivers. (Inferring annotations from buggy programs is an interesting problem, but outside the scope of this paper.) Thus, we also evaluate expanding our training set by sampling a randomly-chosen fraction of the 66 real drivers and including them in the training set. The test set, i.e., the set of unseen drivers over which we evaluate the inferred templates then shrinks to the remaining drivers.

Let Train($f\%, m$), where $0 \leq f \leq 100$ and m is either MT or NT, refer to the experiment where the training set consisted of all RTS drivers and f percent of the real drivers, and the inference was done in mode m. An exception is the special case of Train($100\%, MT$) which denotes that all drivers were used in the training set as well as the test set. We use Train($100\%, MT$) as a limit study on the quality of annotations that we can infer.

Results. For the experiment Train(0%, NT), the running time of ProofMinimization is shown in Figure 4. It takes just a couple of minutes for some rules while it timeouts after 8 hours for five rules. Not including the rules for which ProofMinimization times out, it takes roughly 2.5 hours on average to compute the result.

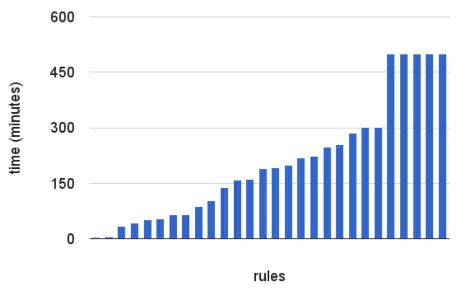

Fig. 4. Annotation inference times for the 28 rules.

The results with various different training modes are reported in Tables 1 and 2. Each of the tables compare three versions of SDV. Version called "None" does not use any templates and captures the performance of Corral without any annotations. "Manual" refers to using manual templates, which is the currently-shipping production system. "Inferred" refers to using the set of annotations inferred by our tool. Each of the tables measure performance in terms of the number of timeouts (#TO), number of times the coverage bound was hit (#Bnd), the number of bugs reported (#Bugs), the average running time of Houdini+Corral (Avg), and the average running time of just Houdini (Houd).

Table 1(left) compares performance for Train(0%, MT). In this mode, our inferred set of annotations was empty for 12 of the 28 rules. The inferred set can be empty when the RTS wasn't rich enough, or because the manual templates were sufficient. In this case, the performance of "Inferred" matches with that of "Manual". Table 1 compares performance on the remaining 16 rules where we did infer annotations. These results demonstrate that our extra set of annota-

				Time (sec)	
Config	#TO	#Bnd	#Bugs	Avg	Houd
None	77	228	46	94.3	0
Manual	27	88	46	71.9	10.0
Inferred	24	37	46	51.6	12.1

				Time (sec)	
Config	#TO	#Bnd	#Bugs	Avg	Houd
None	112	265	64	104.3	0
Manual	50	91	64	70.5	10.1
Inferred	46.6	41	64	59.4	14.6

Table 1. Left: Results for Train(0%, MT) on a total of 16 rules with 873 verification instances. Right: Results for Train(30%, MT), averaged across three runs, on a total of 25 rules with 1002 verification instances.

				Time (sec)	
Config	#TO	#Bnd	#Bugs	Avg	Houd
None	143	342	104	98.1	0
Manual	55	123	108	58.4	9.3
Inferred	45	25	108	46.1	16.2

				Time (sec)	
Config	#TO	#Bnd	#Bugs	Avg	Houd
None	143	342	104	98.1	0
Manual	55	123	108	58.44	9.3
Inferred	59	92	108	56.5	9.1

Table 2. Left: Results for Train(100%, MT) on a total of 28 rules with 1420 verification instances. Right: Results for Train(0%, NT) on a total of 28 rules with 1420 verification instances.

tions are useful. The number of timeouts come down a fraction and the number of times the coverage bound was hit comes down significantly. All of these previously inconclusive cases, 54 in number, convert to a proved verdict. In summary, the total number of inconclusive answers drops down by 47%. Moreover, even though Houdini ran slower because of the extra annotations, the performance improvement in Corral made the overall system much faster (22%).

For Train(30%, MT), we did three runs, each time sampling a different fraction of the drivers. The results for "Inferred" were averaged across the three runs and are shown in Table 1 (right). It is interesting that performance is similar to Train(0%, MT). Using a fraction of drivers did not provide much new information. However, the results of Train(100%, MT) shown in Table 2 (left) do indicate that drivers potentially carry information not present in RTS. Learning over all drivers increased the quality of inferred annotations significantly (even though the running time of Houdini is highest in this setting).

Table 2 (right) shows results of Train(0%, NT). With no help from manual templates, Corral's performance depends even more significantly on the inferred set of annotations. We were able to achieve a similar quality of results compared to using manual templates. There was a near-equal split between rules on which inferred annotations do better and ones on which manual templates do better. In consultation with the SDV team, we realized that the exercise of setting up manual templates often borrowed annotations *across* rules, where annotations useful for one rule would be generated for similar rules as well. In our setting, we did not explore sharing information between rules.

The number of abstract annotations per template, on average, was 12.78 for the manual templates. The Train(0%, MT) experiment produced 13.63 annotations per template, on average. The Train(100%, MT) experiment produced 16.6 annotations per template, on average. The Train(0%, NT) experiment produced 3.3 annotations per template, on average, which is significantly smaller than the manual templates. We discuss some examples of useful annotations that our technique was able to infer, but were missed by manual effort, in Appendix C.

To summarize, Train(0%, MT) results show that we can significantly improve the performance of a production system by augmenting existing manual effort. The results for Train(0%, NT) show that as new rules are developed and test cases are added to RTS, we can automatically generate useful annotations avoiding the need for further manual effort in coming up with new templates.

6 Related Work

This work falls into the category of predicting program properties from codebases. For example, JSNice learns from Javascript repositories on GitHub and predicts more legible identifier names and (unverified) type annotations [23]. In contrast, this work is the first attempt at inferring annotations from verification histories and demonstrating their use in an industrial-scale verification setting. Other approaches use codebases to predict different program properties rather than annotations [21,22,24]. For instance, work presented in [22] applies Bayesian optimization on existing codebases to learn a strategy for deciding for which part of an unseen program a static analyzer should sacrifice time for precision while performing the analysis.

There are verification and testing approaches that leverage previous version of a program under analysis. For example, [10] improves performance of test generation for a program by leveraging existing tests belonging to a previous version. Regression verification verifies the equivalence of two successive versions of a program [11]. For similar programs, [11] argues that this verification task is easier than our goal here, i.e., formal verification of a stand alone program. A recent generalization of regression verification is differential assertion checking where a verifier checks that a bug is not introduced in going from one program version to another [13]. Techniques in [7] extrapolate from predicates used for verifying a program under sequential consistency to verify the same program under relaxed memory model. Our work performs inference using different programs sharing a common vocabulary instead of the previous versions of a given program.

The techniques described in this paper can be used to infer invariants using verification histories. Previous approaches to invariant inference perform analysis only over the program under consideration. These include invariant inference using static analysis [6] or learning from concrete executions [9,25]. Our work is complementary to these.

7 Conclusion

We present a framework for inferring a *small* set of *useful* program annotations from a repository of past verification runs. Our algorithm uses minimization techniques to come up with a small set of annotations that apply broadly to many programs. Using our algorithm, we infer useful annotations that improve the performance of SDV. By utilizing the inferred annotations, we reduce the number of inconclusive answers by 47% while running 22% faster on average, even for a heavily-optimized system.

References

1. Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. Ufo: Verification with interpolants and abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 637–640. Springer, 2013.
2. Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. The static driver verifier research platform. In *Computer Aided Verification*, pages 119–122. Springer, 2010.
3. Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with slam. *Communications of the ACM*, 54(7):68–76, 2011.
4. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. Automatic predicate abstraction of c programs. In *ACM SIGPLAN Notices*, volume 36, pages 203–213. ACM, 2001.
5. Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, SaiDeep Tetali, and Aditya V. Thakur. Proofs from tests. *IEEE Trans. Software Eng.*, 36(4):495–508, 2010.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.
7. Andrei Marian Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 84–104, 2013.
8. Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, pages 500–517, 2001.
9. Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 69–87, 2014.
10. Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 112–128, 2011.
11. Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Softw. Test., Verif. Reliab.*, 23(3):241–258, 2013.
12. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 232–244, 2004.
13. Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 345–355, 2013.

14. Akash Lal and Shaz Qadeer. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 202–212, 2014.
15. Akash Lal and Shaz Qadeer. A program transformation for faster goal-directed search. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 147–154, 2014.
16. Akash Lal and Shaz Qadeer. DAG inlining: a decision procedure for reachability-modulo-theories in hierarchical programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 280–290, 2015.
17. Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 427–443, 2012.
18. K Rustan M Leino. This is boogie 2. *Manuscript KRML*, 178:131, 2008. `http://https://github.com/boogie-org/boogie`.
19. Kenneth L McMillan and Andrey Rybalchenko. Computing relational fixed points using interpolation. Technical report, Technical report, 2012. available from authors, 2013.
20. Microsoft. Static driver verifier. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff552808(v=vs.85).aspx`.
21. Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 997–1016, 2012.
22. Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 572–588, 2015.
23. Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from `big code`. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 111–124, 2015.
24. Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 44, 2014.
25. Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 88–105, 2014.

A Proofs

Proof of Theorem 1.

Proof. We refer to Algorithm 1. Since $\rightarrow$ is finite branching, we have that inner loop at line 4 terminates. From the fact that $\rightarrow$ is well-founded, it follows that outer loop at line 3 also terminates. Since lines 4 and 5 simply follow the definition of a minimal template, we have that the returned template is minimal.

Proof of Theorem 2.

Proof. We refer to Algorithm 2. For every subrepository $R_i = [(p_1, t_1), \dots, (p_i, t_i)]$ the relation $\rightarrow_{R_i}$ is well-founded and finite branching for any chosen reduction operator. Since the body of the loop at line 9 follows the definition of a minimal repository template, then at the end of each iteration of the loop at line 7 C is a minimal template for R_i, as pointed out earlier. The result then follows from the case when $i = |R|$.

B Further Improvements to the Algorithm

We use a slightly modified version of Algorithm 2 for computing a minimal repository template. We describe the algorithm by only explaining the modifications that we introduced.

We first note that the *corral* objective relation has to be computed by actually running Corral. While performing a greedy descent, we simply enumerate all 1-simplified templates and run Corral on each of them. This run tells us whether the program can be proved and the number of inlined procedures, if any. Also, we enumerate first those annotations where the corresponding simplified clause is empty. Then, we enumerate annotations where the simplified clause has one literal, and then when it has two, and so forth, until we have enumerated all 1-simplified templates. This way, we are heuristically choosing the well-founded chains of smaller lengths while searching for a minimal template.

The second modification we introduce has the purpose of keeping C small in Algorithm 2. While computing a minimal template for a single program, we first check whether any of the previously computed minimal templates is perhaps minimal for the new program. If so then we continue by analyzing the next program. As a result, the number of distinct minimal templates in *mints* reduces, and so does the size of C. Further, due to the definition of $\gamma_{proc,p,V}$, our algorithm can produce the same annotated program for different templates. In that case, the result of running Houdini+Corral for the first template is same as the result of running them on the other template. We therefore use the concretized annotations to cache Corral's outcome and reuse the results when possible.

C Supplemental Discussion on the Evaluation

In this section, we highlight some of our experience with inferring useful annotations. SDV rules verify kernel API usage; in many cases this does not require deep reasoning over data structures or the heap. Consequently, we find that annotations solely over scalar variables, such as model variables of the rule and the OS model (which is also our shared vocabulary for annotations), help significantly in establishing proofs of correctness.

Irql-based rules. An *Interrupt Request Level* (IRQL) quantifies the priority associated with a task. Tasks with higher IRQL cannot be interrupted by tasks with a lower IRQL. Drivers often raise IRQL level to perform critical activity uninterrupted, but are required not to spend too much time at a high IRQL. SDV checks that drivers do not call certain kernel APIs when at a high IRQL[6].

Figure 5 shows IRQL modeling and usage. OS model variable `sdv_irql_current` maintains the current IRQL value. APIs `KeRaiseIrql`[7] and `KeLowerIrql` change the IRQL value. Proving correctness of `main` requires a loop invariant that the value of `sdv_irql_current` is unchanged across loop iterations, which in turn requires that the value of `loc` is maintained across iterations. The former annotation is generally useful and was present in the manual template, however, the latter was not. We infer the annotation `$floc` == `old($floc)` where `$floc` is a generic variable of type LOCAL (see Section 2). This allows us to establish the invariant that certain loops do not change the value of certain local variables across loop iterations. Inferring this annotation helped SDV significantly.

Multi-variable correlations. Some SDV rules define and manipulate several model variables. Manual templates mostly have single-variable annotations that capture the effect a procedure's execution on a single variable. However, these are insufficient to capture inter-variable relationships. We infer several annotations over multiple variables which helped significantly with some rules.

[6] https://msdn.microsoft.com/en-us/library/windows/hardware/ff547747(v=vs.85).aspx

[7] https://msdn.microsoft.com/en-us/library/windows/hardware/ff553079(v=vs.85).aspx

```
// DRIVER
procedure main() {
   var loc: int;
   call init();
   call loc := KeRaiseIrql(2);
   while(*)    {
     call KeLowerIrql(loc);
     call do_work_at_low_irql();
     call loc := KeRaiseIrql(2);
   }
   call KeLowerIrql(loc);
}

// RULE
procedure do_work_at_low_irql()
{  assert sdv_irql_current == 0; }

// OS MODEL and RULE
var sdv_irql_current: int;

procedure init()
{  sdv_irql_current := 0; }

procedure KeRaiseIrql(new_irql: int)
  returns (old_irql: int) {
   old_irql := sdv_irql_current;
   sdv_irql_current := new_irql;
}

procedure KeLowerIrql(new_irql: int)
{  sdv_irql_current := new_irql; }
```

Fig. 5. IRQL modeling and usage