Distributed Bounded Model Checking

Prantik Chatterjee*, Subhajit Roy*, Bui Phi Diep†, Akash Lal‡
*IIT Kanpur, {prantik, subhajit}@cse.iitk.ac.in
†Uppsala University, bui.phi-diep@it.uu.se
‡Microsoft Research, akashl@microsoft.com

***Abstract*—Program verification is a resource-hungry task. This paper looks at the problem of parallelizing SMT-based automated program verification, specifically bounded model-checking, so that it can be distributed and executed on a cluster of machines. We present an algorithm that dynamically unfolds the call graph of the program and frequently splits it to create sub-tasks that can be solved in parallel. The algorithm is adaptive, controlling the splitting rate according to available resources, and also leverages information from the SMT solver to split where most complexity lies in the search. We implemented our algorithm by modifying CORRAL, the verifier used by Microsoft's Static Driver Verifier (SDV), and evaluate it on a series of hard SDV benchmarks.**

I. INTRODUCTION

Program verification has a long history of over five decades and it has been consistently challenged over this entire duration by the continued increase in the size and complexity of software. As the efficiency of techniques and solvers has increased, so has the amount of software that is written. For this reason, *scalability* remains central to the applicability of program verification in practice.

This paper studies the problem of automated program verification. In particular, we consider Bounded Model Checking (BMC) [1]: the problem of reasoning over the entire space of program inputs but only over a subset of program paths, typically up to a bound on the number of loop iterations and recursive calls. BMC side-steps the need for (expensive and undecidable) inductive invariant generation and instead directly harnesses the power of SAT/SMT solvers in a decidable fragment of logic. BMC techniques are popular; they are implemented in most program verification tools today [2, Table 5].

Our goal is to scale BMC by parallelizing the verification task and distributing it across multiple machines to make use of larger compute and memory resources. The presence of several public cloud providers has made it easy to set up and manage a cluster of machines. While this distributed platform is available to us, there is a shortage of verification tools that can exploit it.

Parallelizing BMC. BMC works by generating logical encodings, often called *verification conditions* or VCs, for a subset of program paths that are then fed to an SMT solver to look for potential assertion violations in the program. We aim to retain the same architecture, where we continue to use the SMT solver as a black-box, but generate multiple different VCs in parallel to search over disjoint sets of program paths. This allows us to directly consume future improvements in SMT solvers, retaining one of the key advantages of BMC.

Our technique works by *splitting* the set of program paths into disjoint subsets that are then searched independently in parallel. The splitting is done by simply picking a control node and considering (a) the set of paths that go through the node, and (b) the set of paths that do not. Splitting can happen multiple times. The decisions of *what* node to split and *when* to split are both taken dynamically by our technique. We refer to the BMC problem restricted to a set of splitting decisions (i.e., nodes that must be taken, and nodes that must be avoided) as a *verification partition*.

Verification starts by creating multiple processes, each of which have access to the input program and are connected over the network. One process is designated as the *server* while the rest are called *clients*. The search starts sequentially on one of the clients that applies standard BMC on the input program. At some point in time, which is controlled by the *splitting rate*, the client chooses a splitting node, thus creating two partitions. The client continues verification on one of the partitions, and sends the other partition to the server. The server is only responsible for coordination; it does not do verification itself. It accumulates the partitions (represented as a set of splitting decisions) coming in from the clients and farms them off to idle clients for verification. Clients can split multiple times. This continues until a client reports a counterexample (in which case, it must be a counterexample in the original program) or the server runs out of partitions and all clients become idle (in which case, the BMC problem is concluded as *safe*).

The splitting rate is adjusted according to the current number of idle client: it is reduced when all clients are busy, and then increased as more clients becomes available.

Splitting has some challenges that we illustrate using the following snippet of code.

```
procedure main() {
  var x := 0;
  if (...) { call foo(); x := 1; }
  if (...) { call bar(); }
  if (...) { call baz(); }
  assert(x == 1 || expr);
}
```

Suppose that the assertion at the end of `main` is the one that we wish to verify (or find a counterexample) and all uses of variable `x` are shown in the snippet. The `main` procedure calls multiple other procedures, each of which can manipulate global variables of the program (not shown). In this case, if we split on the call to `foo`, then one partition (the one that must take `foo`) becomes trivial: it is easy to see that the assertion

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_11

holds in that partition, irrespective of what happens in the rest of the program. We refer to this as a *trivial* split. Each split incurs an overhead when a partition is shipped to another client where the verification context for that partition must be set up from scratch. Trivial splits are troublesome because they accumulate this overhead without any real benefits in trimming down the search. Unfortunately, it is hard to avoid trivial splits altogether because it can involve custom (solver specific) reasoning (e.g., the fact that variable `x` is not modified outside of `main`). Our technique instead aims to reduce the overhead with splitting when possible. The server prioritizes sending a partition back to the client that generated it. Each client uses the incremental solving APIs of SMT solvers to remember backtracking points of previous splits that it had produced. This allows a client to get setup for one of its previous partitions much faster, thus reducing overhead.

Next, consider splitting on the call to `bar`. In this case, both of the generated partitions must still reason about `baz` because taking or avoiding `bar` has no implications on the call to `baz`. If `bar` turns out to be simple, while most of the complexity lies inside `baz`, then both partitions will end up doing the same work and diminish the benefits of parallelization. In this case, we rely on extracting information from the solver (via an unsat core) to make informed splitting choices and avoid duplicating work across partitions.

Implementation. We have implemented our technique in a tool called HYDRA[1]. The sequential BMC technique used by HYDRA is *stratified inlining* (SI) [3]. SI incrementally builds the VC of a program by lazily inlining procedure calls. HYDRA keeps track of the expanding VC, and frequently splits it by picking a splitting node that has already been inlined in the VC.

We evaluated HYDRA on Windows Device Driver benchmarks obtained using the Static Driver Verifier [4], [5]. These benchmarks extensively exercise the various features of C such as heaps, pointers, arrays, bit-vector operations, etc. [6] and collective require more than 11 CPU days to verify in a sequential setting.

The contributions of this paper are as follows:

- We propose a distributed design to enable solving large verification problems on a cluster of machines (Section IV-A and Section IV-B);
- We design a *proof-guided* splitting strategy that enables a lazy, semantic division of the verification task (Section III-B and Section IV-C);
- We implemented our design in a tool called HYDRA that achieves a 20× speedup on 32 clients, solving 30% additional benchmarks on which the sequential version timed out (Section V).

The rest of the paper is organized as follows. Section II covers background on VC generation and the stratified inlining algorithm. Section III discusses on how the search is decomposed for parallel exploration while Section IV presents the design of HYDRA. Section V presents an evaluation of HYDRA and Section VI discusses related work.

[1]HYDRA is available in the *hydra* branch of https://github.com/boogie-org/corral.git.

```
procedure main() {
  int x, y, z; bool c;
  L0: goto L1, L2;
  L1: assume c;
      call foo(x,z);
      goto L3;
  L2: assume !c;
      call bar(x,z);
      goto L3;
  L3: call baz(y);
      goto L4;
  L4: assume z != 0
      return;
}
procedure baz(int y) {
  L10: assume y == 3;
       return;
}

procedure foo(int x, int z) {
  bool d;
  L5: goto L6, L7;
  L6: assume d;
      assume z == x + 1;
      goto L8;
  L7: assume !d;
      assume z == x - 1;
      goto L8;
  L8: return;
}

procedure bar(int x, int z) {
  L9: assume z == x + 5;
      return;
}
```

Fig. 1: An Example of a Passified Program

II. BACKGROUND

We describe our techniques on a class of *passified* imperative programs. Such a program can have multiple procedures. Each procedure has a set of labelled basic blocks, where each block contains a list of statements followed by a **goto** or a **return**. A statement can only be an **assume** or a procedure **call**. A procedure can have any number of formal input arguments and local variables. Local variables are assumed to be non-deterministically initialized, i.e., their initial value is unconstrained. An **assume** statement takes an arbitrary expression over the variables in scope. An example program is shown in Figure 1. A **goto** statement takes multiple block labels and non-deterministically jumps to one of them.

Passified programs do not have global variables, return parameters of procedures, or assignments. These restrictions are without loss of generality because programs with these features can be easily converted to a passified program [7]; such conversion is readily available in tools like BOOGIE [8]. We also leave the expression syntax unspecified: we only require that expressions can be directly encoded in SMT. Our implementation uses linear arithmetic, fixed-size bit-vectors, uninterpreted functions, and extensional arrays. This combination is sufficient to support C programs [6], [9].

We aim to solve the following safety verification problem: given a passified program P, is the end of `main` reachable, i.e., is there an execution of `main` that reaches its **return** statement? This question is answered YES (or UNSAFE) by producing such an execution and the answer is NO (or SAFE) if there is no such execution. Furthermore, we only consider a *bounded* version of problem where P cannot have loops or recursion. (In other words, loops and recursive calls must be unrolled up to a fixed depth.) This problem is decidable with NEXPTIME complexity [7]. We next outline VC generation for single-procedure (Section II-A) and multi-procedure (Section II-B) programs.

A. VC generation for a single procedure

Let $p(\vec{x})$ be a procedure that takes a sequence of arguments $\vec{x}$. Further, assume that p does not include procedure calls. In that case, we construct a formula $VC(p)(\vec{x})$ such that p has a terminating execution starting from arguments $\vec{c}$ if and only if $VC(p)(\vec{c})$ is satisfiable.

The VC is constructed as follows. For each block labelled l, let b_l be a fresh Boolean variable and i_l be a unique integer constant. Let *succ*(l) be the set of successor blocks of l (mentioned in the **goto** statement at the end of block l, if any). Further, let e_l be a conjunction of all assumed expressions in the block. Let φ_l be $(b_l \Rightarrow e_l)$ if the block l ends in a return statement, otherwise let it be:

$$b_l \Rightarrow (e_l \wedge \bigvee_{s \in succ(l)} (b_s \wedge (i_s == f(i_l)))) \quad (1)$$

where f is an uninterpreted function $\mathbb{Z} \to \mathbb{Z}$ called the *control-flow function*.

The variables b_l are collectively referred to as *control variables*. Intuitively, b_l is *true* when control reaches the beginning of block l during the procedure's execution. The constraint φ_l means that if the control reaches block l, then it must satisfy the assumed constraints on the block (e_l) and pick at least one successor block to jump to. The function f records the chosen successor for each block.

Let l_p be the label of the first block of p (where procedure execution begins). Let *blocks*(p) be the set of block labels in p. Then, $VC(p)$ is $b_{l_p} \wedge \bigwedge_{l \in blocks(p)} \varphi_l$. If the VC is satisfiable, then one can read-off the counterexample trace from a satisfying assignment by looking at the model for f. As an example, the VC of procedure `foo` of Figure 1 is given in Figure 2.

The arguments of a procedure are its *interface* variables and we make these explicit in the VC. For instance, we will write $VC(\texttt{foo})(x, z)$ to make it explicit that x and z are the interface variables (free variables) and the rest of the variables are implicitly existentially quantified.

B. Stratified Inlining

Inlining all procedure calls can result in an exponential blowup in program size. For that reason, the *stratified inlining* (SI) algorithm [3] constructs the VC of a program in a lazy fashion. For ease in description, assume that each block can have at most one procedure call. For a procedure p, let pVC(p), called the *partial VC*, be the VC of the procedure constructed as described in the previous section where each procedure call is replaced with an "**assume** *true*" statement.

Given that programs can only have assume statements, the partial VC of a procedure represents an over-approximation of the procedure's behaviors, one where it optimistically assumes that each callee simply returns. Similarly, for a procedure p, if we replace each call with an "**assume** *false*" statement, then we get an under-approximation of p. The VC of this under-approximation can be obtained by setting the control variables b_l to false for each block l with an "**assume** *false*" statement. For instance, pVC(`main`) is an over-approximation

$$\begin{aligned} VC(\texttt{foo}): \quad & b_{L5} \\ \wedge \quad & b_{L5} \implies (b_{L6} \wedge f(5) == 6) \vee (b_{L7} \wedge f(5) == 7) \\ \wedge \quad & b_{L6} \implies d \wedge z == x + 1 \wedge b_{L8} \wedge f(6) == 8 \\ \wedge \quad & b_{L7} \implies \neg d \wedge z == x - 1 \wedge b_{L8} \wedge f(7) == 8 \\ \wedge \quad & b_{L8} \implies true \end{aligned}$$

$$\begin{aligned} \text{pVC}(\texttt{main}): \quad & b_{L0} \\ \wedge \quad & b_{L0} \implies (b_{L1} \wedge f(0) == 1) \vee (b_{L2} \wedge f(0) == 2) \\ \wedge \quad & b_{L1} \implies c \wedge b_{L3} \wedge f(1) == 3 \\ \wedge \quad & b_{L2} \implies \neg c \wedge b_{L3} \wedge f(2) == 3 \\ \wedge \quad & b_{L3} \implies b_{L4} \wedge f(3) == 4 \\ \wedge \quad & b_{L4} \implies z \neq 0 \end{aligned}$$

Fig. 2: VCs of procedures `foo` and `main` from Figure 1

of `main` (shown in Figure 2), whereas the following is an under-approximation: $pVC(\texttt{main}) \wedge \neg b_{L1} \wedge \neg b_{L2} \wedge \neg b_{L3}$.

A *static callsite* is defined as the pair (l, p) that represents the (unique) call of procedure p in block l. For instance, `main` of Figure 1 has three callsites: (`L1`, `foo`), (`L2`, `bar`), (`L3`, `baz`). A *dynamic callsite* is a stack of static callsites that represents the runtime stack during a program's execution. We assume that `main` is always present at the bottom of the stack for any dynamic callsite. For instance, $[\texttt{main}, (\texttt{L1}, \texttt{foo})]$ represents the call stack where `main` executed to reach `L1` and then called `foo`.

For a procedure p, let *callsites*(p) be the set of static callsites in p. Given a static callsite s, and dynamic callsite c, let $s :: c$ be the dynamic callsite where s is pushed on the top of the stack c. SI can require to inline the same procedure multiple times. Suppose that a procedure p calls p' twice, once in block l_1 and once in block l_2. Dynamic callsites will help distinguish between the two instances of p': the first will have (l_1, p') on top of the stack and the latter will have (l_2, p') on top of the stack.

We must take care to avoid variable name clashes between different VCs as we inline procedures. For a dynamic callsite c and procedure p that is at the top of c, let pVC(p, c) be the partial VC of p (as described earlier in the section), however for the construction of the partial VC, we use globally fresh control variables (variables b_l of Equation 1), globally fresh block identifiers (constants i_l of Equation 1) as well as globally fresh instances for the local variables. In pVC(p, c), the argument c is only used for bookkeeping purposes: let *control-variable*(l, c) refer to the control variable used for block l when constructing pVC(p, c). If c is $(l, p) :: c'$, then let *control-variable*(c) be *control-variable*(l', c'). Similarly, if p' is called from procedure p in block l', then let *interface-variables*$((l', p') :: c)$ be the set of interface variables (actuals) for the call to procedure p' in block l' in pVC(p, c).

The SI algorithm is shown in Algorithm 1. The algorithm requires an SMT solver with the usual interface. We use the *Push* API to set a backtracking point and a *Pop* API that backtracks by removing all asserted constraints until a matching *Push* call. Further, we assume that a counterexample

Algorithm 1: The Stratified Inlining algorithm.

Input: A Program P with starting procedure `main`
Input: An SMT solver $\mathcal{S}$
Output: SAFE, or UNSAFE(τ)

```
1  C ← {[main, s] | s ∈ callsites(main)}
2  S.Assert(pVC(main, [main]))
3  while true do
4      outcome ← SISTEP(P, C, S)
5      if outcome == SAFE ∨ outcome == UNSAFE(τ) then
6          return outcome
7      else
8          let NODECISION(_, _, C') = outcome
9          C ← C'
```

Algorithm 2: SISTEP(P, C, $\mathcal{S}$)

Input: A Program P, a set of callsites C
Input: An SMT solver $\mathcal{S}$
Output: SAFE, UNSAFE(τ), NODECISION(uc, I, C)

```
1  // Under-approximate check
2  S.Push()
3  forall c ∈ C do
4      S.Assert(¬control-variable(c))
5  if S.Check() == SAT then
6      return UNSAFE(S.Model())
7  else
8      uc ← S.UnsatCore()
9  S.Pop()
10 // Over-approximate check
11 if S.Check() == UNSAT then
12     return SAFE
13 else
14     τ ← S.Model()
15     I ← C ∩ callsites(τ)
16     C' ← ∅
17     forall c ∈ I do
18         C' ← INLINE(c)
19     C ← (C − I) ∪ C'
20     return NODECISION(uc, I, C)
```

trace can be extracted from a model returned by the solver.

The algorithm works by iteratively refining over-approximations of the program (in hope of getting an early SAFE verdict) and under-approximations of the program (in hope of getting an early UNSAFE verdict). Both these approximations are refined by inlining procedures.

Line 1 initializes a set C of *open* dynamic callsites. This set represents procedure calls that have not been inlined yet. The partial VC of `main` is asserted on the solver in Line 2.

SI, then, iteratively calls the SISTEP routine (Algorithm 2) that returns one of three possible answers: conclusive verdicts SAFE or UNSAFE, or an inconclusive verdict NODECISION.

The SISTEP routine is shown in Algorithm 2. It does an under-approximate check (Line 5) by assuming that calls at each of the open callsites cannot return (Line 4). If it finds a counterexample trace, SI returns UNSAFE, along with the model that can be used to construct the trace. This trace is guaranteed to only go through inlined procedure calls because

Algorithm 3: INLINE(c, $\mathcal{S}$)

Input: A dynamic callsite c, An SMT solver $\mathcal{S}$
Output: A set of open callsites C'

```
1  let (l, p) :: c' = c
2  S.Assert(control-variable(c) ⟹
       pVC(p, c)(interface-variables(c)))
3  C' ← C' ∪ {s :: c | s ∈ callsites(p)}
4  return C'
```

SISTEP	Action	Open Callsites	Inlined Callsites
Step-0	Assert pVC(`main`)	[`main`, (`L1`,`foo`)], [`main`, (`L2`,`bar`)], [`main`, (`L3`,`baz`)]	[`main`]
Step-1	Underapprox check: `UNSAT` Overapprox check: `SAT` Assert pVC(`foo`) Assert pVC(`baz`)	[`main`, (`L2`,`bar`)]	[`main`, (`L1`,`foo`)] [`main`, (`L3`,`baz`)]
Step-3	Underapprox check: `SAT` Return UNSAFE	[`main`, (`L2`,`bar`)]	

TABLE I: Execution of SI on the program of Fig. 1

all the open ones were blocked. Ignore the call to gather the unsat core shown on Line 8 for now; we use this information in the next section.

Next, SISTEP does an over-approximate check (Line 11). If this is UNSAT, then SI returns SAFE. If the check was satisfiable, then we construct the counterexample trace from the model provided by the solver (Line 14). This trace is guaranteed to go through at least one open call site (because the under-approximate check was UNSAT). The SI algorithm proceeds to inline the procedures called at each of the open callsites that the trace goes through. Such callsites are recorded in variable I (Line 15); these get returned for bookkeeping purposes (used in the next section). Callsites in I are inlined by asserting the partial VC of the callee, as shown in Line 2 in Algorithm 3. Read the asserted constraint as follows: if the control variable of the calling block is set to *true* then the VC of the procedure must be satisfied. The use of *interface-variables* ensures that formals are substituted with actuals for the procedure call. New callsites that are created as a result of the inlining are recorded in C' and then eventually added back to C (Line 19). Finally, SISTEP returns NODECISION back to SI with the set of callsites that it inlined, and the process repeats. An example illustrating the execution of SI is shown in Table I.

Define a *call tree* to be a (prefix-closed) set of dynamic callsites that represents all dynamic callsites that have been inlined by the SI algorithm at any point in time. We call this set as a tree because it can be represented as an unfolding of the program's call graph.

III. SPLITTING THE SEARCH

HYDRA employs a *decomposition*-based strategy to achieve parallelism. During the course of execution of the SI algorithm, HYDRA *splits* the current verification task by picking a dynamic callsite c that has already been inlined by SI. This generates two *partitions*: one that requires executions to pass through c (referred to as the *must-reach* partition), and the other that requires executions to avoid c (referred to as the

must-avoid partition). This strategy provides for an exhaustive and *path-disjoint* partitioning of the search space.

Formally, a *partition* is a pair (T, D) where T is a call tree (i.e., set of inlined callsites) and D is a set of *decisions* (either *must-avoid*(c) or *must-reach*(c) for $c \in T$). As a notation shorthand, for a partition $\rho = (T, D)$ and callsite c, let $\rho + c$ be the partition $(T \cup \{c\}, D)$. Similarly, for a decision d, let $\rho + d$ be the partition $(T, D \cup \{d\})$. Further, let *calltree*$(\rho) = T$ and *decisions*$(\rho) = D$. One can also see the above strategy as dividing the proof obligation (correctness theorem) on the complete program into a set of *lemmas* corresponding to each of the partitions.

This section addresses two primary concerns: (a) how to enforce splitting decisions during search? (Section III-A), and (b) how to choose a callsite for splitting? (Section III-B).

A. Encoding splitting decisions in SI as constraints

The constraint for *must-avoid*(c) is relatively straightforward. It is simply $\neg$*control-variable*(c). Asserting this constraint any time after SI has inlined c will ensure that control cannot go through c, thus SI will avoid c altogether.

We next describe the encoding of the *must-reach* constraint by first looking at the single-procedure case. For a procedure p, we introduce *must-reach* control variables r_l, one for each basic block l of p. Intuitively, setting r_l to *true* should mean that control must go through block l. Recall from Section II that the VC of a procedure uses i_l as a unique integer constant for block l and f as the control-flow function. We define *must-reach*(p) as the following constraint:

$$\bigwedge_{l \in blocks(p)} (r_l \Rightarrow \bigvee_{n \in pred(l)} (r_n \wedge f(i_l) == i_n)) \qquad (2)$$

This constraint enforces that if a block l must be reached, then one of its predecessors must be reached. The use of the control-flow function ties this constraint with the procedure's VC. For any block l, asserting $r_l \wedge$ *must-reach*(p), in addition to the VC of p will enforce the constraint that control *must* pass through block l. The proof is straightforward and we omit it from this paper.

For multi-procedure programs, we construct the *must-reach* constraint inductively. Let *must-reach*(p, c) be the constraint *must-reach*(p), but where the block identifiers $\{i_l\}$ are the same as the ones used in pVC(p, c). We construct *must-reach*(c) inductively over the length of c. If $c = [\texttt{main}]$, then *must-reach*(c) is *true*. Otherwise, if $c = (l, p) :: c'$, then *must-reach*(c) is $r_l \wedge$ *must-reach*$(p, c') \wedge$ *must-reach*(c').

B. Choosing a splitting candidate

Given an unsatisfiable formula Φ, expressed as a conjunction set of clauses $\{\phi_i\}$, a *minimal unsatisfiable core* (*min-unsatcore*) is a subset of clauses $\Psi \subseteq \Phi$ whose conjunction is still unsatisfiable and every proper subset of Ψ is satisfiable.

Consider the under-approximate check made by SI (Line 5 of Algorithm 2) where it blocks open-callsites and attempts to find a counterexample in the currently inlined portion of the program. This check is a conjunction of constraints, passed via $\mathcal{S}$.`Assert`, of two forms. First is the (partial) VCs of inlined callsites (Line 2 of Algorithm 3) and second is the blocked open callsites (Line 4 of Algorithm 2). If the check is unsatisfiable, then we extract its *min-unsatcore* and represent it as a set of callsites uc (that may be inlined or may be open). The set uc represents the current proof of safety of the program. Inlined callsites that are not part of uc are deemed *search-irrelevant* because whether they were inlined or not is immaterial to conclude safety of the program (at this point in the search). Formally, those callsites could have been left open (i.e., over-approximated) and the check would still be unsatisfiable. Therefore, the solver is likely to spend its energy searching and expanding the uc portion of the calltree as the search proceeds further. Consequently, we restrict splitting to a callsite chosen from uc so that we split where the search complexity lies.

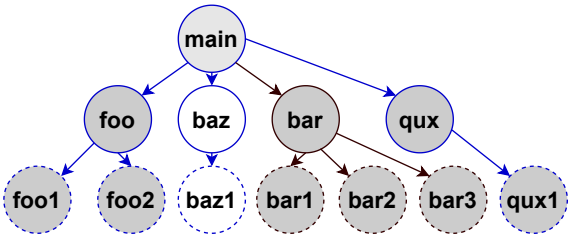


Fig. 3: Proof-guided splitting

Consider the inlining tree shown in Figure 3, where the open callsites appear as dotted circles and the inlined ones are shown as solid circles; the shaded nodes are the callsites that appear on the *min-unsatcore* (uc). In this case, both `baz` and `baz1` are ruled out for falling outside uc. If we pick some other callsite to split, say `qux`, then the *must-reach*(`qux`) partition of that split is likely to search in the subtree rooted at `qux`, whereas the *must-avoid*(`qux`) partition will search the uc portion excluding the subtree rooted at `qux`. We use a simple heuristic that roughly balances these partitions. Let the current inlined calltree be T and let `subtree`(T, c) be the subtree rooted at c. We choose the splitting callsite as the one that has maximum number of relevant callsites in its subtree (excluding `main` because that would be a trivial split). Formally, the splitting callsite is:

$$\underset{c \in uc}{\texttt{argmax}}\{|\texttt{subtree}(T, c) \cap uc|\}$$

In our example, we will pick `bar` for splitting.

We note that this choice of balancing the partitions is just a heuristic. In general, there may be dependencies between callsites. For instance, blocking one callsite can block others or make others be *must-reach* because of control-flow dependencies in the program. Our heuristic does not capture these dependencies. Furthermore, in our implementation, we do not insist on obtaining a *minimal* unsat core in order to reduce the time spent in computing it. Solvers generally provide a best-effort unsat core minimization (e.g., the `core.minimize` option in Z3).

Algorithm 4: Client-side verification algorithm

```
Input: A Program P
Input: An SMT solver S
1 while true do
2     ρ ← SendSync(GET_PARTITION)
3     outcome ← VERIFY(P, ρ, S)
4     SendAsync(OUTCOME, outcome)
```

IV. HYDRA DESIGN AND IMPLEMENTATION

HYDRA employs a client-server distributed architecture with a single server and multiple clients. The server (Section IV-B) is responsible for coordination while verification happens on the clients (Section IV-A). A client can decide to split its current search, at which point it sends one partition to the server while it continues on the other partition. If a client finishes its current search with a SAFE verdict, it contacts the server to borrow a new partition and starts solving it.

A. Client Design

All clients implement Algorithm 4. We use $SendSync$ as a message-response interaction with the server. $SendAsync$ is the asynchronous version where a message is sent to the server but a response is not expected. A client repeatedly requests the server for a partition (Line 2), solves it (Line 3) and sends the result back to the server on completion. Each client uses its own dedicated SMT solver ($\mathcal{S}$) for verification.

VERIFY (Algorithm 5) maintains a stack of decisions $dstack$ and a set of open callsites C. It starts off by preparing the input partition (Lines 3 to 7): it inlines the calltree of ρ and asserts all its splitting decisions. The client then enters a verification loop (Line 8) that repeatedly uses SISTEP (Line 9) to expand its search. If a counterexample is found (Line 10), the client returns an UNSAFE verdict back to the server. If SISTEP returns NODECISION, it implies that some more procedures were inlined but the search remained inconclusive; in this case, we perform the necessary bookkeeping on the set of currently open callsites (C'), new procedures inlined (I), and the minunsatcore from the unsat query (uc').

If SISTEP returned SAFE, then the search on the current partition has finished and the client must pick another partition to solve. This is done by returning the SAFE verdict (Line 22). The check on Line 15 is an optimization that we describe later in this section.

After checking the outcome of SISTEP, the client decides if it is time to split its search. This is referred to abstractly as "*TimeToSplit*" on Line 23: the exact time is communicated by the server to client (see Section IV-C). For splitting, the client picks a callsite c in accordance with our proof-guided splitting heuristic (from Section III-B) using the stored unsatcore uc. We note that the correctness of our technique does not rely on when a split happens or what splitting callsite is chosen. Therefore, these decisions can be guided by heuristics and tuned to optimized performance.

After splitting, the client continues along the partition with the MUSTAVOID(c) decision (let's call this partition ρ_1). The other partition (ρ_2) is sent to the server (Line 27). Note further that on Line 25, the client creates a backtracking point that is *just before* the decision on c is asserted. This backtracking point is exploited in Lines 15 to 20. When the client finishes search on ρ_1, it pings the server to know if ρ_2 has already been solved by a different client or not. If not, it simply backtracks the solver state and asserts the flipped decision MUSTREACH(c) to immediately get set up for search on ρ_2. This way, the client avoids the expensive setup of initializing a new partition. Because splitting can happen multiple times, the loop on Line 19 is necessary to follow along the recorded stack of decisions.

Algorithm 5: VERIFY(P, ρ, $\mathcal{S}$)

```
Input: A Program P, A partition ρ of P, A solver S
Output: SAFE, or UNSAFE(τ)
1  S.reset(), dstack ← [ ], C ← ∅  uc ← ∅
2  // Setup input partition
3  forall c ∈ calltree(ρ) do
4      C' ← INLINE(c), C ← (C − {c}) ∪ C'
5  forall d ∈ decisions(ρ) do
6      if d == MUSTAVOID(c) then S.Assert(must-avoid(c))
7      if d == MUSTREACH(c) then S.Assert(must-reach(c))
8  while true do
9      outcome ← SISTEP(P, C, S)
10     if outcome == UNSAFE(τ) then
11         return outcome
12     else if outcome == NODECISION(uc', I, C') then
13         uc ← uc', C ← C', ρ ← ρ + I
14     else
15         if SendSync(POP)==YES then
16             repeat
17                 let d(c) :: ds = dstack
18                 S.Pop(), dstack ← ds, ρ ← ρ − d(c)
19             until d == MUSTAVOID
20             S.Push(), S.Assert(must-reach(c)),
                dstack ← MUSTREACH(c) :: dstack,
                ρ ← ρ + MUSTREACH(c)
21         else
22             return outcome
23     if TimeToSplit then
24         c ← choose(calltree(ρ), uc)
25         S.Push()
26         S.Assert(must-avoid(c))
27         SendAsync(SEND_PARTITION,
            ρ + MUSTREACH(c))
28         dstack ← MUSTAVOID(c) :: dstack,
            ρ ← ρ + MUSTAVOID(c)
```

B. Server Design

We assume that each client has an associated unique identifier. Each message coming from a client is automatically tagged with the client's identifier. The server maintains two data structures. The first is an array Q of double-ended queues. The queue $Q[id]$ stores all partitions produced by client id. The second is a queue wt of clients that are currently idle.

The server processes incoming messages as follows. On receiving the message $\langle$SEND_PARTITION, $\rho\rangle$ from client id, it

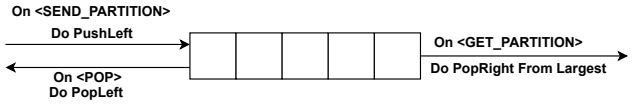


Fig. 4: Maintaining the double-ended queues

does a `push-left` to insert ρ into $Q[id]$. (The manipulation of Q is depicted in Figure 4.) This ensures that later partitions (which have a larger number of decisions and a larger call tree) from a particular client id appear on the left of $Q[id]$.

On receiving message $\langle$`GET_PARTITION`$\rangle$ from client id, the server needs to reply with a partition because id has just become idle. If all queues $Q[i]$ are empty, then id is inserted into wt and the client is kept waiting for a reply. Otherwise, the server picks the longest queue $Q[i]$, does a `pop-right` and replies to the client. This strategy attempts to avoid skew in queue sizes. Further, the rightmost partition is the smallest in that queue, which minimizes the setup time for that partition for the client that will get it. As more partitions are reported to the server (via a `SEND_PARTITION`), the server loops through wt, replying to as many idle clients as possible with partitions popped-right from the currently longest queue.

The message $\langle$`POP`$\rangle$ from client id implies that the client wishes to backtrack to its previously reported partition. Because reported partitions are pushed-left, and other clients (on `GET_PARTITION`) steal from the right, the previously reported partition from client id is exactly the leftmost one in $Q[id]$, if any. Thus, the server replies `YES` back to the client if $Q[id]$ is non-empty, followed by a `pop-left`. Otherwise, the server replies `NO`.

The server additionally listens to `OUTCOME` messages. If any client reports UNSAFE, all clients are terminated and the `UnSafe` verdict is returned to the user. The server returns SAFE verdict to the user when all queues in Q are empty and all clients are idle (i.e., wt consists of all clients).

Our design of the work-queue Q, as an array of sorted (by size) work-queues, is in contrast with using a centralized queue that is standard in classical work-stealing algorithms. It is useful for avoiding skew in queue sizes, distributing smaller partitions first, and enabling the client-backtracking optimization.

C. Adaptive rate of splitting

While a low splitting rate inhibits parallelism, a high rate increases the partition-initialization overhead on the clients. HYDRA uses a dynamic split-rate determined by the number of idle clients and the number of partitions available at the server. Each client maintains a *split time interval* δ (in seconds) and splits the search ("*TimeToSplit*" of Algorithm 5), if δ seconds have elapsed since the last split. The value of δ starts as a constant δ_c and is updated by the server as follows:

$$\delta_i = \begin{cases} \frac{Q[i].count}{wt.count} \times \delta_c & \text{if } wt.count \neq 0 \\ K \times \delta_c, & \text{otherwise.} \end{cases} \quad (3)$$

In the first case, a client's splitting is slowed down in proportion to its queue size (divided by the number of idle clients). The second case applies when there are no idle clients. Increasing δ by a factor of K reduces the rate of splitting drastically. We use $\delta_c = 0.5s$ and $K = 20$ in our experiments.

V. EXPERIMENTAL RESULTS

We evaluated HYDRA on SDV benchmarks [10]. SDV is used by Windows driver developers to statically check various rules on correct usage of kernel APIs in a driver. SDV comes packaged with a set of rules[2] that typically establish that kernel APIs are called in the correct temporal sequence; for instance, that a lock must be released before it can be acquired again.

The SDV benchmarks are obtained from a run of SDV on set of real-world device drivers that exercise all features of the C language: loops and recursion (up to a bounded depth), pointers, arrays, heap, bit-vector operations, etc. Each instance in the benchmark suite is a device driver paired with one of the SDV rules, i.e., it checks for the correct usage of the rule in the driver. SDV compiles the drivers, instruments the property and produces a program in Boogie [8]. The process of compilation to Boogie has been described in detail in previous work [6]. Each Boogie program has a well-defined entry point that is annotated with the tag {`:entrypoint`} and multiple assertions. The verification objective is to find an execution that starts at the entry procedure and ends with an assertion failure. Note that although these benchmarks are all compiled from C, HYDRA itself is source-language agnostic and can accept Boogie programs obtained from any source language.

We compared the performance of HYDRA against CORRAL [3] that implements the sequential Stratified Inlining algorithm. CORRAL forms a good baseline because it has been optimized heavily for SDV over the years [6].

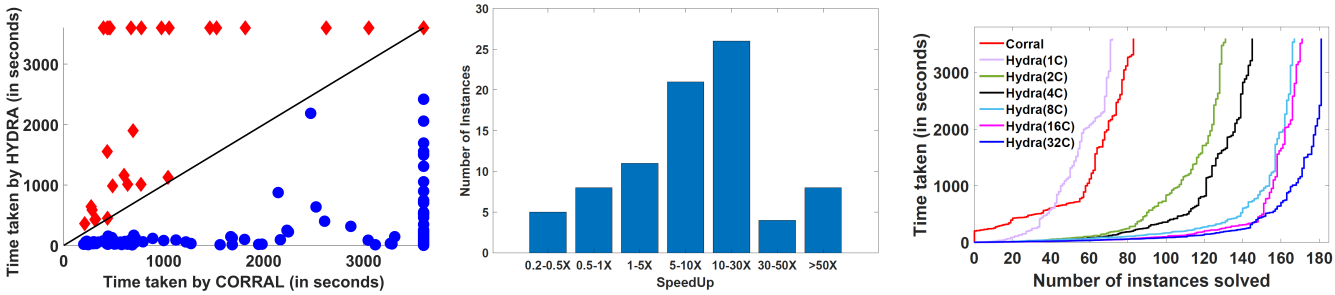
We only selected hard benchmarks (where CORRAL took at least 200 seconds to solve or timed out). We ran HYDRA with 32 clients. Timeout was set to 1 hour. We conducted our experiments with the server running on one machine (16 core, 64 GB RAM) and the 32 clients running on another machine (72-core with Intel Xeon Platinum 8168 CPU and 144 GB RAM), communicating via HTTP calls. As clients never communicate amongst themselves, this setup is equivalent to running clients on different machines.

Both CORRAL and HYDRA use Z3 [11] as the underlying SMT solver. While we used the default setting of a fixed random seed for Z3, we verified that the results reported here do not depend on the random seed. In fact, the behavior of the SI algorithm, which underlies both CORRAL and HYDRA, is not impacted by the choice of the random seed in any statistically significant way.

A. HYDRA *versus* CORRAL

Instances Solved. There were a total of 333 programs. HYDRA solved 99 instances (30%) on which CORRAL timed out (34 of these were SAFE and the rest 65 were UNSAFE). Conversely, CORRAL solved 12 (4%) instances on which HYDRA timed out. We did not investigate these cases in detail;

[2]https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier



(a) Scatter plot of running times

(b) Histogram of speedup of HYDRA over CORRAL

(c) Cactus plot of instances solved

Fig. 5: Comparison of HYDRA against CORRAL on SDV benchmarks

in a practical scenario one can simply dedicate a single client to run CORRAL and get the best of both tools. Overall, HYDRA solved 183 (55%) instances while CORRAL solved only 96 (29%) instances. Interestingly, there were 138 instances (41%) that were unsolved by both HYDRA and CORRAL indicating the need for further improvements.

Verification Time. In terms of running time, HYDRA was significantly faster than CORRAL in most (84%) cases: Figure 5a shows the scatter plot of running times. Figure 5b is a histogram of the speedup of HYDRA over CORRAL. For example, there were 8 instances where HYDRA was more than $50\times$ faster than CORRAL. A small fraction of instances had slowdowns as well, but the worst among these was $0.2\times$, i.e., CORRAL was $5\times$ faster than HYDRA. Over all instances, the mean speedup is $20.4\times$ and median speedup is $9.7\times$. Speedup excludes cases in which one of the tools timed out.

Scalability. Figure 5c is a cactus plot illustrating the scalability of HYDRA with the number of clients. CORRAL is able to solve only 58 instances within 1000 seconds. Running HYDRA with only a single client results in worse performance than CORRAL (solves only 46 instances within 1000 seconds). However, the performance improves significantly with the number of clients (solves 166 instances with 32 clients within 1000 seconds).

B. Effectiveness of proof-guided splitting

Empirical Analysis. We define *dissimilarity* $\eta(i,j)$ of a client i with respect to client j as $1 - \frac{|\mathcal{L}_i \cap \mathcal{L}_j|}{|\mathcal{L}_i|}$, where $\mathcal{L}_i$, $\mathcal{L}_j$ denote the set of callsites that i and j have inlined, respectively, when HYDRA finishes. A high value of $\eta(i,j)$ implies that the clients did a different search. Note, however, that $\eta(i,j)$ will never be 1 because certain callsites (like `main`) will always need to be inlined by each client.

Across all benchmarks and all client pairs, the average dissimilarity value was 0.55. This indicates sufficient difference among the inlined calltrees across clients.

Statistical Analysis. We implemented a randomized splitting algorithm that (1) decides to split/not-to-split at each inlining step uniformly at random, (2) if it has decided to split, it selects the splitting call-site uniformly at random.

We ran this randomized splitting algorithm 5 times for each program and compared the minimum verification time of these 5 runs for each instance against that of HYDRA. Using the Wilcoxon Sign Rank test, we found that HYDRA is statistically better than the randomized splitting algorithm with a p-value of 0.0012, indicating that the performance of the splitting heuristic is not accidental.

C. Server optimizations

We measured the performance impact of the server-side queue implementation on HYDRA. We compared our double-ended queues Q from Section IV-B against a classical work-stealing queue implementation. Our implementation allowed HYDRA to complete on 40% more cases where using the classical version made HYDRA time out. Further, HYDRA's performance was 8.5 times faster when both implementations terminated with a verdict.

In terms of controlling the splitting rate, both the performance (p-value of 5.27×10^{-5}) and the number of splits (p-value of 5.43×10^{-33}) were found to be statistically better with split-rate feedback.

VI. RELATED WORK

Parallelizing SAT/SMT solvers. In contrast to parallelizing verification tasks, parallelizing SAT/SMT solvers has attracted wider attention. There have been two popular, incomparable [12], approaches to parallelizing satisfiability problems: portfolio-based techniques [13], [14], [15] and divide and conquer techniques (decomposition [16], [17] or partitioning [18], [19], [20], [21]). Portfolio-based strategies either run multiple different algorithms or multiple instances of a randomized algorithm. They tend to work well in the presence of heavy-tailed distribution of problem hardness.

Divide and conquer strategies are most similar to our work. They either use static partitioning, based on the structure of the problem [22], or dynamic partitioning [19] based on runtime heuristics. However, unlike partitioning on individual variables at the logical-level, we split at the program-level based on its call graph. In our setting, the VC of a program can be exponential in the size of the program. This makes it hard to directly use parallelized solvers; we must split even

before the entire VC is generated. Furthermore, parallelized solvers are still not as mainstream as sequential solvers. Using solvers as a black-box allows us to directly leverage continued improvements in solver technology

Parallelizing program verification. Saturn [23] is one of the earlier attempts at parallelizing program verification. Saturn performs a bottom-up analysis on the call graph, generating summaries of procedures in parallel. While the intra-procedural analysis of Saturn is precise, it only retains *abstractions* of function summaries, thus cannot produce precise refutations of assertions like BMC.

There have been attempts at parallelizing a top-down abstraction-based verifier [24] as well as the property-directed reachability (PDR) algorithm [25], [26], [22], [13] and k-induction [27], [28]. These all rely on the discovery of inductive invariants for proof generation, a fundamentally different problem than BMC. It would be interesting future work to study the relative speedups obtained for parallelization in these respective domains.

Closer to BMC, parallelization has been proposed by a partitioning of the control-flow graph [29]. This approach does static partitioning (based on program slicing) and does not consider procedures at all (hence, must rely on inlining all procedures). Further, it has only been evaluated on a single benchmark program. Our technique, on the other hand, performs dynamic partitioning, supports procedures and has been much more extensively evaluated.

In a recent work, Inverso et al. [30] propose a parallelization technique for the verification of concurrent programs by partitioning the verification task such that each partition considers a subset of the interleavings of the input program. Next, it uses sequentialization to generate a sequential program for each partition and then verifies the sequential program. The partitioning is static and done up-front. This work is complementary to HYDRA: it addresses the complexity arising from many interleavings, whereas HYDRA addresses complexity arising from many (sequential) procedures calling each other.

REFERENCES

[1] E. M. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and Verilog programs using Bounded Model Checking," in *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, 2003, pp. 368–371.

[2] D. Beyer, "Automatic verification of C and Java programs: SV-COMP 2019," in *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, 2019, pp. 133–155.

[3] A. Lal, S. Qadeer, and S. 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*, 2012, pp. 427–443, https://github.com/boogie-org/corral/.

[4] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg, "The Static Driver Verifier research platform," in *Computer Aided Verification*. Springer, 2010, pp. 119–122.

[5] Microsoft, "Static Driver Verifier," http://msdn.microsoft.com/en-us/library/windows/hardware/ff552808(v=vs.85).aspx.

[6] A. Lal and S. 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*, 2014, pp. 202–212.

[7] ——, "Reachability modulo theories," in *Reachability Problems - 7th International Workshop, RP 2013, Uppsala, Sweden, September 24-26, 2013 Proceedings*, 2013, pp. 23–44.

[8] M. Barnett, K. R. M. Leino, M. Moskal, and W. Schulte, "Boogie: An intermediate verification language," 2009, https://github.com/boogie-org/boogie/.

[9] S. K. Lahiri and S. Qadeer, "Back to the future: revisiting precise program verification using SMT solvers," in *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2008, pp. 171–182.

[10] Microsoft, "Static Driver Verifier Benchmarks," https://github.com/boogie-org/sdvbench.

[11] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[12] M. Marescotti, A. Hyvärinen, and N. Sharygina, "SMTS: Distributed, visualized constraint solving," in *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. EPiC Series in Computing, G. Barthe, G. Sutcliffe, and M. Veanes, Eds., vol. 57. EasyChair, 2018, pp. 534–542. [Online]. Available: https://easychair.org/publications/paper/k7BQ

[13] S. Chaki and D. Karimi, "Model checking with multi-threaded IC3 portfolios," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 517–535.

[14] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä, "Incorporating learning in grid-based randomized SAT solving," in *Artificial Intelligence: Methodology, Systems, and Applications*, D. Dochev, M. Pistore, and P. Traverso, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 247–261.

[15] C. M. Wintersteiger, Y. Hamadi, and L. Moura, "A concurrent portfolio approach to SMT solving," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV '09. Berlin, Heidelberg: Springer-Verlag, 2009, p. 715–720.

[16] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518.

[17] Y. Hamadi, J. Marques-Silva, and C. M. Wintersteiger, "Lazy decomposition for distributed decision procedures," *Electronic Proceedings in Theoretical Computer Science*, vol. 72, p. 43–54, Oct 2011. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.72.5

[18] H. Zhang, M. P. Bonacina, and J. Hsiang, "PSATO: a distributed propositional prover and its application to quasigroup problems," *Journal of Symbolic Computation*, vol. 21, no. 4, pp. 543 – 560, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0747717196900309

[19] R. Martins, V. Manquinho, and I. Lynce, "Improving search space splitting for parallel SAT solving," in *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, vol. 1, Oct 2010, pp. 336–343.

[20] M. Böhm and E. Speckenmeyer, "A fast parallel SAT-solver — efficient workload balancing," *Annals of Mathematics and Artificial Intelligence*, vol. 17, no. 2, pp. 381–400, Sep 1996. [Online]. Available: https://doi.org/10.1007/BF02127976

[21] B. Jurkowiak, C. M. Li, and G. Utard, "Parallelizing Satz using dynamic workload balancing," *Electronic Notes in Discrete Mathematics*, vol. 9, pp. 174 – 189, 2001, lICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S157106530400321X

[22] M. Marescotti, A. Gurfinkel, A. E. J. Hyvärinen, and N. Sharygina, "Designing parallel PDR," in *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '17. Austin, Texas: FMCAD Inc, 2017, p. 156–163.

[23] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the Saturn project," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007, pp. 43–48.

[24] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani, "Parallelizing top-down interprocedural analyses," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 217–228, 2012.

[25] A. R. Bradley, "SAT-based model checking without unrolling," in *Proceedings of the 12th International Conference on Verification, Model*

Checking, and Abstract Interpretation, ser. VMCAI'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 70–87.

[26] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, p. 125–134.

[27] T. Kahsai and C. Tinelli, "PKIND: A parallel k-induction based model checker," *EPTCS*, vol. 72, 11 2011.

[28] M. Blicha, A. Hyvärinen, M. Marescotti, and N. Sharygina, "A cooperative parallelization approach for property-directed k-induction," in *VMCAI*, 01 2020, pp. 270–292.

[29] M. K. Ganai and W. Li, "D-TSR: Parallelizing SMT-Based BMC using tunnels over a distributed framework," in *Haifa Verification Conference*. Springer, 2008, pp. 194–199.

[30] O. Inverso and C. Trubiani, "Parallel and distributed bounded model checking of multi-threaded programs," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 202–216.