Modular Bug Detection with Inertial Refinement

Nishant Sinha
NEC Research Labs, Princeton, USA

***Abstract*—Structural abstraction/refinement (SAR) [4] holds promise for scalable bug detection in software since the abstraction is inexpensive to compute and refinement employs pre-computed procedure summaries. The refinement step is key to the scalability of an SAR technique: efficient refinement should avoid exploring program regions irrelevant to the property being checked. However, the current refinement techniques, guided by the counterexamples obtained from constraint solvers, have little or no control over the program regions explored during refinement. This paper presents *inertial* refinement (IR), a new refinement strategy which overcomes this drawback, by *resisting* the exploration of new program regions during refinement: new program regions are incrementally analyzed only when no error witness is realizable in the current regions. The IR procedure is implemented as part of a generalized SAR method in the F-SOFT verification framework for C programs. Experimental comparison with a previous state-of-the-art refinement method shows that IR explores fewer program regions to detect bugs, leading to faster bug-detection.**

I. INTRODUCTION

Modular program analyzers [30], [28], [12], [6], [32], [31], [4], [7] that exploit the program structure are more scalable since they avoid repeated analysis of program regions by computing reusable summaries. Traditional modular methods [30], [28] target proofs of program assertions by computing and composing summaries in an intertwined manner. For example, to compute a summary for a function F, the methods need to compute and compose the summaries of all the callees of F, even if many of these callees are irrelevant to checking the property at hand. Recent methods based on structural abstraction/refinement (SAR) [4], [31], [12] alleviate this problem by dissociating summary composition from computation: function summaries and verification conditions [16] are first *computed locally* by skipping the analysis of callees (abstraction phase) and then *composed lazily* with callee summaries (refinement phase). Refinement is property-driven and employs an efficient constraint solver (e.g., [15], [13]) for the program logic. In contrast to other abstraction/refinement methods, e.g., predicate abstraction [17], [5], computing a structural abstraction is relatively inexpensive, and refinement is done incrementally via pre-computed function summaries. Owing to these advantages, several recent methods [31], [4], [2], [7] have exploited the idea of SAR for scalable bug detection.

By dissociating summary computation from composition, SAR has the ability to *select* which regions to explore during the refinement phase for checking properties efficiently. The selective refinement strategy determines the efficiency of an SAR-based verification method. Ideally, we desire an *optimal* strategy, which explores (composes with) exactly those program regions which are relevant to a given property. Optimal refinement is as hard as the (undecidable) program verification problem since it may require a knowledge of the complete program behavior for making a selection. Consequently, researchers employ heuristics [4], [31] for performing refinement, guided by the counterexamples obtained when the solver checks the abstract model [23], [10]. The solver, however, is oblivious of the program structure, and may produce spurious counterexamples that continuously drive the refinement towards newer program regions, even though a witness may exist undetected in the currently explored regions. Redundant refinement of this form burdens the solver with irrelevant summary constraints, leading to dramatic increase in solving times, and, in many cases, to the failure of an SAR-based method.

This paper presents a new structure-aware method, called *inertial refinement* (IR) to overcome this drawback. The IR method *resists* exploring new program regions during refinement, as much as possible, in hope of finding a witness *within* the currently explored regions. Given a program assertion A, our method computes an initial abstract *error condition* ϕ for violating A, by exploring program paths in a small set of regions relevant to A, while abstracting the other adjacent regions. To check if ϕ is feasible, IR first symbolically *blocks* all unexplored program regions involved in ϕ, by adding auxiliary constraints to the solver. This forces the solver to find witnesses to ϕ that avoid the unexplored regions. If such a witness exists, IR succeeds in avoiding the costly analysis of the unexplored regions. Otherwise, IR explores a *minimal* set of new program regions that may admit an error witness. The minimal set of regions are computed in a property-driven manner by analyzing the proofs of infeasibility inside the solver (based on the notion of *minimal correcting sets* [24]), which provide hints as to why the currently explored regions are inadequate for checking the property. IR has multiple advantages as a refinement method. IR improves the scalability of SAR-based methods by restricting search to a small set of program regions, leading to more *local* witnesses than other methods. Moreover, IR exploits the fact that most bugs can be detected by analyzing a small number of program regions [3], [26].

All previous methods based on SAR [31], [4], [2], [7] restrict structural abstraction to function boundaries. This paper proposes a *generalized* SAR scheme that may abstract (and later refine on-demand) arbitrary program regions, including loops. As a result, SAR can exploit the entire modular program structure to make a more fine-grained selection of regions to explore for checking properties efficiently. A consequence of this generalization is that we do not statically unroll loops and recursive functions for checking properties; they are dynamically unrolled in a property-driven manner by inertial refinement. The paper makes the following main contributions:

- We present a modular bug detection method based on a new *generalized* structural abstraction/refinement (SAR) approach, which fully exploits the modular structure of a program (functions, loops and conditionals) to perform an efficient analysis.
- We propose a new structural refinement method, called inertial refinement, which avoids exploring new program regions until necessary. The technique is property-guided and employs *minimal correcting sets* [24] produced by

```
1  int x,y;
2  void foo (int *p, int c) {
3   if (p == NULL)
4     x = c;
5   else x = bar (*p);
6   ...
7   assert (x > c);
8   ...
9   if (x == c)
10    y = neg(x);
11  else y = 0;
12  assert (y >= 0);
13 }
```

```
1 int neg (int a) {
2    if (a > 0) return -a;
3    else return a;
4 }
```

```
1 void loopf (int n) {
2    int i=0, j=0;
3    while (i < n) {
4      j = j + 2 * i;
5      i++;
6    }
7    assert (n >= 0 &&
8              j < 2*n);
9 }
```

Fig. 1: Motivating Examples. The complex function `bar` is not described.

constraint solvers [15], [13] to efficiently select new regions to explore.

- The SAR method with inertial refinement is implemented in the F-SOFT verification framework for C programs [19]. Experimental results on real-life benchmarks show that the method explores fewer regions than a state-of-the-art refinement technique [4], and outperforms the previous approach on larger benchmarks.

II. Overview

We illustrate the key ideas of inertial refinement for checking the function `foo` in Fig. 1: `foo` contains a call to a complex function `bar` (line 5) and two assertions at lines 7 and 12, respectively. Consider the assertion (say A) at line 7 in `foo`: to check this assertion, SAR first computes an *error condition* (EC) under which A is violated. This EC, say ϕ, represents the feasibility condition for all program executions in `foo` which terminate at A and violate A. To compute ϕ, our method explores `foo` locally (cf. Sec. III) by performing a precise data-flow analysis: a form of forward symbolic execution [20] with data facts being merged path-sensitively at *join* nodes [21], [3]. The analysis propagates data of form (ψ, σ) through `foo`: ψ is the path condition at the current program location (summarizing the set of incoming paths to the location symbolically) and σ is a map from program variables to their path-sensitive (symbolic) values at the current location.

To avoid exploring `bar` at line 5, the method performs structural abstraction of `bar` during propagation: the effect of `bar` is abstracted by a tuple $(\pi_b, [ret_{bar} \mapsto \lambda_{b,ret}])$, where the placeholder (essentially, a free variable) π_b abstracts the set of paths through `bar` symbolically, and the placeholder $\lambda_{b,ret}$ abstracts the return value of `bar`. For example, the value of x computed at line 6 (obtained by merging data from the branches of the conditional at line 3) is $x_1 = ite(p \neq 0 \wedge \pi_b,\ \lambda_{b,ret},\ c)$ and the path condition is $\psi' = ((p \neq 0 \wedge \pi_b) \vee (p = 0))$. The EC ϕ computed for A at line 7 (ψ' conjoined with the negated assertion) is $\phi = \psi' \wedge (x_1 \leq c)$. Note that ϕ depends on the two unconstrained placeholders π_b and $\lambda_{b,ret}$ corresponding to `bar`. Now, ϕ is checked with a constraint solver, e.g., [15], [13] using structural refinement. We will see how the placeholder π_b plays a crucial role to avoid exploring paths into `bar`.

Checking ϕ with the solver may return a witness (lines 2-5-6-7) that includes a call to the complex function `bar`. This witness relies on the abstraction of `bar` by π_b and $\lambda_{b,ret}$ and hence may be spurious, e.g., if `bar` returns a value always greater than c. To check if the witness is an actual one, refinement will expand π_b and $\lambda_{b,ret}$ with the corresponding precise summaries from `bar`. Note, however, that line 4 sets x to c, and hence an actual witness for ϕ exists inside `foo` (line 2-3-4-6-7) that does not require exploring `bar`. However, this is not apparent from ϕ syntactically and a naive SAR checker will perform spurious refinement by expanding both the placeholders.

More sophisticated refinement procedures may also succumb to spurious refinement. For example, the state-of-the-art structural refinement strategy (referred to as `DCR`) [4], [3] uses the satisfying model from the constraint solver to compute a set of irrelevant placeholders, to avoid expanding them subsequently. Since `DCR` is guided only by the structure-unaware solver, it may expand placeholders spuriously even if a witness exists in the current regions. For example, suppose the solver generates the following model for the EC ϕ above: $(p \neq 0)$ and $(\pi_b = true)$. `DCR` analyzes the expression for ϕ guided by this model and concludes that both π_b and $\lambda_{b,ret}$ are relevant to ϕ being satisfiable. Therefore, `DCR` must perform the costly expansion of both the placeholders. Similarly, another structural refinement procedure [31] driven only by models from a constraint solver may also explore `bar` when trying to concretize an abstract counterexample.

In contrast, our inertial refinement (`IR`) procedure (cf. Sec. IV) *resists* expansion and checks if a proof/witness to ϕ exists within the currently explored region. To this goal, the analysis *blocks* paths leading to the unexplored function `bar` by adding a constraint $\neg\pi_b$ to ϕ and then checks for a solution. If a solution is found, as in this case, the method is able to avoid the cost of a spurious refinement. Otherwise, `IR` selects a *minimal* set of new regions to explore, which may admit an error witness (cf. Sec. IV).

Most bug finding approaches [4], [9], [32] statically unroll the loops to a fixed depth, which may lead to several errors being missed. Although loops may be also handled as tail-recursive functions in SAR (as in [31]), conventional static analysis [11] seldom does so. We propose a structural abstraction specific to loops, so that inertial refinement corresponds to *dynamically* unrolling loop iterations in a property-driven manner (cf. Sec. IV-A). As a result, our method can check non-trivial assertions, e.g., the assertion at line 7 in the `loopf` function in Fig. 1 is violated only when $n \geq 3$.

III. Generalized Structural Abstraction

We start with describing our *generalized* structural abstraction, which forms the basis of our SAR method and may abstract arbitrary program *regions*, as defined below.

Program Regions. A program region R corresponds to a structural unit of the program syntax, i.e., a function body, a loop or a conditional statement. To formalize regions precisely, we view a sequential C program $\mathcal{P}$ as a hierarchical *recursive state machine* (RSM) M [1]. The RSM M consists of a set of *regions*: each region contains a control flow graph, which in turn consists of (i) a set of *nodes* (labeled by assignments), (ii) a set of *boxes* (each box is, in turn, mapped to a region), and (iii) control flow edges among nodes and boxes (labeled with guards). Each region also has special *entry* and *exit* nodes. An *unfolding* [1] of M is obtained by recursively inlining

SAR (Program $\mathcal{P}$)
$\mathcal{R}$:= Partition $\mathcal{P}$ into regions
foreach *region* $R \in \mathcal{R}$ **do**
    $(\psi_R, \sigma_R, \Phi_R)$:= LOCSUMMARIZE(R)
    Φ := HOIST(Φ_R)
    foreach $\phi \in \Phi$ **do**
        res := REF(ϕ)
        /* *Report witness if* res *is* SAT */
	REF(ϕ)
	while ϕ *contains placeholders* **do**
	    if CHECK (ϕ) = UNSAT **then**
	        return UNSAT
	    Pick a placeholder λ in ϕ
	    t = GETSUMMARY (λ)
	    ϕ := $\phi[\lambda \mapsto t]$
	return SAT

Algorithm 1: A generic modular analysis algorithm SAR.

each box by the corresponding region. An edge from a node to a box is said to be a *call edge*. A program region R_1 is said to *precede* another region R_2, if R_1 contains a box that maps to R_2, i.e., control flow enters R_2 on leaving R_1. We also say that R_2 *succeeds* R_1 in this case. We assume that assertions for property checking, e.g., dereference safety, array bound violations, etc., are modeled as special *error nodes* in the RSM M; the reachability of error nodes implies that the corresponding assertion is violated.

For example, the program fragment in Fig. 1 consists of the following top-level regions: function bodies `foo`, `neg` and `loopf`. Region `foo` contains two boxes mapped to *if-then-else* (conditional) regions C_1 (lines 3-5) and C_2 (lines 9-11); both regions succeed region *foo*. C_1 and C_2, in turn, contain boxes mapped to `bar` and `neg` function regions, respectively. Similarly, the `loopf` function region contains a box corresponding to the *loop body* region (lines 3-6). For ease of description, we will refer to an inlined instance of a region in a box as a region also. In the following, we use the standard program analysis terminology [30], [12], extended to RSM regions in a straightforward manner. In the following, we will assume that the regions corresponding to conditionals are inlined in the corresponding boxes; we will only differentiate between function and loop regions.

Side-effects. For each program region R, the *side-effects set* $\mathcal{M}(R)$ denotes the set of program variables that may be modified on executing R (together with its successors) under all possible calling contexts. The *inputs* to region R consist of the set of variables that are referenced in R. To compute side-effects for programs with pointers, we assume that the heap size is bounded (to handle dynamic allocation and recursive data structures) and employ a whole-program side-effect analysis [29], [31] to compute the side-effects.

Error Conditions. Given an error node eb in the program RSM and a set of paths T terminating at eb, the formula representing the feasibility condition for the set T is said to be an *error condition* (EC). In contrast to verification conditions (VCs) [16], which express sufficient conditions for existence of proofs, the satisfiability of ECs implies existence of assertion violations. We say that an EC ϕ has a witness, if ϕ has a satisfying solution; otherwise, we say that the EC has a proof. Note that an infinite number of ECs may be derived from a location eb (due to loops and recursion). Our *under-approximate* analysis checks only a finite subset of all the ECs and therefore, guarantees only the soundness of bugs detected; the proofs do not imply that eb is unreachable (cf. Theorem 1).

Structural Abstraction. Analyzing all program regions may neither be feasible for a given program analysis nor necessary for checking a given property. Structural abstraction enables a property-driven modular analysis of programs while avoiding the analysis of undesired regions, e.g., one or more nested successor regions of a region Q can be abstracted during analysis of Q. The structural abstraction of a region R is a tuple (π_R, σ_R), where (i) π_R is a *Skolem constant* (basically, a fresh variable) summarizing the paths in R and (ii) σ_R is a map with entries of form $(v \mapsto \lambda_{v,R})$, where $v \in \mathcal{M}(R)$ is a side-effect of R and $\lambda_{v,R}$ is a Skolem constant which models arbitrary modifications of v in R. In the following, the Skolem constants π_R and $\lambda_{v,R}$ are jointly referred to as *placeholder* variables. We also refer to placeholders of form π_R as π*-variables*. The set of placeholders in the range of σ_R are said to *depend* on π_R, and are denoted by $Dep(\pi_R)$. For example, the call to `bar` in the function `foo` in Fig. 1 (cf. Sec. II) is abstracted by the tuple $(\pi_b, [ret_{bar} \mapsto \lambda_{b,ret}])$, where $\lambda_{b,ret} \in Dep(\pi_b)$.

When the analysis encounters a call to R in a preceding region Q, it conjoins the placeholder π_R with the current path condition ψ, updates the current value map σ with σ_R, and continues analyzing Q. If R is later found relevant to an assertion in Q, the initial abstraction of R is refined on-demand. The abstraction has several advantages: first, it is cheap (computation of side-effects $\mathcal{M}(R)$ is done once for the whole program); second, it allows on-the-fly refinement using a summary of R, which is computed only once, and finally, it allows us to analyze program *fragments* in absence of the whole program. Note that our formalization generalizes the earlier approaches [31], [4] to handle all modular units of a program, i.e., functions, loops and conditionals, uniformly. As a result, SAR can perform a more fine-grained selection of program regions to explore when checking an EC.

Alg. 1 presents a generic modular algorithm SAR for checking assertions in a program $\mathcal{P}$, having these phases:

- The algorithm first partitions the program into a set of regions $\mathcal{R}$.
- For each region $R \in \mathcal{R}$, a procedure LOCSUMMARIZE is used to compute a *local* summary (by a forward data flow analysis over program expressions [21], [3] or using weakest preconditions [14], [16], [4]) while abstracting all the successor regions of R as above. The local summary $(\psi_R, \sigma_R, \Phi_R)$ consists of the predicate ψ_R summarizing the paths in R, the map σ_R summarizing the outputs (side-effects) of R in terms of symbolic expressions over inputs to R, and a set of error conditions (ECs) Φ_R which correspond to assertion violations in R.

- The ECs Φ_R are local to R; in order to find violating executions starting from the program entry function, these ECs are *hoisted* [3], [7], [16] to the entry function of the program by the HOIST procedure, which computes weakest preconditions of ECs with respect to a bounded set of calling contexts [30] to the region R. Note that HOIST may also use structural abstraction during backward propagation [3].
- Finally, the procedure REF is used to check each hoisted EC ϕ using structural refinement based on a constraint solver, e.g., an SMT solver [13], [15]. REF proceeds iteratively by choosing a placeholder λ in ϕ, *expanding* λ using its summary expression t (computed by the GETSUMMARY procedure), and checking if the resulting ϕ is satisfiable. The procedure REF terminates when the solver finds the EC ϕ unsatisfiable (UNSAT) or if ϕ does not contain any placeholders and is satisfiable (SAT).

In this paper, we assume a partition of the program into only function and loop regions, i.e., conditionals are inlined in the predecessor regions. The details of the LOCSUMMARIZE, GETSUMMARY and HOIST procedures can be found elsewhere [3], [21], [7], [16]; we will only concern ourselves with the REF procedure, which is the prime bottleneck for the SAR method.

SAR is an *under-approximate* analysis, i.e., it analyzes only a subset of all possible paths reaching an assertion violation. Hence, it can only detect bugs soundly (cf. Theorem 1). SAR can natively handle programs with arbitrary recursive functions and loops: however, it may not terminate if an unbounded number of iterations of `IR` are needed during the check.

Example 1. Recall the program fragment shown in Fig. 1. Our analysis first partitions the fragment into four regions: `foo`, `neg`, `loopf` functions, and the loop body region (lines 3-6 in `loopf`). The procedure LOCSUMMARIZE then summarizes each region, e.g., the summary of `foo` (shown below) consists of path and side-effect summaries, $\psi_{\texttt{foo}}$ and $\sigma_{\texttt{foo}}$, resp., and a set of ECs $\Phi_{\texttt{foo}}$. To summarize `foo`, the calls to `bar` and `neg` are abstracted by placeholder pairs (π_b, $\lambda_{b,ret}$) and (π_n, $\lambda_{n,ret}$) respectively.

$\psi_{\texttt{foo}}$	$(\psi_1 \wedge \psi_2)$ where $\psi_1 = (p = 0 \vee (p \neq 0 \wedge \pi_b))$, $\psi_2 = ((x_1 = c \wedge \pi_n) \vee (x_1 \neq c))$
$\sigma_{\texttt{foo}}$	$[x \mapsto x_1, y \mapsto y_1]$, where $x_1 = ite(p \neq 0 \wedge \pi_b, \lambda_{b,ret}, c)$ and $y_1 = ite(x_1 = c \wedge \pi_n, \lambda_{n,ret}, 0)$
$\Phi_{\texttt{foo}}$	$\{\Phi_1, \Phi_2\}$; $\Phi_1 = (\psi_1 \wedge x_1 \leq c)$, $\Phi_2 = (\psi_{\texttt{foo}} \wedge y_1 < 0)$

All the ECs are then hoisted to the entry functions (`foo` and `loopf` here): in this case, the ECs for `foo` are already hoisted. Finally, REF analyzes each EC ϕ in the entry function by iteratively checking ϕ and expanding placeholders.

Theorem 1: Let SAR compute an EC ϕ for an error location l after hoisting. If REF(ϕ) returns SAT, then there exists a true error witness to l.

Selective Refinement. In general, many placeholders in an EC ϕ are not relevant for finding a proof or a witness, and expanding them leads to wasteful refinement iterations along with an increased load on the solver. *Selective* refinement, therefore, focuses on selecting a subset of placeholders in ϕ that are relevant to the property. This allows REF to terminate early if there exist no relevant placeholders in ϕ. An additional benefit of selective refinement is that, in many cases, recursive programs can be analyzed without unbounded expansion of the placeholders. We now present a new strategy for selective refinement, called *inertial refinement*.

IV. INERTIAL REFINEMENT

The key motivation behind inertial refinement (`IR`) is to avoid exploring irrelevant regions during modular analysis, based on the insight that most violations involve only a small set of program regions. To this goal, `IR` first tries to find a witness/proof for an EC *inside* the program regions explored currently, say $\mathcal{R}$. If `IR` is unsuccessful, then $\mathcal{R}$ is inadequate for computing a witness or a proof. Therefore, `IR` augments $\mathcal{R}$ by a minimal set of successor program regions, which may admit a witness. The new regions are selected efficiently based on an analysis of why the current region set $\mathcal{R}$ is inadequate. In order to describe the details of `IR`, we first introduce the notion of *region blocking*.

Region blocking. Recall (cf. Sec. III) that SAR may abstract a region R (when analyzing a predecessor region Q) in form of a tuple (π_R, σ_R), where π_R is the path summary placeholder of R and σ_R maps output variables in R to unique placeholders. A *region blocking* constraint (π-constraint, in short) for a π-variable π_R is defined to be $\phi_\pi = \neg\pi_R$. Asserting ϕ_π when checking an EC ϕ in the region Q, forces the solver to find witnesses by *blocking* the program execution paths that lead from Q to R.

Figure 2 shows the `IR` procedure in form of a flow diagram. `IR` proceeds by iteratively adding or removing π-constraints, until the result is satisfiable (SAT) or unsatisfiable (UNSAT). In order to resist exploration of irrelevant regions, the procedure first asserts π-constraints (Φ_π) for all π-variables in the current EC ϕ. If ϕ remains satisfiable even after adding Φ_π, the procedure returns $true$, implying that a witness for ϕ exists that does not involve traversing the blocked regions. Otherwise (the constraints are UNSAT), a subset ϕ_π of π-constraints is computed, whose removal leads to a satisfiable solution. Note that the set ϕ_π corresponds to a set of blocked regions whose exploration may lead to the discovery of a concrete witness to ϕ. If the set ϕ_π is empty, then no witness for ϕ exists (see Theorem 2), and `IR` returns $false$. Otherwise, `IR` performs exploration of the regions corresponding to ϕ_π in the following way. First, the paths to the blocked regions are exposed by removing all π-constraints in ϕ_π. Then, `IR` refines ϕ by expanding the placeholders V_π in ϕ_π and their dependent placeholders $Dep(V_\pi)$ with the corresponding summary expressions (cf. Sec. III).

The key step in the `IR` procedure is that of computing $\phi_\pi \subseteq \Phi_\pi$ efficiently. To this goal, we employ the notion of a *correction set* (CS) of a set of constraints [24]: given an unsatisfiable set of constraints Ψ, a correction set ψ is a subset of Ψ such that removing ψ makes $\Psi \setminus \psi$ satisfiable. To obtain efficient inertial refinement, i.e., explore a small set of blocked regions, we are interested in a *minimal* correcting set (MCS), none of whose proper subsets are correction sets. The notion of correction sets is closely related to that of *maximal satisfiable subsets* (MSSs) [24], which is a generalization of the solution of the well-known Max-SAT problem [24]. An MSS is a satisfiable subset of constraints that is maximal, i.e., adding any of one of the remaining constraints would make it UNSAT. The *complement* of an MSS consisting of the remaining set of unsatisfied constraints is an MCS. For example, the UNSAT constraint set $((x), (\neg x \vee y), (\neg y))$ admits three MCSs, (x), $(\neg x \vee y)$, and $(\neg y)$, all of which are *minimum*. Note that many approaches utilize unsatisfiable cores [27] during refinement, e.g., for proving infeasibility of abstract

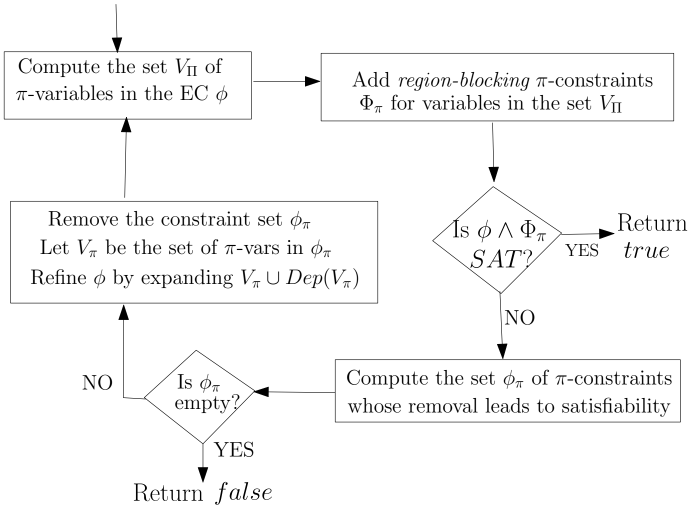

Fig. 2: Flow diagram for checking EC ϕ using *inertial* refinement.

counterexamples with predicate abstraction [18] or procedure abstraction [31]. In contrast to the above approaches which try to prove infeasibility in the concrete model (using cores), we try to obtain constraints (MCS) that allow a witness *to appear* in the abstract model. The notion of MCSs is also related to computing an *interesting* witness to a satisfiable temporal logic formula by detecting vacuous literals [22]. Note that computing MCSs is NP-hard and hence makes `IR` expensive as compared to the light-weight `DCR` method [4] (cf. Sec. II), which only needs a model from the solver. However, we expect that exploring fewer regions in `IR` will compensate for the extra cost.

An MCS for a set of constraints Φ can be computed by obtaining all the proofs of infeasibility (UNSAT cores) of Φ and then computing the minimal *hitting* literal set for this set of UNSAT cores [24]. Many modern constraint solvers, e.g., [15], allow for constraints with weights and solving Max-SAT (MSS) problems natively. Therefore, we can compute MCSs of π-constraints using these solvers by first asserting π-constraints with non-zero weights and then computing the subset of unsatisfied π-constraints in the weighted Max-SAT solution. In our experiments, however, we used the previous method of computing hitting sets: Max-SAT results obtained from [15] were unfortunately erroneous and not usable.

The `IR` procedure can be implemented efficiently using an incremental SMT solver (e.g., [15], [13]). These solvers maintain an internal *context* of constraints to provide incremental checking; constraints can be asserted or retracted iteratively from the context while checking, and the solver is able to reuse the inferred results effectively from the previous checks. Alg. 2 shows the pseudo-code of the inertial refinement algorithm REF-IR using such an SMT solver. REF-IR replaces the naive REF procedure in the overall SAR algorithm (cf. Sec. 1). The description uses the symbol ctx to denote the context of the incremental solver and the methods ASSERT and RETRACT [15] are used for adding and removing constraints to the context incrementally. The procedure starts at the **BEGIN** block by asserting the current EC ϕ in the solver's context. Depending on whether the context is satisfiable or not, the control switches to the locations labeled by **BLOCK** and **EXPAND** respectively (cf. Alg. 2). In the **BLOCK** case, the region-blocking constraints Φ_π are asserted first. If the resultant context is satisfiable, REF-IR returns with SAT result. Otherwise, the control switches to the UNSAT label. Here, a MCS ϕ_π of π-constraints is computed to check if removing any π-constraints may admit a witness to ϕ. If the MCS is empty, no witness is possible and the procedure returns UNSAT. Otherwise, all the π-constraints in the MCS are retracted and the EC ϕ is refined by expanding π-variables V_π in ϕ_π together with their dependent variables $Dep(V_\pi)$.

Theorem 2: The inertial refinement procedure REF-IR returns *SAT* while checking an EC ϕ only if there exists a concrete witness to the error node for ϕ.

Example 2. Recall the summary of the procedure `foo` (Fig. 1) presented in Example 1. The EC ϕ for the assertion at line 12 is $(\psi_{\texttt{foo}} \wedge y_1 < 0)$, where $y_1 = ite(x_1 = c \wedge \pi_n, \lambda_{n,ret}, 0)$ and $x_1 = ite(p \neq 0 \wedge \pi_b, \lambda_{b,ret}, c)$; ϕ contains two π-variables π_b (`bar`) and π_n (`neg`). (**BEGIN**) Initially, ϕ is satisfiable, and REF-IR (Alg. 2) switches to the **BLOCK** label.

(**BLOCK**) The REF-IR procedure first blocks both π_b and π_n (adds π-constraints $\neg\pi_b$, $\neg\pi_n$), and checks for a solution. No solution is found since all feasible paths in `foo` contain a function call. Therefore, the control switches to **EXPAND**.

(**EXPAND**) Here, REF-IR computes an MCS ϕ_π, which is $(\neg\pi_n)$. Since π_n corresponds to function `neg`, `IR` must explore `neg` to find a witness. The procedure then removes $\neg\pi_n$ and refines ϕ by adding summary constraints for π_n and the dependent placeholder $\lambda_{n,ret}$. These constraints ($\pi_n = true$ and $\lambda_{n,ret} = ite(x_1 > 0, -x_1, x_1)$ respectively) are generated by analyzing the `neg` function (cf. Fig. 1).

(**BEGIN**) On checking ϕ again after expansion, the solver finds a witness (lines 2-3-4-6-7-8-9-10-12), with say, $c = 1$, $p = 0$, $x_1 = 1$, $y_1 = -1$. REF-IR now checks if the witness is an actual one (**BLOCK** label) by blocking all π-variables. Note that π_b is the only π-variable remaining in ϕ and the corresponding π-constraint is already asserted. Therefore, REF-IR concludes that the witness is an actual one and terminates. Note how REF-IR avoids the redundant expansion of the complex function `bar`, guided both by the abstract EC ϕ as well as the modular program structure. Also, the efficiency of REF-IR crucially depends on the computed MCSs.

A. Example: IR with Loop-specific Abstraction

Consider the function `loopf` in Fig. 1. The assertion at line 7 checks if on loop exit, the value of j is less than $2 * n$, and is violated only when $n \geq 3$. To see this, consider the data computed at the loop exit (line 7) by a symbolic execution [20] of `loopf` after few initial iterations: (0) $(0 \not< n, j \mapsto 0; i \mapsto 0)$, (1) $(0 < n \wedge 1 \not< n, j \mapsto 0; i \mapsto 1)$ (path condition reduces to $n = 1$), (2) $(n = 2, j \mapsto 2; i \mapsto 2)$, (3) $(n = 3, j \mapsto 6; i \mapsto 3)$, respectively. Note that the value (3) violates the assertion at line 7, while (0), (1) and (2) do not.

In general, a violation like above may require an arbitrary number of iterations of the loop, depending on one or more inputs. Many bug finding methods [9], [4], [32] unroll all program loops to a fixed depth, and may miss bugs like these. The approach in [31] transforms loops to tail-recursive functions; however, conventional static analysis seldom does so. In contrast, we show how inertial refinement can be used to perform a *dynamic* property-driven unrolling of loop regions, with the help of an abstraction *specific* to loop regions. Note that methods based on refining predicate abstractions [5], [18] may detect this violation by refinement; however, constructing

REF-IR(ctx, ϕ)

```
BEGIN:  ASSERT (ctx, φ)
        if ctx is satisfiable then
          goto BLOCK
        else
          goto EXPAND

BLOCK:  V_Π := set of π-variables in φ
        /* Assert π-constraints */
        Φ_π := ∧{(v = false) | v ∈ V_Π}
        ASSERT(ctx, Φ_π)
        if ctx is satisfiable then
          return SAT
        else
          goto EXPAND

EXPAND: φ_π := MCS(ctx)
        if φ_π = false then
          return UNSAT
        /* Witness may exist */
        RETRACT (ctx, φ_π)
        /* Select placeholders to refine */
        V_π := Variables in φ_π
        V'_π := V_π ∪ Dep(V_π)
        foreach λ ∈ V'_π do
          t = GETSUMMARY (λ)
          φ := φ[λ ↦ t]
        goto BEGIN
```

Algorithm 2: The REF-IR procedure for checking an EC ϕ with an incremental SMT solver using inertial refinement. The variable ctx denotes the context of the solver.

and refining predicate abstractions is expensive. In contrast, SAR with cheap abstraction and inertial refinement using loop summaries can detect such violations at a much lower cost.

SAR first computes a local loop body summary, $(i_o < n, [j \mapsto j_o + 2 * i_o;\ i \mapsto i_o + 1])$, where j_o and i_o represent the values of j and i respectively at the beginning of the body. Recall that SAR first checks `loopf` by skipping the loop region with an abstraction of form (ψ, σ); in this case, however, the abstraction is *specific* to the loop region and allows dynamic loop unrolling. More precisely, (1) $\psi = (\pi_0 \vee \pi_{1+})$, where $\pi_0 = (n \leq 0)$ and π_{1+} are path conditions after zero or ≥ 1 loop iterations, respectively, and the map (2) $\sigma = [j \mapsto ite(\pi_0, 0, \lambda_{j,1+}); i \mapsto ite(\pi_0, 0, \lambda_{i,1+})]$, where $\lambda_{j,1+}$ and $\lambda_{i,1+}$, respectively, are the values of j and i obtained after ≥ 1 loop iterations ($i = j = 0$ after zero iterations). Using the above abstraction, the symbolic data obtained at the assertion at line 7 is (ψ, σ) so that the EC ϕ is

$$\phi = ((\pi_0 \vee \pi_{1+}) \wedge n \geq 0 \wedge ite(\pi_0, 0, \lambda_{j,1+}) \geq 2 * n))$$

The procedure REF-IR first checks ϕ by blocking all the loop iterations, i.e., it adds a π-constraint $\neg\pi_{1+}$. The solver checks $(\phi \wedge \neg\pi_{1+})$ and returns UNSAT with the MCS $\neg\pi_{1+}$. As a result, `IR` removes $\neg\pi_{1+}$ and refines ϕ by adding summary constraints for π_{1+}, $\lambda_{j,1+}$ and $\lambda_{i,1+}$, i.e., $\pi_{1+} = (n = 1 \vee \pi_{2+})$, $\lambda_{j,1+} = ite(n = 1, 0, \lambda_{j,2+})$, etc.. `IR` again proceeds iteratively by blocking π_{2+}, π_{3+}, and so on, obtaining MCSs and refining ϕ. A satisfiable solution is obtained in the fourth iteration (with π_{4+} blocked), which corresponds to a true violation witness.

Note that if a witness requires a large number of loop unrollings, refinement using `IR` is inefficient. One solution is to expand multiple loop iterations simultaneously. However, we observed that in many real-life programs having input-dependent loops, few loop unrolls are sufficient for finding bugs; inertial refinement is effective in such cases.

V. EXPERIMENTAL EVALUATION

We implemented the modular analysis SAR (cf. Sec. III) in the F-SOFT [19] framework for verification of C programs. The framework constructs an *eager* memory model for C programs [19] by bounding the heap, flattening aggregate data types into simple types (up to depth 2 for our experiments), and modeling the effect of pointer dereferences by an explicit case analysis over the points-to sets for the pointer variables. Also, F-SOFT instruments the program for properties being checked, e.g., dereference safety (N), array bounds violation (A) and string related checks (S). Therefore, SAR is able to check multiple types of properties in an uniform manner in the F-SOFT framework. The initial model is simplified by the tool with constant folding, program slicing and other light-weight static analysis, and is then provided as an input to the SAR procedure.

We used a wide collection of open-source and proprietary industrial examples for evaluation: L2 is a Linux audio driver (`ymfpci.c`), L9 implements a Linux file-system protocol (`v9fs`), M1, M3 are modules of a network controller software, N1, N2 belong to a network statistics application, F consists of the `ftp-restart` module from the `wu-ftpd` distribution, and Spin corresponds to the SPIN model checker (without the parser front-end). The analyzed benchmarks range from LOC sizes of 1K to 19K. Our analysis focused on discovering known bugs efficiently.

Our implementation of SAR computes summaries and ECs for all program regions locally (cf. Alg. 1), stores them efficiently by representing terms as directed acyclic graphs (DAGs) and manipulates them using memoized traversal algorithms. The local ECs were hoisted up to the entry function and checked using the YICES SMT solver [15] in an incremental manner with refinement (cf. Alg. 1). To precisely model non-linear operators, e.g., modulo, which occur in many of our benchmarks, we encode all variables as bit-vectors.

We evaluated four structural refinement schemes: (i) *Naive*: expand all placeholders in the EC, (ii) `DCR`: use don't-cares for expansion, expand only *one selected* placeholder in each iteration (cf. Sec. II, similar to the state-of-the-art Calysto algorithm [4]), (iii) `DCR`$^+$, same as `DCR` except expand *all selected* placeholders (set V'_π in Alg. 2) in each iteration, and (iv) `IR`, the new inertial refinement scheme. In our experience, expanding all the selected placeholders (*set-expansion*) in each refinement iteration converges much faster than one placeholder at a time (*one-expansion*), and, therefore, is our default mode for *Naive* and `IR` schemes. The experiments were done on a Linux 2.4Ghz Core2Duo machine, with timeout of 1 hr and 8GB memory limit.

Figure 3 shows the experimental comparison between the

Bm	LOC	#EC	Naive		DC-based				IR	
					DCR		DCR$^+$			
			#R	T	#R	T	#R	T	#R	T
F-A	1K	48	162	73	75	282	58	**71**	**51**	78
F-N	1K	18	78	12	63	71	**32**	**11**	51	17
F-S	1.3K	54	100	2044	-	TO	27	**844**	**17**	2359
N1-N	1.2K	77	4	65	2	62	2	62	**0**	**61**
N2-S	1.4K	230	7	9	3	11	3	10	**1**	**9**
L2-A	5.4K	135	550	27	**292**	58	304	29	450	**28**
L9-A	6K	314	978	279	-	TO	549	589	**257**	**162**
L9-N	6K	124	721	22	**114**	139	144	**15**	205	27
M1-A	6K	356	906	**59**	-	TO	527	64	**408**	87
Spin	9K	233	662	2173	-	TO	295	2018	**192**	**1472**
M1-S	12K	196	800	68	338	124	354	62	**283**	**57**
M3-S	19K	419	-	TO	-	TO	253	1599	**221**	**1334**

Fig. 3: Experimental comparison of structural refinement schemes: (i) (Naive) without any selection of placeholders, (ii) DCR [4] (iii) DCR$^+$ with set-expansion and (iv) the new IR scheme. Benchmarks (Bm) are named in "Name-Checker" format, where Checker is either A (array bounds), N (NULL dereference) or S (string checker). LOC shows the lines of code analyzed post-simplification. #EC = the number of ECs checked for the benchmark. #R denotes the number of regions expanded. Time out (TO) of 3600s. Memory limit 8GB. Best figures are in bold.

various structural refinement techniques. The results confirm that structural abstraction methods scale to industrial benchmarks while retaining precision: many of these examples cannot be handled by other techniques, e.g., based on monolithic BMC [9] and predicate abstraction. We report the total time (T) including the summary computation and EC checking times. For each benchmark, we report the total number of regions (#R) expanded during the run. Note that ECs may have either a proof or a witness, and many of them may be checked without any refinement. Also, the set of regions explored (#R) may include the same function under multiple contexts. The results show that DCR$^+$ and IR clearly outperform the naive refinement scheme, which time-outs on the largest example M3-S, implying that selective refinement is essential. However, we observe that DCR time-outs in many cases where even *Naive* with set-expansion finishes. In the following, we compare DCR, DCR$^+$ and IR systematically.

(DCR vs DCR$^+$). Since DCR performs one-expansion, it calls the solver large number of times. As a result, it time-outs on 40% of the benchmarks, while DCR$^+$ finishes in time, showing that DCR$^+$ converges much faster than DCR. However, in most cases where DCR finishes, it expands fewer regions and variables than DCR$^+$, due to one-expansion.

(IR vs DCR). DCR time-outs on many benchmarks, especially the bigger ones, due to one-expansion, whereas IR finishes. The results show that IR outperforms DCR [4] in terms of run-times on all benchmarks. To permit a fair comparison, we augment DCR with set-expansion (DCR$^+$) and compare with IR below. Note, however, that for benchmarks L2-A and L9-N, DCR does expand fewer regions than IR. We discuss this below.

(IR vs DCR$^+$). Both these approaches use set-expansion and finish on all benchmarks. We observe that, in most cases, IR expands fewer regions than DCR$^+$, showing that inertial refinement is indeed useful, and that many properties can be checked while restricting to a smaller region set. For example, benchmarks N1-N and N2-S show the effectiveness of IR: in case of N1-N, DCR$^+$ needs to perform two expansions, while IR doesn't need any expansions. On an average, IR expands about 20% fewer (54% in the best case) regions than DCR$^+$. Moreover, IR outperforms DCR$^+$ in terms of run-times on bigger examples (e.g. Spin, M1-S, M3-S), in spite of being more computationally expensive (requires computing MCSs). Since IR expands fewer regions than DCR$^+$, we believe that the improvement will be more dramatic on larger benchmarks.

On a few examples (F-N, L2-A and L9-N), however, IR expands more regions than DCR$^+$. This is because IR depends crucially on MCSs generated during refinement, which may not be optimal; in these examples, non-optimal MCSs led to exploration of irrelevant program regions. We believe that using more sophisticated MCS computing algorithms [24], [25], based on native MAX-SAT solving inside a constraint solver (as opposed to our method of computing hitting sets of UNSAT cores, cf. Sec. IV) will lead to faster computation of MCSs and hence improve the performance significantly.

We were unable to compare thoroughly with the previous work Calysto [4], [3], since it is not available publicly and the memory models used by F-SOFT and Calysto are different. However, the refinement scheme in Calysto is similar to DCR with one-expansion; in our experience, set-expansion is more powerful since the total number of SMT solver calls are reduced. Refinement based on counterexample-driven analysis of the concrete model [31] as opposed to abstract models is orthogonal to our approach; however, these approaches can also benefit from inertial refinement.

VI. RELATED WORK

Modular methods for sequential programs have been investigated extensively: most techniques perform an over-approximate analysis to obtain proofs of assertion validity via abstract interpretation [11], [12]. In contrast, our focus is on modular bug finding methods, which perform an *under-approximate* program analysis [12]. Taghdiri and Jackson proposed a method based on *procedure abstraction* [31] for detecting bugs in Java programs. To analyze a caller function, the method automatically infers relevant specifications for all the callee functions: it starts from empty specifications, and gradually refines them using proofs derived from analyzing spurious counterexamples in the concrete program model. Babic and Hu introduced the *structural abstraction* methodology in the tool Calysto [4], [3] for analyzing large-scale C programs. Again, the method analyzes the caller by abstracting the callees with summary operators (placeholders). When checking abstract verification conditions (VCs) having these placeholders, structural refinement *expands* the placeholders with the corresponding summaries derived from the callees. In contrast to [31], structural refinement avoids the potentially expensive analysis of the concrete model: placeholders are selected by analyzing the abstract VC using a *don't-care* analysis of the abstract counterexample [4]. Both the above approaches [31], [4] perform refinement based purely on the counterexamples produced by the solver, which is oblivious of the program structure, and hence may explore new program regions even if a witness is realizable in the current regions.

PREfix [6] performs modular bug detection using path-enumeration based symbolic execution [20] to compute bottom-up summaries. These summaries only model partial procedure behaviors and the method may succumb to path explosion. In contrast, we compute precise summaries effectively using a merge-based data flow analysis [21], [3], and employ SAR to explore all program paths relevant to the property in an incremental fashion. The tool Saturn [32] performs bit-precise modular analysis for large C programs; however, the analysis is not path-sensitive inter-procedurally, and leads to infeasible witnesses. Chandra et al. [7] employ property-driven structural refinement to incrementally expand the call graph of Java programs in the presence of polymorphism, to avoid an initial call graph explosion. The ESC/Java tool [16] introduced verification condition (VC) generation based on intra-procedural weakest precondition [14] computation but requires pre/post specifications to reason inter-procedurally. In contrast, inter-procedural VCs are generated automatically in our approach using structural abstraction (cf. Sec. III). Compositional symbolic execution [2] also uses structural abstraction of functions with uninterpreted functions to make coverage-oriented testing more scalable: inertial refinement can also benefit these methods. In context of symbolic trajectory evaluation, Chockler et al. [8] present a method to refine circuit node placeholders using the notion of *responsibility*.

VII. CONCLUSIONS

We presented a modular software bug detection method using structural abstraction/refinement, based on analyzing program *regions* corresponding to modular program constructs. A new inertial refinement procedure `IR` was proposed to address the key problem of structural refinement: `IR` resists the exploration of abstracted program regions by trying to find a witness for an assertion inside the program regions explored previously. The procedure `IR` implemented in the F-SOFT framework scales to large benchmarks and is able to check properties by exploring fewer program regions than the previous don't-care based refinement technique [4]. Future work includes combining `IR` with other schemes, e.g., `DCR`, for more effective placeholder selection. Methods to dynamically expand the heap during analysis will also be investigated. Partitioning a program automatically for efficient SAR is also an interesting open problem. Finally, we plan to perform a detailed usability study of the SAR method for finding bugs in large benchmarks.

Acknowledgements. We would like to thank Domagoj Babic and the members of the Verification group at NEC for several useful discussions. We are also indebted to the anonymous reviewers for their helpful feedback.

REFERENCES

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.
[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, pages 367–381, 2008.
[3] D. Babic. *Exploiting Structure for Scalable Software Verification*. Dissertation, Univ. of British Columbia, Vancouver, Canada, 2008.
[4] Domagoj Babic and Alan J. Hu. Structural abstraction of software verification conditions. In *CAV*, pages 366–378, 2007.
[5] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, volume 36(5), pages 203–213. ACM Press, June 2001.
[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.
[7] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, pages 363–374, 2009.
[8] H. Chockler, O. Grumberg, and A. Yadgar. Efficient automatic ste refinement using responsibility. In *TACAS*, pages 233–248. Springer-Verlag, 2008.
[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5):752–794, Sept. 2003.
[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
[12] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *CC*, pages 159–178. Springer-Verlag, 2002.
[13] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
[14] Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
[15] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.
[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.
[17] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *CAV'97*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.
[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
[19] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *CAV*, pages 301–306, 2005.
[20] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
[21] A. Kölbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *IJPP*, 33(6):645–666, 2005.
[22] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, 2003.
[23] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton U.P., 1994.
[24] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.
[25] Mark H. Liffiton and Karem A. Sakallah. Generalizing core-guided max-sat. In *SAT*, pages 481–494, 2009.
[26] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA'08*, pages 213–224, NY, USA, 2008.
[27] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. pages 530–535. ACM Press, June 2001.
[28] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, NY, USA, 1995. ACM.
[29] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.
[30] M. Sharir and A. Pnueli. Two approaches to interprocedureal data flow analysis. In *Program Flow Analysis: Theory and Applications*, volume 5, pages 189–234. Prentice Hall, 1981.
[31] M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *Autom. Softw. Eng.*, 14(1):87–121, 2007.
[32] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.