DC2: A Framework for Scalable, Scope-Bounded Software Verification

Franjo Ivančić[1], Gogul Balakrishnan[1], Aarti Gupta[1], Sriram Sankaranarayanan[2],
Naoto Maeda[3], Hiroki Tokuoka[3], Takashi Imoto[3], Yoshiaki Miyazaki[3]
[1]NEC Laboratories America, Princeton, USA. [2]University of Colorado, Boulder, USA
[3]NEC Corporation, Kawasaki, Japan
{ivancic,bgogul,agupta}@nec-labs.com, srirams@colorado.edu
{n-maeda@bp,tokuoka@ay,t-imoto@ak,y-miyazaki@bq}.jp.nec.com

***Abstract*—Software model checking and static analysis have matured over the last decade, enabling their use in automated software verification. However, lack of scalability makes these tools hard to apply. Furthermore, approximations in the models of program and environment lead to a profusion of false alarms. This paper proposes DC2, a verification framework using scope-bounding to bridge these gaps. DC2 splits the analysis problem into *manageable parts*, relying on a combination of three automated techniques: (a) techniques to infer useful specifications for functions in the form of pre- and post-conditions; (b) *stub inference* techniques that infer abstractions to replace function calls beyond the verification scope; and (c) automatic refinement of pre- and post-conditions from false alarms identified by a user. DC2 enables iterative reasoning over the calling environment, to help in finding non-trivial bugs and fewer false alarms. We present an experimental evaluation that demonstrates the effectiveness of DC2 on several open-source and industrial software projects.**

I. Introduction

Software Model Checking is a promising technique for finding subtle bugs in software [19]. The primary utility of model checkers lies in their ability to present a *witness* to help explain a bug to a developer. However, the lack of scalability (due to fundamental hardness) and the reporting of false alarms (due to modeling abstractions) form a major hindrance to the adoption of software model checking in industry. A practically viable software model checker must exhibit many key desirable qualities: (1) *Scalability*: handle 1 MLOC and beyond (for C/C++); (2) *Performance*: complete verification within the allotted time; and finally, (3) *Accuracy:* yield accurate bug reports with a low rate of *false alarms*, so that human effort is not wasted in examining them. The capabilities of model checkers have made spectacular advances over the last two decades, especially with the advent of modern SAT/SMT solvers. However, in spite of these advances, when used "*out-of-the-box*" even the most advanced software model checking engine cannot handle large projects that are frequently encountered in industry. In this paper, we present DC2, a verification methodology that enables scalable software model checking.

DC2—which stands for **D**epth-**C**utoff with **D**esign **C**onstraints—uses *scope bounding* along with *automatic specification inference* and *environment refinement* techniques. Scope bounding limits the size of the generated model by excluding functions that are deeply nested in the call graph, thereby enhancing scalability. *Environment constraints* restrict the environment (global variables, unknown calling context, and other external influences) at function interfaces. *Function stubs* capture the effect of calls to library functions, missing source code, or functions deemed outside the scope by DC2. The environment constraints and function stubs for available code are inferred automatically using a light-weight and scalable whole program analysis called SpecTackle.

SpecTackle can infer constraints from pointer/array indirections and user-defined assertions. Furthermore, it propagates (hoists) these constraints across program locations within a function body, and across function calls. As a result, constraints originating from deeply-nested function calls can be automatically *hoisted* many levels higher in the call graph to provide constraints at key interface functions. Scalability is ensured in SpecTackle by focusing on specific syntactic *specification templates* and using a hand-crafted interprocedural static analysis for these templates. Note that our main requirement for SpecTackle is scalability rather than precision. In this respect, it is different from other recent work on automatic generation of preconditions [14, 23], since our inferred preconditions are used only to bootstrap the application of a bit-precise and path-sensitive model checker in the next phase.

The specifications generated by SpecTackle may be incomplete. Furthermore, since aliasing is not treated soundly by SpecTackle, they may also be unsound. Therefore, a model checker (even with a precise program model) can produce false alarms due to lack of a precise environment. To deal with such false alarms, we use an approach we call CEGER (CounterExample-Guided *Environment* Refinement). It is inspired by CEGAR [10], where counterexamples are used to guide refinement of an abstract model. In CEGER, if the user deems a reported witness as a false alarm, the model checker is used to generate a refinement of the environment constraint, computed using a data-sliced weakest precondition backwards over the witness trace. The refined environment constraints are used in subsequent runs of the verification tool, to eliminate the previously found false alarms. The application of CEGER is quite useful in practice since it helps iterative

978-1-4577-1639-3/11/$26.00 © ASE 2011, Lawrence, KS, USA

reasoning over the calling environment and guides the analysis towards non-trivial bugs and fewer false alarms.

We have implemented DC2 as part of VARVEL, a software verification tool developed by NEC. We present an experimental evaluation of DC2 on several open source and industry software projects. Our results show that use of DC2 leads to much fewer failures, enabling the application of software verification techniques on large projects.

Contributions. In this paper, our main contribution is the DC2 verification framework that combines scope bounding with automatic inference and refinement of the environment to enhance applicability of software model checkers on large projects.

- ❒ We present light-weight specification inference techniques to automatically infer environment constraints and function stubs.
- ❒ We present a counterexample-guided environment refinement approach (CEGER) to further refine the environment constraints, triggered by a user classifying a reported warning as a false alarm. The CEGER approach can potentially leverage work done in examining a set of false alarms for eliminating others in future runs.
- ❒ We present an experimental evaluation of DC2 on large open source and industry software, using an implementation in VARVEL, a tool developed by NEC and utilized within a perpetual verification environment.

Overview. The rest of the paper is organized as follows. First, we provide some background on VARVEL and its program modeling in section II. We describe scope bounding in section III and the global analysis for automatic specification inference in section IV. Section V presents our counterexample-guided refinement procedure for environment constraints. In section VI, we report the experimental results on application of DC2. Section VII discusses some related work. Section VIII concludes the paper.

II. BACKGROUND: VARVEL AND MEMORY MODELING

VARVEL is a software verification tool based on an earlier research prototype called F-SOFT [18]. VARVEL uses a combination of abstract interpretation and model checking to find common errors in C/C++ programs including pointer usage errors, buffer overruns, C string errors, API usage violations, and violations of user-defined assertions. At its core, VARVEL utilizes a bit-precise SAT-based bounded model checker (BMC), which operates on a model automatically extracted from a given program. The formulas generated by BMC [8] are solved by a SAT solver to generate witnesses that correspond to property violations. VARVEL has been engineered to handle numerous low-level aspects of C programs including pointer arithmetic, dynamic memory allocation, function pointers, and bitwise operations.

Scalability in VARVEL is enhanced by staging various analyses such that cheaper methods are used first. It uses abstract interpretation [13] on numerical domains of increasing precision (e.g., intervals, octagons [22]) to statically prove properties. Once a property is proved, it is removed from further consideration. In addition, program slicing is used to remove portions of the program that are irrelevant to the unresolved properties. In practice, roughly 60-80% of the automatically instrumented checks in a program are rapidly proved using static analysis, before the bounded model checker is deployed on the rest. Thus, the number of properties to check, as well as the model size is reduced across the different stages.

However, it still remains prohibitively expensive to perform whole program model checking for large programs. Therefore, VARVEL treats every function appearing in a given program as a possible entry function. Although this helps to handle some functions, challenges remain for functions with large call graphs where the extracted models may become too large. Furthermore, when a function other than `main` (or an interface function) is treated as an entry point for verification, the input parameters and global variables are assumed to hold arbitrary values. In reality, however, these parameters are constrained based on values set by its callers. Therefore, the model checker may report a witness, where it assigns a value to an input that cannot be realized in an actual execution, thereby resulting in a false alarm. The aim of DC2 is to enable effective deployment of VARVEL on large systems for finding more bugs and reducing false alarms *with minimal user interaction*.

VARVEL supports different memory models and checkers that use the same underlying analyses. Here, we focus primarily on three of these, namely (a) a fine-grained array-bounds model to check for overflows, (b) a light-weight *validity* model that handles bugs due to `malloc` failures, double-free and free of statically allocated memory, and (c) a light-weight model for detecting memory leaks.

A. Array Bounds Checking

The array-bounds model tracks pointer addresses, allocated bounds for each pointer, and the position of the null-terminator sentinel for strings. This model is along the lines of the CSSV model [15] with some key differences, to reduce model complexity and make it easier to analyze.

The bound $[\mathsf{ptrLo}(p), \mathsf{ptrHi}(p)]$ represents the range of legal values for pointer p, such that p may be dereferenced in our model *without causing an out-of-bounds violation*. If $p \in [\mathsf{ptrLo}(p), \mathsf{ptrHi}(p)]$ then $p[i]$ *underflows* iff $p + i < \mathsf{ptrLo}(p)$. Similarly, $p[i]$ *overflows* iff $p + i > \mathsf{ptrHi}(p)$. By convention, a pointer p is regarded $\mathsf{invalid}$ (uninitialized, null, freed, etc.) whenever $\mathsf{ptrHi}(p) < \mathsf{ptrLo}(p)$. Addresses associated with a pointer *do not* correspond to the physical layout of memory in a program; providing an address to a pointer allows us to model pointer arithmetic and aliasing in an easier fashion. Fig. 1 illustrates the modeling attributes.

For statically allocated arrays, the bounds and addresses are set to fixed values at the start. Dynamically allocated pointers are not provided with a priori fixed addresses. Instead, `malloc` calls are instrumented to assign a new address and bound at every invocation of `malloc`.

Tracking String Lengths. A majority of the buffer overflows in C result from the unsafe use of the standard library

```
char *s=malloc(10);
if (!s) exit(-1);
strcpy(s,"ASE2011");
char *t= s+3;
```

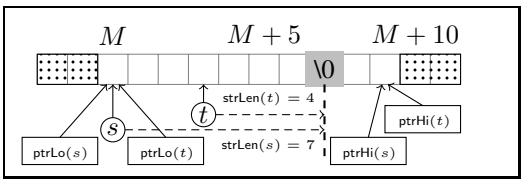

Fig. 1. The memory model for the array bounds model after successfully executing the four statements on the left-hand side: The successful allocation returns a pointer to some new address M, and the lower bound addresses $\mathsf{ptrLo}(s) = \mathsf{ptrLo}(t) = M$. The higher bound addresses are $\mathsf{ptrHi}(s) = \mathsf{ptrHi}(t) = M+9$. Finally, the string lengths are determined using the size abstraction, namely $\mathsf{strLen}(s) = 7$ and $\mathsf{strLen}(t) = 4$.

functions. We extend our memory model to check for such buffer overflows along the lines of CSSV [15].

Corresponding to each character pointer p, we associate a variable $\mathsf{strLen}(p)$ to track the position of the first null-terminator character starting from p. The updates to $\mathsf{strLen}(p)$ are derived along the same lines as those for the pointer bounds with the exception of assignments involving pointer indirections, where updates to a string pointer are propagated to potentially *overlapping* strings as well.

B. Pointer Validity Checking

The pointer-validity checker handles those aspects of buffer overflow checking that do not require tracking of bounds ptrHi and ptrLo for pointers. The validity checker instruments each pointer using a seven-valued monitor $\mathsf{ptrVal}(p)$ to denote the state of a pointer at runtime: (a) null: a NULL pointer; (b) invalid: a non-null invalid pointer, whose dereference may cause a segmentation violation; (c) static: global variables, arrays and static variables; (d) stack: local variables, `alloca` calls, local arrays, formal arguments; (e) heap: dynamically allocated memory on the heap; (f) code: code section, e.g., string constants; and (g) environment: input pointer parameters.

Unlike the overflow checker, the validity checker does not track addresses of pointers and ignores address arithmetic. A pointer expression $p + i$ is *assumed* to have the same validity status as its base pointer p. At a dereference $*p$, our modeling adds an assertion check that is violated if $\mathsf{ptrVal}(p)$ is null or invalid. In case of an assignment to $*p$, we may also report a bug if $\mathsf{ptrVal}(p) = \mathsf{code}$. Calls to `free` set the validity monitor of the argument and its aliases to `invalid`. Finally, leaving a functional scope changes pointers that are set to `stack` to `invalid`.

C. Memory Leak Checking

We also provide a stand-alone model for detecting memory leaks. In comparison to a standard approach of combining memory leak detection with memory safety checkers, our model is very light-weight. Further, we reduce the model size based on the observation that, to find memory leaks, pointer aliasing relationships need to be tracked only between pointers and allocation sites. That is, rather than having to track quadratically many aliasing predicates between all pointers, we simplify the model to track only linearly many aliasing predicates. We omit further details for sake of brevity.

III. Scope Bounding

The basic idea behind scope bounding in DC2 is to *cut off* function calls beyond a desired call depth in the call graph of a given entry function. Here, call depth is defined as the length of the shortest path from the entry function in the call graph. After the cut-off, the function call is replaced by a *stub* that abstracts its effect in terms of preconditions, post-conditions, and modified variables. DC2 is supported by a three-pronged inference methodology:

(a) Infer constraints in the form of preconditions for each function so that the calling environment at the entry function can be captured.
(b) Infer stubs in the form of pre-conditions, post-conditions, and summaries for functions that are cut off.
(c) Support refinement of the inferred pre-conditions and post-conditions upon demand.

In our framework, the inference of pre/post-conditions and stubs is supported by a simple whole program analysis called SpecTackle. The refinement process, called CEGER, employs an analysis around a counterexample generated by a model checker and identified as a false alarm by the user.

The algorithm for function call rewriting in DC2 is shown in Fig. 2. $\phi_{\texttt{entry}}$ and $\phi_{\texttt{g}}$ are the constraints and $stub\_g$ is the stub inferred by SpecTackle. This scope bounding instrumentation is done as a preprocessing step, and is independent of the verification performed later. The depth cutoff scheme is illustrated in Fig. 3. Note that for the entry function of the analysis, the precondition inferred by SpecTackle is assumed, whereas it is asserted for other called functions. The function `h` is deemed outside the scope. Therefore, a stub replacement `h_stub` is automatically created and the inferred preconditions, post-conditions, and stubs are used in place of h. The inference of preconditions, post-conditions, and stub functions to support DC2 are described in the next section, while the refinement process is described in Section V.

IV. Specification Inference

Scope bounding largely alleviates the issue of scalability, enabling verification to cover large parts of the code by using depth cutoff on each entry function. However, cutting off deep function calls or replacing them by over-approximate stubs adversely impacts accuracy, leading to false bugs. Further, we may miss bugs that occur outside the scope. To cope with these issues, DC2 utilizes a light-weight global analysis called SpecTackle. It has two main roles: (1) to generate and hoist preconditions for functions, and (2) to generate stubs

Function **depthCutoff**(`entry`, `depth`)
Map `depthMap`: Func $\mapsto$ Integer.
Insert `assume(`$\phi_{\texttt{entry}}$`)` at the start of `entry`.
for all functions `f` in callgraph **do**
  `depthMap(f)` := Length of the shortest path in call-graph from `entry` to `f`.
for all `f` such that `depthMap(f)` = `depth` **do**
  for all call-sites `c` in function `f` **do**
    let `g` be the function called at `c`
    if depthMap(`g`) > `depth` **then**
      Insert `assert(`$\phi_{\texttt{g}}$`)` before `c`.
      // Cut off function call `c`.
      Replace `c` with call to `stub_g`.

Fig. 2. Function call rewriting in DC2.

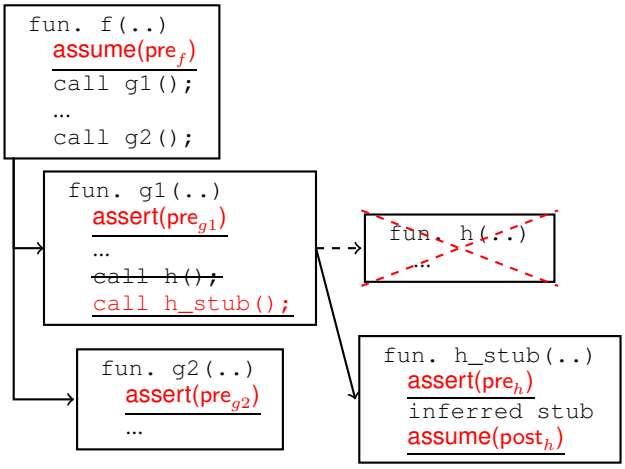

Fig. 3. A schematic illustration of depth cutoff in DC2 on a call graph.

that capture important side-effects of functions. Both these are tailored specifically to the checkers in VARVEL.

A. Preconditions

SPECTACKLE visits functions in a reverse topological order with respect to the call graph of the program. For each function, it hoists error conditions within the body to the start of the function. Hoisting is done by back-propagating the conditions across the statements. In addition, SPECTACKLE also hoists preconditions of the callees to the start of the caller. Hoisted conditions are used for two purposes in the scope bounding algorithm: (1) to constrain the inputs of an internal function `f` when it is used as an entry function for verification, and (2) to assert the correctness of inputs to a callee `g1` of `f`. Note, in particular, that an assertion check `assert(`$\phi_{\texttt{g}}$`)` will trigger a violation during model checking and generate a witness if the hoisted condition for `g` is not correct. In other words, we do not require SPECTACKLE to generate correct preconditions. Instead, we attempt to automatically generate *likely* preconditions. The witnesses generated by VARVEL can be additionally used for manual refinement of the preconditions as described in Sect. V.

In the following, we briefly describe the preconditions that SPECTACKLE generates for a given function.

Pointer validity: For every pointer `p` that is dereferenced, SPECTACKLE *hoists* the condition `(p != NULL)` to the beginning of the function, provided p is not checked for null along all paths that lead to the dereference. If the resulting condition involves a formal parameter or a global variable, it will be retained as a precondition for the function.

Array bound: For every variable `i` that is used as an index expression for accessing a static array of size n, it back-propagates the condition $(0 \leq \texttt{i} < n)$ to the start of the function. If the resulting condition depends only on the inputs to the function, it will be used as a precondition that constrains the range of the respective inputs.

Allocated heap size: If a dereferenced expression is an input variable itself or is aliased with an input, SPECTACKLE tries to capture the constraints on the size of heap area pointed-to by the expression by analyzing pointer arithmetic operations. For example, consider the following function:

```
void f(T* t1,int k)
   {T* t2; t2 = t1 + k; *t2 = 7;}
```

From the expression `*t2` and the pointer arithmetic operation `t2=t1+k`, SPECTACKLE generates the following precondition for `f`: `t1`$+k \in [$ptrLo(`t1`), ptrHi(`t1`)$]$.

Field relation inference: SPECTACKLE captures specific patterns of field usage. For instance, we often see struct definitions that have two fields, one pointing to the heap and the other indicating the length of the heap as below:

```
struct S{char* buf; size_t buflen;};
```

SPECTACKLE automatically extracts such relations and globally constrains the type of inputs.

Assertion hoisting: SPECTACKLE hoists assertions in the program to the beginning of functions. Currently, SPECTACKLE can hoist simple (linear equality or inequality) expressions with some simplification for the hoisted assertions. As a heuristic, we drop non-linear assertions and assertions involving complex conditional expressions.

Other Properties: SPECTACKLE includes support for inferring null-terminator preconditions for strings. It can also be easily extended to infer other properties such as type states.

B. Hoisting of Annotations

SPECTACKLE is designed to be fast and scalable even for large software. Therefore, we avoid computing weakest preconditions since it is typically expensive for the whole program. This section briefly discusses how annotations are hoisted to the beginning of functions and across calls using purely syntactic domain operations.

Let φ_l be a precondition annotation generated at a program point l inside a function f. Our goal is to compute an appropriate precondition ψ corresponding to the entry point of function f. To hoist an annotation φ_l at location l, SPECTACKLE initalizes l with φ_l and every other location with the condition *false*. ψ is computed by means of a backwards data-flow

analysis that captures the effect of various assignments and conditional branches between program point l and the entry point of f.

The backwards transfer function is the (weakest liberal) precondition operator. Preconditions across assignment statements are treated by substitution of the right-hand side in place of the LHS expression. The pre operator is also defined to propagate a constraint φ backwards across a branch condition c. Computing this operation involves the syntactic search for a conjunct in φ that contradicts the branch condition c. If such a conjunct is obtained, the precondition is set to $false$. If, on the other hand, φ contains a conjunct that is identical to c (syntactically), the result of the precondition is given by $true$. Failing this, the precondition is set to φ, instead of the more general constraint $c \Rightarrow \varphi$. This is done in part to keep the syntactic form of the preconditions simple so that the dataflow analysis can be performed efficiently. On the other hand, in some cases, the resulting precondition tends to be stronger, resulting in a false alarm that may need refinement using CEGER. Formally,

$$\mathsf{pre}(\varphi, c) = \begin{cases} false & \text{if } \varphi \text{ "syntactically contradicts" } c \\ true & \text{if } \varphi \text{ "is identical to" } c \\ \varphi & \text{otherwise} \end{cases}$$

A join operator ($\sqcup$) is used to merge two preconditions φ and ψ obtained from the two parts of a branch. The join operator works by matching its operands syntactically. If one of the operand syntactically matches $false$, the result is taken to be the other operand. If ψ can be obtained by conjoining some assertion ψ' to φ then the join chooses the weaker assertion φ. Finally, if the operands do not fall into any of the categories above, the result of the join is the trivial annotation $true$. Formally,

$$\varphi \sqcup \psi = \begin{cases} \varphi & \text{if } \psi \text{ is syntactically } false \\ \varphi & (\psi \equiv \varphi \wedge \psi') \\ \psi & \text{if } \varphi \text{ is syntactically } false \\ \psi & (\varphi \equiv \psi \wedge \varphi') \\ true & \text{otherwise} \end{cases}$$

When the analysis converges, the assertion labeling the entry point of the function denotes the entry precondition ψ. If ψ is an assertion other than $true$ or $false$, it can be propagated to the callers of the function. If $\psi = false$, then a warning is issued to the user.

Example 4.1: Consider function `f1` shown in Fig. 4(a). The pointer dereference $q[4]$ in line 3 of `f1` gives rise to two annotations φ_3 and ψ_3. Consider the assertion φ_3: $q \neq$ `NULL`. Because φ_3 is identical to branch condition c: `q!=NULL` at line 2, the precondition $\mathsf{pre}(\varphi_3, c)$ is $true$. Note that the assertion φ_4 labelling line 4 is initially $false$. Therefore, joining the contribution across the two branches at line 2, yields φ_2 : $true$. As a result, the annotation $q \neq$ `NULL` does not yield a precondition for `f1`.

On the other hand, hoisting the annotation ψ_3: $q + 4 \in [\mathsf{ptrLo}(q), \mathsf{ptrHi}(q)]$ produces the precondition $p + 4 \in [\mathsf{ptrLo}(p), \mathsf{ptrHi}(p)]$. In a case like this, we generate the following precondition: `if`$(p)\,\{p + 4 \in [\mathsf{ptrLo}(p), \mathsf{ptrHi}(p)]\}$.

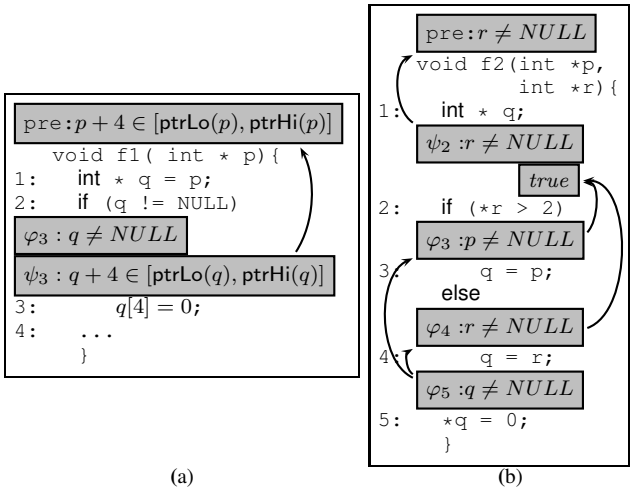

Fig. 4. Hoisting annotations across statements and branches.

Example 4.2: Consider function `f2` shown in Fig. 4(b). The pointer dereference `*q` at line 5 gives rise to the annotation φ_5. Similarly, the pointer dereference `*r` at line 2 gives rise to the annotation ψ_2. Annotation φ_5 can be hoisted across the assignments in lines 3 and 4 yielding φ_3: $p \neq NULL$ and φ_4: $r \neq NULL$, respectively. The join operation at the branch in line 2 yields the assertion $true$. As a result, annotation φ_5 does not contribute to the precondition for `f2`. On the other hand, when the annotation ψ_2 at line 2 is hoisted to the start of `f2`, we generate the preconditon $r \neq NULL$ for `f2`. ■

In practice, a sound and complete precondition is not strictly necessary. If the precondition is overly restrictive, it would lead to a violation at some call site. Similarly, if the precondition is overly relaxed, it would cause false alarms due to an under-specified environment. In practice, our implementation of SPECTACKLE sacrifices soundness in its handling of pointer indirections. Nevertheless, the number of unsound preconditions generated is very few in practice, and such instances are detected through witnesses generated by our model checker.

C. Stub Generation

Recall that scope bounding in DC2 removes functions that are nested deeply in the callgraph. Another practical issue is that verifiers for large projects (especially commercial systems) often need to work with incomplete code bases, because the source code for many third-party libraries or modules is unavailable. For verification purposes, it would be useful to have a summary of the behavior of cut-off functions (for which source code is available) as well as missing functions (for which source code is unavailable). We use SPECTACKLE to automatically generate stubs in both situations.

SPECTACKLE uses the following analyses to generate stubs that model the side-effects of cut-off functions:

Mod-ref analysis: A conservative update to all variables accessible in a cut-off function may generate too many false

alarms. Thus, SPECTACKLE conducts a light-weight mod-ref (modification and reference) analysis to find variables that may be modified. Then, it generates a stub that updates only those variables that are modified within the function and its transitive callees.

Key effects extraction: Library function calls are important for verification. For instance, if a function calls `free` or `exit` internally, this is captured in the stub.

The stubs for missing functions are generated according to default environment assumptions (described in the next section), including suitable preconditions for appropriate initializations and postconditions. In general, the default stubs generated by SPECTACKLE are not sound. They are designed to avoid false alarms that arise frequently in practice, while still exposing problematic situations. In addition, VARVEL provides a mechanism for the user to provide (or refine existing) stubs through use of Stub APIs. This is described in the next section.

Example 4.3: We applied SPECTACKLE on an H.264 video decoder software project comprising about $25k$ LOC. SPECTACKLE ran in a matter of seconds and generated 1473 preconditions for all the functions. Overall, a majority (69%) of the generated preconditions corresponded to direct pointer dereferences of the form $*p$ without any indexing. A significant number (22%) corresponded to preconditions generated due to indexing of statically allocated arrays with known sizes, while the rest corresponded to indexing of pointers of unknown sizes.

Applying VARVEL on the H.264 project without the inferred preconditions yielded 658 witnesses, almost all of them being false alarms. However, the number of witnesses was drastically reduced to 30, when we used VARVEL with the preconditions generated by SPECTACKLE. Of these 30 witnesses, 10 were plausible buffer overflows. ■

V. ENVIRONMENT CONSTRAINTS AND REFINEMENT

The preconditions and stubs described in Sect. IV are collectively referred to as the *environment*. We use an approach we call CEGER (CounterExample-Guided Environment Refinement) for refining the environment. The basic idea behind CEGER is to first apply the model checker using the *default* stubs and preconditions (e.g. those generated by SPECTACKLE). Afterwards, the user examines the witnesses reported and decides which are false alarms. The environment is refined iteratively to avoid the false alarms encountered in the previous iterations. We start by describing the default environment assumptions.

A. Default Environment Assumptions

SPECTACKLE makes some default assumptions about functions when generating the preconditions and function stubs. These assumptions are inspired by common use scenarios. Tab. I shows some of the default assumptions about values of input arguments, bounds of pointers, aliasing, and functions whose code is not included in the analysis.

TABLE I
AUTOMATIC DEFAULT ENVIRONMENT ASSUMPTIONS

Feature	Default assumption
`int`/`char` arg.	Nondet. value
Pointer Aliasing	No aliasing of input argument pointers
Pointer Address	Nondet.: NULL or $p > 0$.
Pointer Bounds	Nondet.: Invalid or $[\mathsf{ptrLo}(p), \mathsf{ptrHi}(p)]$
String Length	Nondet.: Invalid or nullTerm
Func. missing src.	ptr. args are set to nondet., globals are not changed

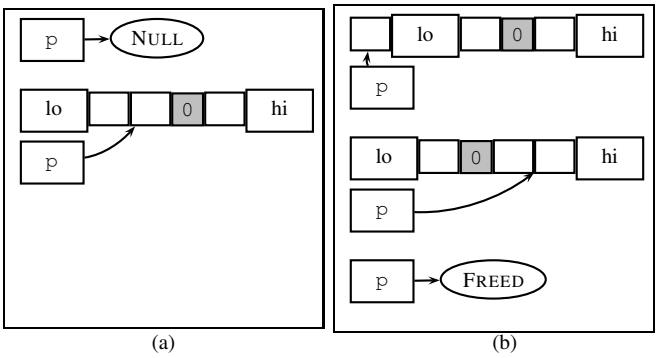

Fig. 5. Possible pointer configurations of an input string p: (a) part of default assumptions, and (b) not permitted by the default assumptions.

Input pointers are nondeterministically set to be NULL, or to a valid pointer address and bounds. If p is chosen to be valid, then p is set to a unique non-NULL address, $[\mathsf{ptrLo}(p), \mathsf{ptrHi}(p)]$ forms a valid range, and $p \in [\mathsf{ptrLo}(p), \mathsf{ptrHi}(p)]$. The allocated length of p is set to be positive but nondeterministic. Finally, strings are assumed to be null-terminated. Note that these assumptions do not lead to a strictly sound treatment of the environment. For example, it is possible to call a function p that has a valid base, but points outside its allocated region at the call site. However, interfaces in C are seldom designed to handle such cases. Fig. 5 highlights some cases allowed by the default assumptions for a string input pointer p, and some other cases that are not allowed.

Stub API. To aid the user in supplying and refining the environment using C/C++, VARVEL supports a special *Stub API* that provides access to auxiliary (instrumented) variables introduced in the verification models. These include variables such as $\mathsf{ptrHi}(p)$, $\mathsf{ptrLo}(p)$, $\mathsf{strLen}(p)$, $\mathsf{ptrVal}(p)$, and so on. The APIs also support assertions, assumptions, preconditions and assignments involving instrumented variables (to set up an initial environment). These allow the user to directly control the model and verification in VARVEL. Indeed, the preconditions and stubs automatically generated by SPECTACKLE utilize the same APIs, providing a common interface for modeling the environment. In addition, VARVEL comes equipped with about 650 pre-defined stubs for standard libraries (e.g. strings), along with embedded assertions for different checkers.

Tab. II shows some of the Stub APIs. The assert function is part of the C library. It is treated as a check. The assume function blocks the execution unless the assumed expression

TABLE II
STUB API TO SPECIFY ENVIRONMENT

preCond(expr)	C expr. expr is a precondition.
assert(expr)	expr is asserted.
assume(expr)	expr is assumed.
NONDET(e_1, e_2)	A nondet. integer in $[e_1, e_2]$.
setStrLen(p, e)	Insert the assignment strLen$(p) := e$.
setValidity(p, e)	Assignment ptrVal$(p) := e$.

is valid. The precondition function preCond has two different meanings depending on where it is encountered. It is used as an assumption when it is encountered at the entry function of the analysis. If encountered in a non-entry function, it is treated as a check. This corresponds to the natural semantics of a precondition in annotation checking.

B. Counterexample-Guided Environment Refinement

The CEGER framework uses a software model checker in two ways:

1) The model checker provides a concrete witness trace showing the control flow and the data values at each program point in the execution leading up to the error. This trace aids the user considerably in determining whether the witness is a false alarm.
2) The model checker suggests an environment constraint to rule out a false alarm, by propagating weakest preconditions backwards along the trace, starting from the violation and ending at the function interface. In our experience, users find it considerably easier to improve upon (e.g. generalize) a suggested constraint rather than derive a suitable constraint from scratch.

We consider refinement of the preconditions for each function. Let φ_f be the existing environment precondition corresponding to a function f in the code. For each witness w, let ψ_w be the weakest precondition computed at the function interface starting from the assertion violation. When a witness w is marked as a false alarm by the user, there are two possible diagnoses:

- In the witness, starting from function f, we observe an assertion violation in the code that is dependent on the input environment. In this case, φ_f is weak. We term such witnesses as *Type 1* witnesses.
- In the witness, starting from another function g, the precondition φ_f (treated as an assertion) is violated when calling f. In this case, either φ_f is strong or φ_g is weak. We term such witnesses as *Type 2* witnesses.

To eliminate a *Type 1* witness w, we refine the precondition φ_f for the entry function f by conjoining it with $\neg\psi_w$: $\varphi'_f = \varphi_f \wedge \neg\psi_w$. This suffices to eliminate w from future iterations of the verifier. In many cases, this also rules out other related witnesses to the same assertion from alternative program paths. However, note that φ'_f is presented to the user as a suggestion. In many cases, the user may be able to rule out additional false witnesses by weakening φ'_f further.

To handle a *Type 2* witness, we re-run the analysis starting from function g and including the code for function f, but excluding the assertion (from the precondition) for f. If the analysis succeeds in producing a violation inside function f that is deemed to be false by the user, then φ_g is weak. Therefore, f's precondition is used to refine φ_g as a *Type-1* witness. If the analysis fails, the user may choose to treat the original witness w as a *Type-1* witness to refine φ_g, or relax the precondition of f such that $[\![\varphi'_f]\!] \supseteq [\![\varphi_f]\!] \cup s$, where s is the concrete state in the witness causing the violation of φ_f. Such a relaxation will also eliminate the witness w from future iterations of the verifier.

Note that since the possibility of a witness being false is arbitrated by the user, mistakes by the user can lead to missed bugs. In particular, the user may introduce a fallacious environment assumption, such as φ_f: $false$, proving all properties trivially. The static analysis can detect such cases and warn the user. Although CEGER is not completely automatic, it is nevertheless quite effective in our experience at enabling users to discover real bugs on large code bases.

TABLE III
DESCRIPTIONS OF BENCHMARKS.

name	version	LOC (analyzed)	#Functions
thttpd	2.25b	6.7K	145
GenericNQS	3.50.10-pre1	15.1K	253
libupnp	1.6.6	17.9K	363
Product P1	—	54.5K	408
Product P2	—	143.6K	727

TABLE IV
RESULTS OF DC2 EXPERIMENTS. Success ratio reports the percentage of functions successfully verified within the given timeouts. Likely Bugs reports the number of bugs that were communicated to developers with numbers in parenthesis representing the number of interprocedural bugs.

	Success Ratio		#Likely Bugs	
	w/o DC2	w/ DC2	w/o DC2	w/ DC2
thttpd	68%	96%	0 (0)	5 (3)
GenericNQS	82%	94%	6 (2)	6 (2)
libupnp	81%	98%	8 (0)	19 (8)
Product P1	89%	96%	8 (1)	11 (6)
Product P2	88%	91%	14 (3)	22 (9)

VI. EXPERIMENTS

This section describes experiments conducted on open source and proprietary software projects. First, we highlight the usage of DC2 on several benchmarks and show the effectiveness of scope-bounded model checking to find interesting interprocedural bugs. Then, we present some experiments on counterexample-guided environment refinement.

A. DC2 Experiments

We applied VARVEL to five benchmarks, including two industry programs from NEC, with and without DC2. The description of the benchmarks is shown in Tab.III. Due to time limitations, especially for investigating results, we decided to employ only a part of each benchmark for the experiments. The column LOC shows the sizes of modules analyzed by VARVEL. Product **P1** is a developer tool whose original LOC is $100k$ and product **P2** is a business application software whose original LOC is $1400k$.

For the experiments, the DC2 depth was set to 1, i.e., each scope consisted of two levels of function calls in the

urlconfig.c:390-396

```
L1   err_code = config_description_doc( doc, ipaddr_port, &root_path );
L2   if( err_code != UPNP_E_SUCCESS ) { goto error_handler;}
L3   err_code = calc_alias( alias, root_path, &new_alias );
```

urlconfig.c:203-340

```
  int config_description_doc(IXML_Document *doc, const char *ip_str, char **root_path_str){
      ...
L4   err_code = ixmlNode_appendChild( rootNode, ( IXML_Node * ) element );
L5   if( err_code != IXML_SUCCESS ) { goto error_handler;}
L6   textNode = ixmlDocument_createTextNode( doc, ( char * )url_str.buf );
L7   if( textNode == NULL ) { goto error_handler;}
     ... // *root_path_str is updated here.
     error_handler:
L8   if( err_code != UPNP_E_SUCCESS ) { ixmlElement_free( newElement );}
     ...
L9   return err_code;}
```

Fig. 6. A subtle inter-procedural bug in libupnp-1.6.6.

call graph, and we did not change any preconditions or stubs generated automatically by SPECTACKLE. The analysis was performed with a time bound of $800s$, and the results are shown in Tab. IV. As expected, we observed that the success ratio (percentage of functions successfully verified within an allotted time) improved with DC2 for all benchmarks. Clearly, scope bounding enabled application of VARVEL on even those functions with large call graphs. Further, the number of detected bugs also increased with DC2.

Interestingly, many of the bugs found without DC2 were intraprocedural. On the other hand, VARVEL was able to find deep interprocedural bugs with DC2. One such (previously unknown) bug in libupnp is shown in Fig. 6. It was only found when we utilized DC2. The variable `root_path` is supposed to be initialized by `config_description_doc` at `L1`. If the initialization fails, it is designed to jump to an error handler at `L2`. However, VARVEL found a case where the initialization fails and the returned error code is `UPNP_E_SUCCESS`. The witness generated by VARVEL indicates that if `IXML_SUCCESS` is assigned to `err_code` at `L4` and `ixmlDocument_createTextNode` returns `NULL`, then the execution will proceed to the error handler without changing `err_code`. `IXML_SUCCESS` and `UPNP_E_SUCCESS` have the same value of 0. Thus, uninitialized `root_path` will be passed as a parameter to the `calc_alias` function (`L3`) which checks that the corresponding parameter is not `NULL`. Note this bug is produced even without analyzing `ixmlNode_appendChild` and `ixmlDocument_createTextNode`, which are out of scope of the verification run.

B. CEGER Experiments

In this section, we only present VARVEL experiments that do not utilize SPECTACKLE. This allows us to highlight the advantage of CEGER independently of DC2. However, this also means that these experiments do not take full advantage of DC2 and thus result in an increased rate of false alarms.

TABLE V
DATA FROM INITIAL RUN OF THE ZITSER ET AL. BENCHMARKS. **LEGEND:** AI: ABSTRACT INTERPRETATION; MC: MODEL CHECKING, PRF.: PROOFS, WIT.: WITNESSES, T/O: TIME OUT.

Code size (kLOC)			Analysis Result (# Properties)				
Total	Avg	Max	Total	AI-Prf.	MC-Prf.	Wit.	T/O
46	1.9	4.5	3281	2155	4	90	500

TABLE VI
ANALYSIS OF WITNESSES FROM ZITSER ET AL. BENCHMARKS

Status	#Wit(1)	#Wit(2)
Plausible Bugs	7	10
Missing function getopt, optarg, ...	25	0
Missing function dn_skipname	12	0
Missing functions setpwent, getpwent	43	0
Modeling limitation: Array/string elements	3	3

Zitser Benchmark Suite. We ran VARVEL on some public benchmarks put forth by Zitser et al. to evaluate the performance of academic and commercial static analysis tools [28]. The benchmark suite consists of 25 programs, which form a part of a larger open source application with known overflow bugs involving arrays and strings. We allowed each instance to run for $1800s$. The programs themselves range in size from $1 - 5kLOC$ (after preprocessing, forward slicing and instrumentation). Table V shows the results of running the analysis on these examples. Note that we are able to prove a majority ($\sim 65\%$) of the properties statically. The model checker produces 90 concrete violations. Each violation was manually examined by one of the authors. Of the 90 witnesses found, 7 were found to be real violations based on the witness. The remaining 83 were classified into 4 categories. Table VI shows an analysis of these violations (`#Wit(1)`). Note that missing preconditions are not a problem for these benchmarks, since they contain drivers that initialize the environment. Indeed, a large number of false positives arose due to missing functions that did not have any stubs. Overall, about one man hour was spent in analyzing all the witnesses in the first run.

TABLE VII
RESULT DATA FOR TWO INDUSTRIAL CASE STUDIES USING *CEGER*

	kLOC	#Functions	Witnesses		#Preconds	#Stub	Bugs	
			R1	**R2**			PV	AB
P3	24	188	211	19	17	6	5	2
P4	70	654	951	181	7	3	6	3*

Our subsequent iteration included stubs for the missing functions. The stubs were created by consulting the description of function behavior specified by the manuals. Table VI shows the results of the second iteration (`#Wit(2)`). This iteration results in a total of 13 witnesses, of which 10 were found to be plausible.

Industrial Benchmarks. Next, we applied VARVEL on a small embedded software application (in industrial use) with $6k$ LOC. The application lacks a single "`main`" function. Therefore, we ran the analysis on each entry function in the source. No partitioning was required for this application. We first used the pointer validity checker. The check ran in 60 minutes and produced 15 witnesses in all. One of the authors analyzed all 15 witnesses in 30 minutes. It yielded one plausible bug, where the result of a `malloc` was dereferenced without a check. We formulated preconditions based on the other witnesses. These preconditions covered both pointer and array overflow checks. Overall, the entire analysis involved 3 rounds of the validity checker and 4 rounds of the array overflow checker, requiring 200 minutes to run. The resulting witnesses revealed 7 bugs overall.

We helped in conducting case-studies on two larger embedded software projects **P3** and **P4**. The study involved the use of VARVEL by two engineers who received tool training and documentation, but were not verification experts. Each study was time limited: 6 person days for **P3** and 7 person days for **P4**. Table VII summarizes the results. Each analysis carried out two iterations with the pointer validity checker followed by two iterations using the more complex array bounds checker. The first iteration used the default environment assumptions. Subsequently, the witnesses were all examined, and preconditions and stubs were written for the next iteration. Table VII shows the number of witnesses in the first round using default environment assumptions (**R1**), and the number of witnesses after CEGER (**R2**). It also classifies the environment assumptions into preconditions and missing stubs written by the engineers. These preconditions and stubs are shared by multiple entry functions. Finally, we report on the number of real bugs for pointer validity (`PV`) and array overflows (`AB`) (only about a quarter of AB witnesses were examined by the engineers).

VII. RELATED WORK

VARVEL uses abstract interpretation and bit-precise bounded model checking for analysis. Approaches based on abstract interpretation [13] have been used in tools such as *PolySpace* [2], *Astrée* [9], *C Global Surveyor* [27]. These tools focus on checking embedded applications with special features such as simple aliasing, no dynamic allocation, simple control flow and no recursion. However, our approach is designed to be more general purpose. The CBMC tool due to Clarke et al. [11] also uses bit-precise bounded model checking, but does not use abstract interpretation and has limited scalability. The CoVerity verifier [1] has been successfully applied to large industrial as well as open source projects, but an experimental comparison is outside the scope of this paper.

Abstraction Refinement. CEGAR (CounterExample Guided Abstraction Refinement) was proposed and successfully used in many efforts [6, 10, 20]. In particular, it has been used with predicate abstraction and refinement in the SLAM project, with continuing improvements in industry applications [7]. Our *CEGER* approach is very much inspired by CEGAR. Whereas other CEGAR efforts use spurious counterexamples to refine the abstract model of the program, we use them only to refine the calling environment of the function being verified or to model missing functions. CEGER is not completely automated, and relies upon the user to determine whether the reported witness is a false alarm.

Scope Bounding. Several approaches utilize scope bounding to achieve better scalability in verification [5, 21, 24, 25]. Taghdiri and Jackson proposed a CEGAR-like refinement-based method to find bugs in Java programs [25]. It iteratively expands the scope of called functions by utilizing information from counterexamples, and continues until it finds a proof or a witness that does not rely on any unconstrained value. Babić and Hu proposed *structural abstraction* [5] that gradually relaxes the boundary of verification by inserting function summaries on demand for a given property. In contrast, DC2 works for all properties within a statically-determined scope. Instead of expanding the scope iteratively, it refines the preconditions and stubs for cut-off functions, based on witness traces.

Automatic Environment Generation. Recently, numerous formalisms have focused on automatically generating models of the environment, based on automatic sound abstraction [26], specification mining [4], automata learning [3], and so on. Our environment generation is based on a light-weight whole program analysis, where we are occasionally unsound. The main motivation is to suppress false alarms and to capture conditions spanning deeply-nested function calls to find interprocedural bugs.

Automatic Contract Inference. Techniques for synthesizing contracts automatically from pointer indirections and assertions in the code have also been proposed by Moy and Marche [23] and by Cousot et al. [14]. Our approach for precondition synthesis is similar in spirit, but differs radically on the choice of abstract domains: (a) our approach is mostly syntactic in nature, based on matching expressions in the code, and (b) our handling of pointer indirection is simple, albeit unsound in theory. However, the preconditions inferred seem to be accurate in practice. Furthermore, a wrong precondition can be detected by VARVEL as a type-1 or a type-2 witness. Our simpler approach can infer preconditions for large projects, with $500k$LOC and beyond, in a matter of minutes. This design choice works well in combination with precise modeling in our model checker.

Annotation Checking. Our work also relates to annotation checking tools that utilize function interface specifications, e.g. ESC/Java [16] and more recently VCC [12]. Although these efforts can handle complex specifications, they require significant manual effort to write them, and support for automatic annotation inference or refinement is largely absent. Tools such as SALInfer [17] provide some automated annotation inference for runtime errors. In contrast to standard annotation checking, note that DC2 handles multiple levels of function calls in the analysis, thereby utilizing more precise interprocedural contexts for finding bugs.

VIII. Conclusions

We presented the DC2 framework for program analysis supported by automated specification inference. Our experimental results for DC2 in VARVEL support our experience that a software model checker can accommodate the requirements from industry by carefully designing and engineering its application. We are investigating other directions to improve DC2, such as adaptively tuning scope-bounding based on program metrics and prior verification runs.

IX. Acknowledgments

We would like to acknowledge the assistance and support of Shinichi Iwasaki-san and Fusako Mitsuhashi-san from NEC Corporation, and Mustafa Hussain and Naveen Sharma from NEC HCL Systems Technologies during the development of VARVEL.

References

[1] CoVerity Inc. program verifier. www.coverity.com.
[2] PolySpace program analysis tool. www.polyspace.com.
[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proc. POPL*, pages 98–109. ACM Press, 2005.
[4] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
[5] D. Babić and A. J. Hu. Structural abstraction of software verification conditions. In *CAV*, 2007.
[6] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*, pages 203–213. ACM Press, 2001.
[7] T. Ball, E. Bounimova, R. Kumar, and V. Levin. Slam2: Static driver verification with under 4% false alarms. In *FMCAD*, pages 35–42, 2010.
[8] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207, 1999.
[9] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, volume 548030, pages 196–207. ACM, 2003.
[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
[11] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.
[12] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*. Springer, 2009.
[13] 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.
[14] P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *VMCAI*. Springer, 2011.
[15] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. PLDI*. ACM Press, 2003.
[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] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.
[18] F. Ivančić, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-Soft. In *ICCD*. IEEE, 2005.
[19] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.
[20] R. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.
[21] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA*, 2008.
[22] A. Miné. The octagon abstract domain. In *WCRE*, 2001.
[23] Y. Moy and C. Marché. Modular inference of subprogram contracts for safety checking. *Symbolic Computation*, 45, 2010.
[24] D. Shao, S. Khurshid, and D. E. Perry. An incremental approach to scope-bounded checking using a lightweight formal method. In *FM*, 2009.
[25] M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *ASE*, 14(1):87–121, 2007.
[26] O. Tkachuk, M. B. Dwyer, and C. Pasareanu. Automated environment generation for software model checking, 2003.
[27] A. Venet and G. P. Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI*, pages 231–242. ACM Press, 2004.
[28] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSoft/FSE*. ACM, 2004.