Augmenting Counterexample-Guided Abstraction Refinement with Proof Templates

Thomas E. Hart*, Kelvin Ku*, Arie Gurfinkel†, Marsha Chechik*, David Lie‡
*Department of Computer Science, University of Toronto, {tomhart,kelvin,chechik}@cs.utoronto.ca
†Software Engineering Institute, Carnegie Mellon University, arie@sei.cmu.edu
‡Department of Electrical and Computer Engineering, University of Toronto, lie@eecg.utoronto.ca

***Abstract*—Existing software model checkers based on predicate abstraction and refinement typically perform poorly at verifying the absence of buffer overflows, with analyses depending on the sizes of the arrays checked. We observe that many of these analyses can be made efficient by providing *proof templates* for common array traversal idioms idioms, which guide the model checker towards proofs that are independent of array size.**

We have integrated this technique into our software model checker, PtYasm, and have evaluated our approach on a set of testcases derived from the Verisec suite, demonstrating that our technique enables verification of the safety of array accesses independently of array size.

I. Introduction

Software model checking based on predicate abstraction and counterexample-guided abstraction refinement (CEGAR) has been shown to be effective for checking correctness of highly non-trivial programs and is now part of commercial tools such as SDV [1]. The abstraction, a set of predicates, is improved dynamically, based on identification of infeasible counterexamples, until it is sufficiently precise to prove the property of interest. In practice, the power of CEGAR software model checkers (henceforth referred to as SMCs) is limited by their ability to choose predicates well. Perfectly selecting predicates is impossible due to the undecidability of software verification, so this process always relies on heuristics.

We want to use SMCs to verify the absence of buffer overflows, which are a major threat to the security of C programs. Current SMCs typically perform poorly at this task due to *loop unrolling* [2]. Our goal is to improve this common-case performance, so that analyses are independent of the sizes of the arrays being checked. Our solution is to define *proof templates* designed to work when a program uses common idioms to traverse an array, and to attempt to guide an SMC towards these proofs automatically. The templates are modular, separating the analysis of the loop's body from the analysis' preconditions on loop entry. We have implemented an algorithm to heuristically map ⟨*loop, variable*⟩ pairs in a program to proof templates. When the algorithm detects that a template may apply, it guides our SMC towards the template proof by supplying it with a set of predicates and assumptions. If the SMC is able to prove the original property using these predicates and assumptions, it then proceeds to discharge the assumptions (prove they hold). If any stage of the analysis fails, the SMC backtracks to an earlier stage, making the overall process *sound*, despite the unsoundness of our algorithm for suggesting proof templates.

```
    void example () {
1:    int i=0, sz=1024;
      int M = sz-1;


2:    while (i != M) {
3:      if (! (i<1024))
4:        ERROR:;
5:      i++; } }
```
(a)

```
    void example () {
1:    int i=0, sz=1024;
      int M = sz-1;
      assume(i <= M);
      assume(M <= 1024);
2:    while (i != M) {
3:      if (! (i<1024))
4:        ERROR:;
5:      i++; } }
```
(b)

Fig. 1. (a) Example program, (b) Application of proof template.

In the remainder of this paper, we give an algorithm for using proof templates within an SMC (§ II-B), and describe proof templates corresponding to common array traversal idioms and a method for heuristically identifying when they may apply (§ II-C). We compare our implementation of this framework, called PtYasm, with other state-of-the-art SMCs, using testcases derived from the Verisec suite [2] (§ III).

II. Technique

We present our technique for using proof templates in SMCs; a technical report [3] gives a more thorough treatment.

A. CEGAR Software Model Checking

SMCs check a property ψ on a program P, where ψ is the reachability of a given line of P, labelled "ERROR". SMCs can be used to check assertions by transforming a statement **assert**(p) into a conditional, such as the one on lines 3–4 of Fig. 1(a), and checking whether the assertion's failure branch is unreachable, in which case the assertion is considered to be *safe*. We assume that the program is annotated with bounds checking assertions on each array access. See [1], [2], [4] for more details on CEGAR.

SMCs often perform poorly when verifying array bounds checks in loops, such as the one in Fig. 1(a) (for succinctness, the array is not shown). A typical SMC first finds the spurious error trace $\tau_1 = \langle 1, 2, 3, 4 \rangle$, which it can eliminate by adding the predicates $i < 1024$ and $i = M$ to its model of the program. In the new model, the values of these predicates will be unknown after line 5; hence, the SMC finds the path $\tau_2 = \langle 1, 2, 3, 5, 2, 3, 4 \rangle$, in which ERROR is reachable in *two* iterations of the loop on lines 2–5. This new trace can be eliminated by adding predicates $i + 1 < 1024$ and $i + 1 = M$, resulting in a model in which ERROR is reachable in *three* iterations. The SMC continues eliminating paths containing increasing numbers of loop iterations, finally proving ERROR unreachable after 1024 iterations of the abstract-check-refine loop. Such *loop unrolling* makes the analysis impractical.

Algorithm 1 CEGAR+PT — CEGAR with proof templates.

```
1:  procedure CEGAR+PT($P, \psi$)                                  ▷ Program, Property
2:    $DB \leftarrow$ BUILDDB ($P$)                                   ▷ Template occurrences
3:    $(E, \psi_0, S) \leftarrow (\emptyset, \psi, \text{empty stack})$ ▷ (Preds, Init. prop., Backtracking stack)
4:    loop
5:      $\mathcal{M} \leftarrow$ ABSTRACT ($P$, $\psi$, $E$)                    ▷ $\mathcal{M}$ = model
6:      $\tau \leftarrow$ MODELCHECK ($\mathcal{M}, \psi$)
7:      if $\tau = \epsilon$ then                                   ▷ No path to ERROR
8:        if NOASSUMPTIONSLEFT($S$) then return SAFE
9:        else $(P, E, \psi) \leftarrow$ DISCHARGENEXT($S$)         ▷ Discharge assumptions
10:     else
11:       if SPURIOUS($\tau, P$) then                              ▷ Refine abstraction
12:         if TIMEOUT ($S$) then $(P, E, \psi) \leftarrow$BACKTRACK ($S$)
13:         else
14:           if $\exists \ell \cdot$ UNROLLING($\ell$) $\wedge$ HAVETEMPLATE($\ell, DB, \psi, S$) then
15:             $(P, E, S) \leftarrow$USETEMPLATE($\ell$,$DB$,$\psi$,$S$)
16:           else $E \leftarrow E \cup$ REFINE ($\tau$)
17:       else
18:         if $\psi = \psi_0$ then return UNSAFE
19:         else $(P, E, \psi) \leftarrow$BACKTRACK ($S$)          ▷ Assumption did not hold
```

B. CEGAR with Proof Templates

Iteratively removing paths of increasing length from loops like the one in Fig. 1(a) is an inefficient and unnatural way to prove safety. A more natural proof shows inductively that $i \leq M \leq 1024$ is an invariant at line 2:

1) Initially, $i \leq M \leq 1024$ holds trivially at line 2.
2) If $i \leq M \leq 1024$ at line 2 and the loop is entered, then $i \leq M \leq 1024 \wedge i \neq M$, so $i < M \leq 1024$ at line 3, and $i \leq M \leq 1024$ after the `i++` on line 5.

Since $i \leq M \leq 1024$ is an invariant at line 2, if the loop is entered, $i < M \leq 1024$ at line 3, so the assertion is safe.

We present CEGAR+PT (Alg. 1), a variation on the classical CEGAR algorithm used by SMCs that attempts to guide an SMC towards proofs like the one above by introducing *proof templates*, which provide outlines of correctness proofs. The call to BUILDDB on line 2 builds a database which maps $\langle$*loop, variable*$\rangle$ pairs in the program to proof templates, by examining the structure of the loop. When a loop is being unrolled, USETEMPLATE (line 15) queries this database to see if a template may be useful. BUILDDB may suggest templates which do not help, so the calls to BACKTRACK on lines 12 and 19 ensure that unhelpful templates are eventually abandoned.

We illustrate CEGAR+PT on the example program in Fig. 1(a), but stress that it works on more complex programs, such as those shown in [5]. BUILDDB records that i appears to be bounded by M in the loop on lines 2–5 of the program in Fig. 1(a), and guesses that this bound can be used to prove the safety of the assertion. When CEGAR+PT realizes that the loop on lines 2–5 is being unrolled, it invokes USETEMPLATE, which applies a corresponding proof template. USETEMPLATE adds predicates to E based on the structure of the loop. This alone is insufficient, as some of these predicates must be true before the loop is entered — for example, in Fig. 1(a), the loop invariant $i \leq M \leq 1024$ only leads to a proof of correctness if it holds initially. Often proving that these predicates are in fact true on loop entry requires the discovery of additional "support" predicates. To ensure that these support predicates are discovered, USETEMPLATE also adds explicit assumptions to P (see Fig. 1(b)).

Since BUILDDB may suggest proof templates whose assumptions do not hold, CEGAR+PT must discharge all assumptions used (line 9), and backtrack if any do not hold (line 19). This is facilitated by a backtracking stack S. Whenever USETEMPLATE supplies a template on line 15, it adds a stack frame to S, containing (a) the current iteration of the loop beginning on line 6 of CEGAR+PT (to enable the TIMEOUT check on line 12), (b) the current values of P, E, and ψ (so that the calls to BACKTRACK on lines 12 and 19 can restore the state before the template was applied), and (c) the assumptions associated with the template (to enable the call to DISCHARGENEXT on line 9). S also keeps track of the number of times a template has been applied to $\langle \ell, i \rangle$, to ensure that each candidate template can be applied in turn.

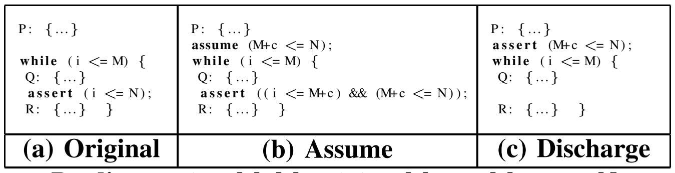
In the example program in Fig. 1(a), USETEMPLATE adds to E the predicates $i \leq M$, $M \leq i$, and $M \leq 1024$ and the assumptions shown in Fig. 1(b). These are part of the definition of the template, and are instantiated using the parameters i, M, and 1024, which come from the program. With these, the SMC can prove ERROR unreachable using the inductive argument described at the beginning of this section.

HAVETEMPLATE checks whether a template can be used for the pair $\langle \ell, i \rangle$; our implementation checks that (a) there is a proof template for $\langle \ell, i \rangle$, (b) no template for $\langle \ell, i \rangle$ is already in use, and (c) $\psi = \psi_0$, the last being because we do not yet support the use of proof templates for multiple loops.

C. Proof Templates for Array Traversals

We have defined four parameterized proof templates corresponding to common array traversals in which array indices are kept in-bounds via (a) explicit numerical comparisons, (b) sentinel null characters (as in string traversals), or (c) updates correlated with a second variable kept in bounds by one of the above two methods. We choose among these templates based on combinations of two conditions: whether the iterator (defined below) in the loop condition is the same as the iterator in the assertion being checked, and whether the loop condition is an arithmetic comparison on an iterator or a test on an array cell. We have found that this information is often sufficient to choose the correct template. Our template descriptions assume structured loops with *loop conditions* at their heads; however, our templates work equally well for common less structured loops with *exit branches* at their heads *or* within their bodies. We outline PTYASM's handling of such loops, and how we derive template parameters for them, in [5].

Preliminaries. We use standard compiler concepts [6] to describe our proof templates. A statement s is a *definition* of a variable v (or *s defines v*) if s contains an assignment to v. If s reads the value of v, we say that *s uses v*. For any loop ℓ, constants and variables which are used but not defined in ℓ are called *loop constants*. If s_1 and s_2 are statements, we say that s_1 is *dependent* on s_2, written $s_1\ \delta\ s_2$, if there exists a variable v such that s_1 uses v, s_2 defines it, and the definition reaches s_1. We write $s_1\ \delta^\star\ s_2$ if there exists a set of statements $s'_1, \ldots s'_n$, such that $s_1\ \delta\ s'_1\ \delta\ \cdots\ \delta s'_n\ \delta\ s_2$.



```
P:  {...}

while (i <= M) {
 Q:  {...}
 assert (i <= N);
 R:  {...}   }
```

(a) Original

```
P:  {...}
assume (M+c <= N);
while (i <= M) {
 Q:  {...}
 assert ((i <= M+c) && (M+c <= N));
 R:  {...}   }
```

(b) Assume

```
P:  {...}
assert (M+c <= N);
while (i <= M) {
 Q:  {...}

 R:  {...}   }
```

(c) Discharge

Predicates: $i \leq M, M \leq i, i \leq M + c, M + c \leq N.$

Fig. 2. Structure of single-variable explicit template.

We introduce the concept of loop iterators to define our templates. Given a loop ℓ, a variable i is an *iterator of* ℓ iff there exists a statement s such that (1) s is a definition of i within ℓ; (2) $s\ \delta^\star\ s$; and (3) s is not dominated by any other definition s' in ℓ such that s' assigns a loop constant to i. Unlike induction variables [6], iterators need not change by a fixed amount on every loop iteration.

Single-Variable Explicit Template. We use the single-variable explicit template when a loop iterator i appears in a bounds check within a loop, and the loop condition is a comparison between i and some loop constant M, e.g., as in Fig. 1(a). Fig. 2 shows how the template is communicated to an SMC. Our description assumes that the loop condition is $i \leq M$, but the details are similar if the loop condition is $i < M$ or $i \neq M$. The symbols P, Q, and R in Fig. 2 denote regions of the program. The template's parameters are i, M, N, and c; roughly, c represents (a guess at) the maximum amount by which i can be increased in Q. The template breaks the proof of safety down as follows:

1) $\{true\}\ P\ \{M + c \leq N\}$,
2) $\{M + c \leq N\}\ Q; R\ \{M + c \leq N\}$, and
3) $\{i \leq M \wedge M + c \leq N\}\ Q\ \{i \leq M + c \wedge M + c \leq N\}$.

Fig. 2(b) shows the effect of the changes that USETEMPLATE makes to the program's internal representation in order to guide the SMC towards this proof. The template supplies the predicates $i \leq M$, $M + c \leq N$, and $i \leq M + c$, since they appear in the proof outline, and the predicate $M \leq i$, which we have often found to be useful: together with $i \leq M$, it allows us to describe any comparison $(<, \neq, \ldots)$ between i and M. In cases where the loop condition is $i \neq M$, such as in our example in Fig. 1(a), USETEMPLATE also inserts an **assume**$(i \leq M)$ statement before the loop, and the SMC must additionally prove that $\{i < M\}\ Q; R\ \{i \leq M\}$ holds, so that i is bounded on each iteration. As shown in Fig. 2(c), the SMC must discharge all assumptions used.

Extending to Strings and Two-Variable Traversals. Loops over arrays often involve two iterators — for example, when copying data using pointers. We use the two-variable explicit template when the loop condition is a test on one iterator (the *leader*), but the bounds check is on another. The main idea is to try to prove that the change in the second iterator on any iteration of the loop is bounded by the change in the leader. The parameters are i, j, M, N, and c. Assume that the loop condition is $i \leq M$, and that the bounds check is **assert**$(j \leq N)$; the details are similar if the loop condition is $i \neq M$ or $i < M$. We introduce the variables i_s and j_s to denote the values of i and j before the loop is entered; these variables allow us to represent the notion of change in i and j.

	All Testcases (59)	String (14)	Two-variable (10)
PTYASM	49	11	7
YASM	17	0	0
BLAST	19	0	0
SATABS	22	0	0

TABLE I
NUMBER OF TESTCASES SOLVED AT BUFFER SIZE 1024, 600S TIMEOUT.

Let P' be the composition of statements $(P; i_s = i; j_s = j)$, and let $\Phi = (M + c - i_s \leq N - j_s)$ and $\Psi = (j - j_s \leq i - i_s)$. Then, the template breaks the proof of safety down as follows:

1) $\{true\}\ P'\ \{\Phi\}$,
2) $\{\Phi \wedge \Psi\}\ Q; R\ \{\Phi \wedge \Psi\}$, and
3) $\{i \leq M \wedge \Phi \wedge \Psi\}\ Q\ \{i \leq M + c \wedge \Phi \wedge \Psi\}$.

The template thus supplies the SMC with the predicates $i \leq M$, $M \leq i$, $i \leq M+c$, $j - j_s \leq i - i_s$, and $M + c - i_s + j_s \leq N$, and with an **assume**$(M + c - i_s + j_s <= N)$ before the loop. The template works similarly if the leader decreases on each loop iteration; for example, if the leader is a count of the amount of space remaining in a buffer.

For loops over strings, we use a *length abstraction* [7], implemented using program instrumentation, to conservatively approximate the semantics of *strlen*; for details, see [5]. String templates are similar to explicit templates; for example, the single-variable string template uses the predicates $i \leq$ *strlen*(A), *strlen*$(A) \leq i$, *strlen*$(A) \leq N$, and $A[i] =$ '\0', and adds **assume** $(i <= strlen(A))$ and **assume**$(i <= strlen(A))$ statements before the loop. The two-variable explicit template generalizes to string traversals similarly.

III. EVALUATION

We have compared our SMC augmented with proof templates, PTYASM, against three other SMCs: YASM [8] (without proof templates), BLAST [4], [9], and SATABS [10]. We derived 59 testcases from the Verisec suite [2], selecting 26 patched testcases from 18 suite entries, excluding entries which either did not contain buffer-dependent loops or which contained loop structures already represented in the set. Since our methodology is currently limited to analyzing loops in isolation, we constructed, by hand, single-loop testcases isolating each bounds checking assertion in each selected program. We limited our evaluation to safe testcases since proof templates are not designed to aid falsification. To pass a testcase, a tool had to verify the assertion within a 600s timeout period; crashes, timeouts, and incorrect results were counted as failures. We set the buffer sizes to 1024 so that the tools could not feasibly solve testcases by loop unrolling. All tests were run on a quad-core Intel 2.66GHz machine with 16GB of RAM.

Table I summarizes our results. Overall, PTYASM is able to verify 49 of our 59 testcases — more than twice as many as the next best tool, SATABS. Of our 59 testcases, 14 involved traversing strings. Of these 14 testcases, PTYASM was able to verify 11, whereas the other tools were able to identify none. Out of the ten of our 59 testcases which involved two-variable traversals, PTYASM was able to verify 7, whereas the other tools were able to verify none. More details are available in our technical report [3], and the complete experimental data and test materials are available online at `http://www.cs.toronto.edu/~kelvin/ase08`.

```
1   while (A [i] != '\0') {
2     if (! (j >= M)) {
3       assert (j < N);
4       j++;  }
5     i++; }
```
(a)

```
1   while (A[i] != '\0') {
2     if (A[i] == '?') {
3       assert (i < N);
4       A[i] = '\0';  }
5     i++; }
```
(b)

Fig. 3. Programs where (a) exit branches do not yield the correct template and (b) the current template is insufficiently general.

PTYASM failed to check ten testcases within the timeout period. In one of these testcases, the correct template was supplied to PTYASM, but it was unable to converge within the timeout period. The remaining nine failures fall into two groups: (1) four testcases in which BUILDDB did not suggest the correct template, either because no template exists, or because the correct template cannot be inferred from loop exit branches, and (2) five testcases in which our current templates are insufficiently general. In both groups, PTYASM backtracked to the standard refinement strategy (loop unrolling) and eventually timed out. Fig. 3(a) and (b) show simplified representatives of these two groups. In Fig. 3(a), the single-variable explicit template can be used to prove safety; however, since the bound on the iterator j is established on line 2, which is not a loop exit branch, BUILDDB does not suggest this template. We plan to enhance BUILDDB to suggest templates based on checks which dominate assertions. Fig. 3(b) requires a generalization of our string template, since $strlen(A)$ can change within the loop body. We plan to extend our system to keep track of the value of $strlen(A)$ on loop entry in such cases; doing so would enable PTYASM to verify this testcase.

IV. RELATED WORK

Proof templates enable efficient modular safety proofs for array traversals in the context of an SMC. Other projects have made various tradeoffs to analyze loops, with array bounds checking being a common motivation.

Using stubs for string functions makes verification efficient when programs only traverse arrays using these functions, but does not address custom array traversals [11]. Augmenting SMCs with looping counterexamples [12], [13], or concrete execution [14] enables efficient *falsification* rather than verification. Replacing boolean programs with linear programs enables some problems involving array traversal to be solved efficiently, but sacrifices completeness of the SMC's model-checking phase [15]. Using a split prover [16] for refinement ensures eventual convergence if a proof of safety exists within difference logic, but does not guarantee that the proof is efficient, and can drastically increase the number of predicates used if a predicate with a large difference bound is needed. ACSAR [17] uses *transfinite refinement* to abstract the effects of loops, and has been used to verify small loops.

Beyer et al. use *invariant templates* within an SMC to avoid loop unrolling [18], but do not address strings or modularize proofs. The user must specify the invariant template, which requires intuition about the structure of the loop and the property being checked. It may be possible to combine our approach with theirs, by using our algorithm to suggest proof templates, casting them as invariant templates, and using their invariant synthesis to customize them.

Outside the domain of SMCs, several tools check the safety of array accesses using abstract interpretation or Hoare-style deductive verification. Bounds-checking tools based on abstract interpretation [7], [19] rely on fixed abstractions which cannot be refined at analysis time, thus leading to false alarms. Tools based on deductive verification typically require an inference procedure to provide loop invariants [20]. Denney and Fischer [21] have applied a pattern-based approach which is similar in spirit to ours to deductive verification, but not for the problem of array bounds checking, and their scope is restricted to automatically-generated programs.

V. CONCLUSIONS AND FUTURE WORK

Proof templates supply SMCs with the predicates and assumptions needed to prove the safety of array bounds checks in common loops. We plan to extend our framework to handle multiples loops, and to explore proof templates for other problem domains.

ACKNOWLEDGMENT

Thomas E. Hart and Kelvin Ku were funded by MITACS; Thomas E. Hart was also funded by an NSERC CGS.

REFERENCES

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough Static Analysis of Device Drivers," in *Proc. EuroSys'06*.

[2] K. Ku, T. E. Hart, M. Chechik, and D. Lie, "A Buffer Overflow Benchmark for Software Model Checkers," in *Proc. ASE'07*.

[3] T. E. Hart, K. Ku, M. Chechik, A. Gurfinkel, and D. Lie, "Augmenting Counterexample-Guided Abstraction Refinement with Proof Templates," Dept. Computer Science, Univ. Toronto, Tech. Rep. CSRG-581, 2008.

[4] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," in *Proc. POPL'02*.

[5] T. E. Hart, K. Ku, M. Chechik, A. Gurfinkel, and D. Lie, "PTYASM: Software Model Checking with Proof Templates," in *Proc. ASE'08 — Tool Demonstrations Track*.

[6] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[7] N. Dor, M. Rodeh, and S. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," in *Proc. PLDI '03*.

[8] A. Gurfinkel, O. Wei, and M. Chechik, "YASM: A Software Model-Checker for Verification and Refutation," in *Proc. CAV'06*.

[9] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from Proofs," in *Proc. POPL'04*.

[10] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based Predicate Abstraction for ANSI-C," in *Proc. TACAS'05*.

[11] S. Chaki and S. Hissam, "Certifying the Absence of Buffer Overflows," SEI, CMU, Tech. Rep. CMU/SEI-2006-TN-030, 2006.

[12] D. Kroening and G. Weissenbacher, "Counterexamples with Loops for Predicate Abstraction," in *Proc. CAV'06*.

[13] C. Wang, A. Gupta, and F. Ivancic, "Induction in CEGAR for Detecting Counterexamples," *Proc. FMCAD'07*.

[14] D. Kroening, A. Groce, and E. Clarke, "Counterexample Guided Abstraction Refinement via Program Execution," in *Proc. ICFEM'04*.

[15] A. Armando, M. Benerecetti, and J. Mantovani, "Abstraction Refinement of Linear Programs with Arrays," in *Proc. TACAS'07*.

[16] R. Jhala and K. L. McMillan, "A Practical and Complete Approach to Predicate Refinement," in *Proc. TACAS'06*.

[17] M. N. Seghir and A. Podelski, "ACSAR: Software Model Checking with Transfinite Refinement," in *Proc. SPIN'07*.

[18] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, "Path Invariants," in *Proc. PLDI'07*.

[19] A. Venet and G. Brat, "Precise and Efficient Static Array Bound Checking for Large Embedded C Programs," in *Proc. PLDI'04*.

[20] Y. Moy, "Sufficient Preconditions for Modular Assertion Checking," in *Proc. VMCAI'08*.

[21] E. Denney and B. Fischer, "Annotation Inference for Safety Certification of Automatically Generated Code (Extended Abstract)," in *Proc.ASE'06*.