PBE-Based Selective Abstraction and Refinement for Efficient Property Falsification of Embedded Software

YOEL KIM, Kyungpook National University, South Korea
YUNJA CHOI*, Kyungpook National University, South Korea

Comprehensive verification/falsification of embedded software is challenging and often impossible mainly due to the typical characteristics of embedded software, such as the use of global variables, reactive behaviors, and its (soft or hard) real-time requirements, to name but a few. Abstraction is one of the major solutions to this problem, but existing proven abstraction techniques are not effective in this domain as they are uniformly applied to the entire program and often require a large number of refinements to find true alarms. This work proposes a domain-specific solution for efficient property falsification based on the observation that embedded software typically consists of a number of user-defined auxiliary functions, many of which may be loosely coupled with the main control logic. Our approach selectively abstracts auxiliary functions using function summaries synthesized by Programming-By-Example (PBE), which reduces falsification complexity as well as the number of refinements. The drawbacks of using PBE-based function summaries, which are neither sound nor complete, for abstraction are counteracted by symbolic alarm filtering and novel PBE-based refinements for function summaries. We demonstrate that the proposed approach has comparable performance to the state-of-the-art model checkers on SV-COMP benchmark programs and outperforms them on a set of typical embedded software in terms of both falsification efficiency and scalability.

CCS Concepts: • **Software and its engineering → Model checking**; *Embedded software*; *Programming by example*.

Additional Key Words and Phrases: selective abstraction, function summary refinement, property falsification

ACM Reference Format:
Yoel Kim and Yunja Choi. 2024. PBE-Based Selective Abstraction and Refinement for Efficient Property Falsification of Embedded Software. *Proc. ACM Softw. Eng.* 1, FSE, Article 14 (July 2024), 23 pages. https://doi.org/10.1145/3643740

1 INTRODUCTION

Embedded software is prevalent in our daily lives, controlling almost all electrical systems around us, from small-scale devices such as smartwatches or pacemakers to large-scale devices such as medical devices, automobiles, rail systems, and avionics. They are often safety-critical, and their failure can be catastrophic, as we have witnessed more often than desired. So, naturally, rigorous and comprehensive verification of embedded software is extremely important.

Decades of effort have been spent on comprehensive validation and verification of embedded software using formal methods, dominantly using model-based development [10, 23, 37]. These approaches formally model functional/non-functional requirements of embedded software and validate the model by formally checking essential properties. The validated model can be used

*Corresponding author.

Authors' addresses: Yoel Kim, Kyungpook National University, Daegu, South Korea, kimyoel2305@gmail.com; Yunja Choi, Kyungpook National University, Daegu, South Korea, yuchoi76@knu.ac.kr.

ACM 2994-970X/2024/7-ART14
https://doi.org/10.1145/3643740

for code generation, test generation, or for both. The model-based approach has seen substantial success in practice, especially in the automotive and avionics domains, but unfortunately, most other domains are still unfamiliar with the technique and largely rely on code-based verification [38, 42, 50, 55].

Comprehensive verification/falsification of embedded software programs is challenging and often impossible mainly due to the typical characteristics of embedded software: It (1) typically uses (a great number of) global variables, which increases the coupling level and the verification complexity of the program, (2) is reactive in terms of responding to external entities (users, hardware devices, etc.), and (3) includes (soft or hard) real-time behavior, which often requires temporal properties with an explicit timing concept. The complexity caused by these characteristics is beyond the limitations of state-of-the-art verification capability.

Abstraction is perhaps the only solution to this problem, but existing proven abstraction techniques [8, 14, 17, 24, 45, 52] are not effective in this domain as they are uniformly applied to the entire program without considering those intrinsic characteristics, which often means that a large number of refinements are necessary to find true alarms. For example, when verifying programs with complex control structures, predicate abstraction [45] undergoes numerous iterations of refinements if it starts with abstract models that are too coarse (i.e., minimal and imprecise). This is not desirable in embedded software, where most behaviors are executed within an infinite loop, requiring the exploration of a potentially infinite number of execution paths when checking a property that can be verified or violated only after numerous iterations. In fact, the well-known model checker CPAchecker [17] configured to use predicate abstraction with interpolants failed to find a true alarm after performing 426 refinements over three days when applied to an elevator controller program (see Section 5).

Based on these observations, we derived the following hypothesis: If, instead of abstracting the entire program, only specific parts (e.g., functions) of the target program can be *selectively abstracted and refined*, then the level of abstraction may be sufficient for verification/falsification efficiency while the concrete parts can contribute to maintaining a high level of verification/falsification precision. To validate our hypothesis, we propose *Selective Abstraction and Refinement using Function Summary* (FS), an approximation of a function's behavior, by leveraging *Programming-By-Example* (PBE) [35]. With PBE, we synthesize FSs that are simple, accurate, and generalized from input-output (I/O) examples to abstract *auxiliary functions* (detailed in Section 3.3) in the program. The drawbacks of using PBE-based FSs, which are neither sound nor complete, for abstraction are counteracted by symbolic alarm filtering and novel PBE-based refinements for FSs.

Fig. 1 shows the overall process of our approach: Given a target program and I/O examples of functions defined in the program, we first identify *auxiliary functions* that are decoupled from global variables as the first selection. These auxiliary functions are synthesized iteratively to construct FSs aimed at low *complexity* and reasonable *similarity* (Definition 1 and 2 in Section 3.2) compared to the original functions. The constructed FSs go through the second selection where they are compared with each other in terms of similarity and complexity. Falsification for a property is performed by bounded model checking (BMC), after replacing selected auxiliary functions with FSs. If a counterexample that leads to a violation of the property is generated, then *symbolic alarm filtering* (detailed in Section 4.1) checks whether a symbolic program path constructed from the counterexample is feasible. If it is infeasible, we can be certain that this is due to the use of FSs as they are the only abstracted parts, and thus we only need to refine the FSs from the infeasible program path and invalid I/O pairs, which can guide the model checker to explore another program path. This falsification process iterates using the refined FSs until a true alarm is identified.

We implemented the proposed approach named PBEAR (**PBE**-based Selective **A**bstraction and **R**efinement) in a toolset for a proof of the concept, which uses DUET [44] for synthesizing FSs

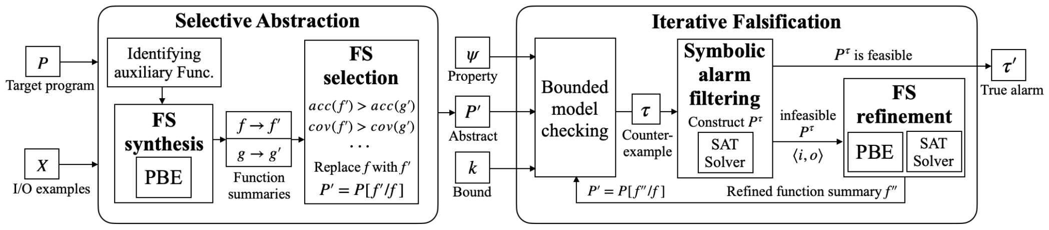


Fig. 1. Overall process of our approach.

and the CBMC [25] for falsification and SAT solving. The falsification ability of PBEAR was evaluated in comparison with that of CBMC [25], CBMC-refine [20], and CPAchecker [17] over 16 SV-COMP benchmark programs [11] and three embedded software programs with 15 safety properties. PBEAR showed competitive performance on the SV-COMP benchmark programs and outperformed the others on three embedded software examples, reducing falsification time and succeeding in falsifying 3 to 12 more properties than the other tools.

The major contributions of this paper are as follows:

- PBEAR is the first technique that efficiently falsifies properties of embedded software by using PBE-based function summaries and selective abstraction. It provides criteria for selection of function summaries.
- PBEAR provides a novel PBE-based refinement approach for function summaries using hints from symbolic alarm filtering to ensure that the refined function summaries do not revisit infeasible symbolic program paths already identified.
- An experimental evaluation on three embedded software programs with 15 safety properties showed that PBEAR improves falsification efficiency and scalability.

The remainder of this paper is organized as follows: Section 2 shows a motivating example; Section 3 describes details of PBE-based selective abstraction; Section 4 describes details of symbolic alarm filtering and PBE-based FS refinement; Section 5 reports the experiment results; Section 6 discusses related work; and Section 7 concludes the paper with future work.

2 MOTIVATION

We illustrate our motivation with Fig. 2, an example of simplified code exhibiting major characteristics of embedded software: (1) It operates as a reactive system with an infinite loop (lines 2–10); (2) it uses the global variable speed; and (3) it has a safety property that involves multiple time steps (lines 7–10). The main control logic invokes a few functions: `read_sensor`, which reads input from an external sensor (line 3); `process`, which processes the input (lines 11–15); `check`, which verifies whether speed and `data` are valid (lines 16–24); and `update`, which updates the value of speed (lines 25–29). The property on line 10 specifies that the system shall reduce speed to below 50 within 10 time steps.

Verifying or falsifying this type of software and properties is challenging. The presence of infinite loops with external interactions may indefinitely increase the search depth, the use of global variables complicates this exploration as the temporal changes of their values must be traced to identify true alarms, and properties involving time steps may need at least a minimum number of loop iterations to explore in order to identify reasonable property violations.

This difficulty is not even made easier through *lazy abstraction with interpolants* [45], which is one of the most common approaches in predicate abstraction, as 13 of the 15 predicate abstraction tools in SV-COMP [11] are based on lazy abstraction with interpolation. It initially replaces each

```
 int speed = 0;
 void main() {
    int in, data, timer;
1: timer = check(0, 0);
2: while (true) {
3:   in = read_sensor();
4:   data = process(in);
5:   if(check(speed, data))
6:     update(data);
7:   if(speed>=50)
8:     timer = timer + 1;
9:   else timer = 0;
10:   assert(timer < 10);}}
```

```
 int process(int x) {
11: if (x < 30)
12:   return 0;
13: else if (x < 50)
14:   return 1;
15: else return 2;}
```

```
 int check(int a, int b){
16: if (a < 0)
17:   a = -a;
18: if (b < 0)
19:   b = -b;
20: while (b != 0) {
21:   int t = b;
22:   b = a % b;
23:   a = t;}
24: return a;}
```

```
 void update(int data) {
25: if (data == 0)
26:   speed = speed - 30;
27: else if (data == 1)
28:   speed = speed - 20;
29: else speed = speed + 50;}
```

Fig. 2. Example of embedded control code.

predicate, a Boolean expression at a program location, with `true`, implying that any statement in the control flow is executable regardless of guarding conditions. For example, in Fig. 3 (a), regardless of the actual result of `[speed>=50]` at node 7, either node 8 or node 9 is reachable in the initial search. Thus, in Fig. 3 (b), an infeasible error path where node 9 is executed is explored by a model checker, producing a false alarm. To avoid exploring such infeasible paths, refinement is performed by calculating new predicates using Craig interpolation [28]; e.g., `[timer==0]` from nodes 9 to 10, and the model checker searches for a new error path. However, as there still remain several guard-free transitions with abstraction, we may witness a large number of counterexample-guided abstraction refinement (CEGAR) [24] cycles. In fact, the example in Fig. 3 (a) costs 33 refinements to find a true alarm using predicate abstraction.

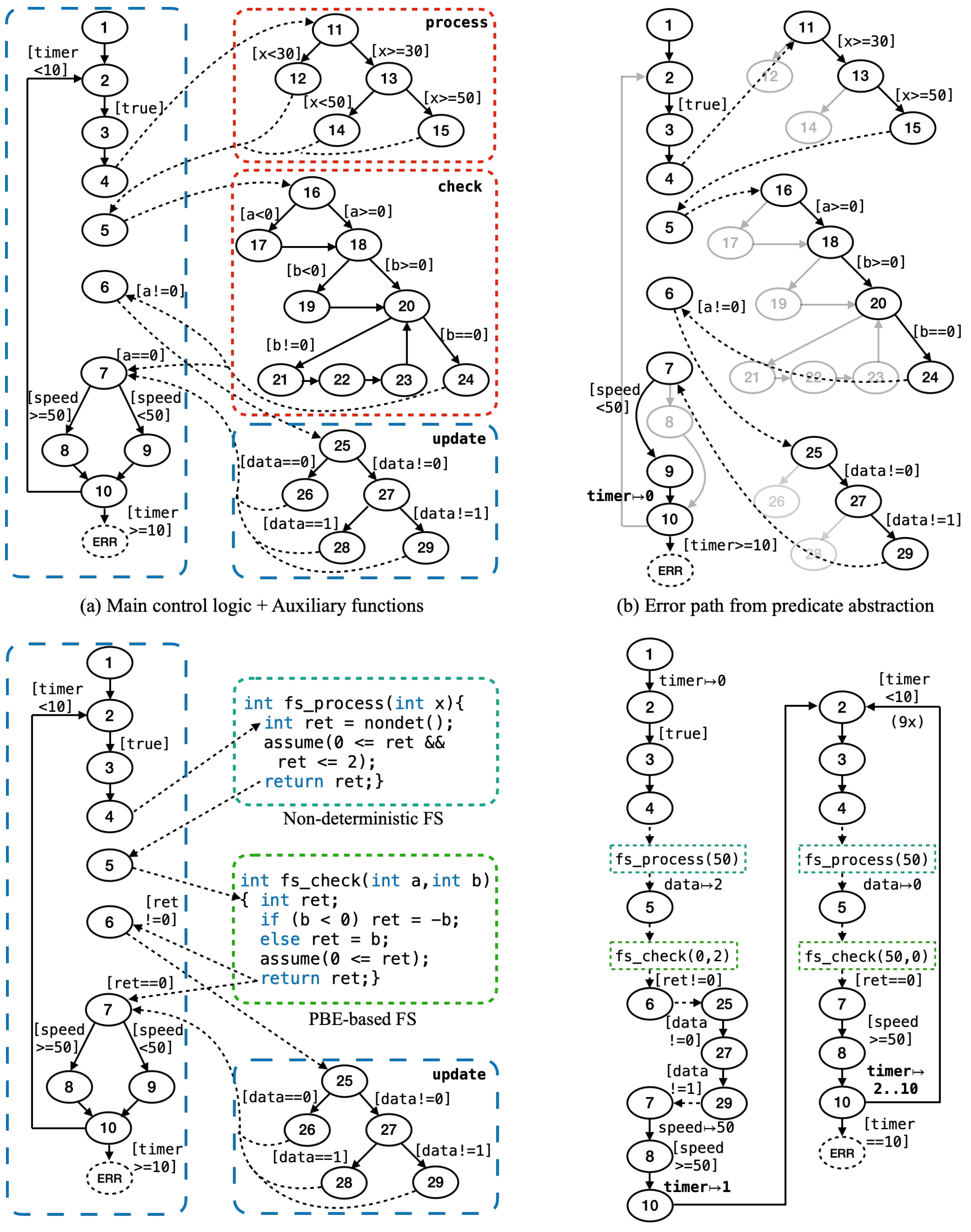
On the other hand, Fig. 3 (c) illustrates our selective abstraction, which replaces only auxiliary functions with FSs, leaving the main control logic as it is (these terminologies will be detailed in Section 3.3). In the figure, `process` is replaced with a non-deterministic FS `fs_process`, which returns a non-deterministic value within its output range, and `check` is replaced with an FS `fs_check` synthesized by PBE, while `main` and `update` remain the same without applying abstractions. A falsification attempt using model checking for the property at node 10 may generate a counterexample trace as illustrated in Fig. 3 (d). The trace consists of two parts. Firstly, one iteration is executed from the left side to increase `speed` to 50 and `timer` to 1. Secondly, 9 iterations are executed from the right side to increase `timer` until `[timer==10]` is true while keeping `[speed==50]` true. As we did not apply abstractions to the main control logic (`main` and `update`), transitions in the counterexample trace are concrete and valid except for those influenced by FSs `fs_process(x)` and `fs_check(a,b)`. Therefore, we only need to refine the FSs, not the whole program, thereby reducing the cost for refinements. Even if Fig. 3 (d) is a false alarm because `fs_check` returns an incorrect output value (`check(50,0)` should be 50, but it returned 0), our selective abstraction shown in Fig.3 (c) results in successful falsification within one refinement of the FSs.

3 PBE-BASED SELECTIVE ABSTRACTION

3.1 Programming-By-Example (PBE)

PBE [5, 34, 35, 44] solvers take I/O examples as *constraints* to find a program that meets the given examples by searching through an infinite set of candidate programs. PBE has been more widely applied in various domains than other program synthesis techniques, such as those based on logical specification [36] and program sketching [53], due to its ease of use. Representative application examples include auto-completion of spreadsheets [48], automated mock object generation [29], and automated program repair [43], to name but a few.

Syntax-Guided Synthesis (SyGuS) [3] aims to satisfy not only the given I/O examples (as semantic constraints), but also *syntactic* constraints that restrict the search space of candidate programs



```
int fs_process(int x){
  int ret = nondet();
  assume(0 <= ret &&
    ret <= 2);
  return ret;}
```

Non-deterministic FS

```
int fs_check(int a,int b)
{ int ret;
  if (b < 0) ret = -b;
  else ret = b;
  assume(0 <= ret);
  return ret;}
```

PBE-based FS

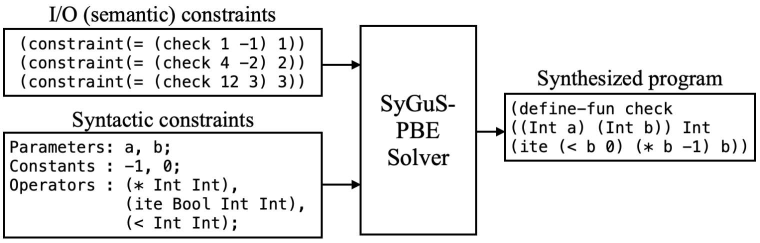
(a) Main control logic + Auxiliary functions

(b) Error path from predicate abstraction

(c) Main control logic + Function summaries

(d) Error path from selective abstraction

Fig. 3. Difference between predicate abstraction and selective abstraction using FSs. (a) is the CFG of Fig. 2 divided into main control logic (dashed boxes) and auxiliary functions (dotted boxes). Solid and dotted edges represent intra- and inter-procedural control flow edges, respectively. [expr] is a guarding condition to reach the next node.



```
(constraint(= (check 1 -1) 1))
(constraint(= (check 4 -2) 2))
(constraint(= (check 12 3) 3))
```

```
Parameters: a, b;
Constants : -1, 0;
Operators : (* Int Int),
            (ite Bool Int Int),
            (< Int Int);
```

```
(define-fun check
((Int a) (Int b)) Int
(ite (< b 0) (* b -1) b))
```

Fig. 4. Overview of SyGuS-PBE.

by limiting available parameters, constants, and operators, to achieve a more efficient PBE. As shown in Fig. 4, it takes a set of I/O constraints and syntactic constraints as input, and produces a SyGuS-formatted program that meets the given I/O constraints using only the given syntactic constraints. This program can be converted into a C program (e.g., `ite` is converted into `if-else`).

However, there are several limitations. Firstly, overfitting is an intrinsic issue, as it aims at finding a program satisfying only given I/O constraints. We do not know whether the synthesized program would produce a correct output for an input outside the I/O constraints. Secondly, the choice of I/O constraints can greatly affect its performance. Our preliminary observations showed that if we imposed too many arbitrary constraints, the complexity (code size) became too large or might not be synthesized within a given time budget. Thirdly, SyGuS-PBE tools support only limited data types, e.g., bit vector, integer, or string. Floats and dynamic objects are not generally supported [4].

3.2 Iterative Function Summary Synthesis

Due to its high accessibility, construction of FS using PBE seems quite attractive if we can mitigate limitations of PBE for the falsification of embedded software. Our approach tries to mitigate the overfitting problem and the complexity issue of PBE through iterative synthesis using *similarity* and *complexity* measures. Our goal is to construct FSs with reasonable *similarity* and lower *complexity* compared to the original functions. Our iterative process is shown in Fig. 5 and comprises four steps: (1) *I/O Constraint Selection* selects an I/O pair $\langle i, o \rangle$ from a set of I/O examples X and adds it to a set of I/O constraints C; (2) *SyGuS-PBE* generates a SyGuS-PBE specification by combining semantic constraints C and syntactic constraints extracted from a target function f and synthesizes a FS f' for f; (3) *FS Evaluation* evaluates f' in terms of its complexity and similarity, and provides feedback for the selection of new I/O constraints; (4) *Output Range Analysis* statically analyzes f to identify its output range and annotates the final f' with an `assume` statement that guards the valid output range of f'. The process iterates over I/O Constraint Selection, SyGuS-PBE, and FS Evaluation, until f' achieves 100% accuracy (is a part of *similarity*) or timeout[1].

3.2.1 The Iterative Process. The set of I/O examples X[2] is the collection of executions from the target function f, which is the source of inputs for the PBE solver as well as the oracle for the evaluation of the generated FSs. In the beginning, a small subset of X is randomly selected and is added to the set of I/O constraints C as semantic constraints. A SyGuS-PBE solver generates a FS f' for f using C and syntactic constraints from f whose quality is evaluated by *FS Evaluation* w.r.t. its *complexity* and *similarity.*

Definition 1. The *complexity* of a FS f', $cmplx(f')$ is the number of branches used within f'.

[1]By setting a high accuracy goal, we can aim for the highest possible accuracy within a time budget (1 hour), even though achieving 100% accuracy is rare, and timeouts are commonly encountered.
[2]We collect X through random and (mainly) concolic execution of f to cover various behaviors.

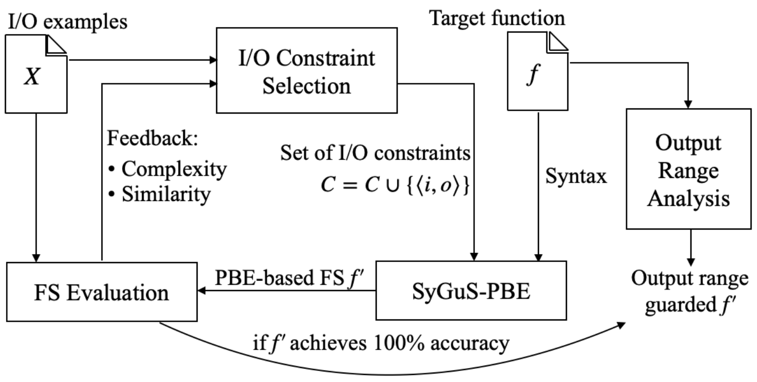


Fig. 5. Process of iterative function summary synthesis.

Definition 2. The *similarity* of a FS f' is measured by X w.r.t. accuracy and output coverage:

- *Accuracy* ($acc(f')$): The ratio of the number of times f' returned the correct output value when executed with a given input from the set of I/O examples X.
- *Output Coverage* ($cov(f')$): The ratio of the number of distinct output values f' *correctly* returned compared to the number of distinct output values in X.

$$acc(f') = \frac{|\{i \mid f'(i) = o,\ \langle i, o\rangle \in X\}|}{|X|} \qquad cov(f') = \frac{|\{o \mid f'(i) = o,\ \langle i, o\rangle \in X\}|}{|\{o \mid \langle i, o\rangle \in X\}|}$$

In addition to accuracy, output coverage is a useful evaluation metric to avoid potential bias from the output distribution of X. For example, if X contains 99 `0`s out of 100 as output values, a FS $f' \equiv$ `0` will achieve 99% accuracy, while its output coverage will only be 50%.

If f' is evaluated with unsatisfactory quality, our iterative process tries to select more meaningful I/O examples from X to improve it, while refraining from selecting too many I/O examples, with the following selection rules:

- Given a set of input values that revealed differences when applied to f and f', $I_{diff} = \{i \mid f(i) \neq f'(i), \langle i, o\rangle \in X\}$, let $O_{diff} = \{f(i) \mid i \in I_{diff}\} \setminus \{f(i) \mid f(i) = f'(i), \langle i, o\rangle \in X\}$ be a set of output values that are *not* covered by f'.
- If f' did not achieve 100% output coverage, the iterative process selects I/O examples from $\{\langle i, f(i)\rangle \mid i \in I_{diff},\ f(i) \in O_{diff}\}$, i.e., from those that were not covered by $f'(i)$ when applied to input values in I_{diff}.
- If f' achieved 100% output coverage, we replace an I/O constraint in C with a new one from $\{\langle i, f(i)\rangle \mid i \in I_{diff}\}$ to balance the number of I/O constraints.

After synthesizing a new FS f' using the above selection rules, the current best FS f'_{best} is updated with f' if (1) $cmplx(f') \leq cmplx(f)$, and (2) either $cov(f') > cov(f'_{best})$ or $cov(f') = cov(f'_{best}) \wedge acc(f') > acc(f'_{best})$. We put higher priority on $cov(f'_{best})$ to $acc(f'_{best})$ when comparing similarity. We choose this strategy mainly because auxiliary functions used in embedded software tend to have limited output ranges, and thus, it is easier to achieve.

3.2.2 Output Range Analysis and FS Annotation. At the end of the iterative process, *Output Range Analysis* is conducted to guard the valid output range of f' because PBE may synthesize FSs with unreasonable output ranges by overfitting the given examples. To give an extreme case: A FS `fs_process(x)` $\equiv$ `x` for `process` (shown in Fig. 2) with one I/O constraint $\langle\{\texttt{x} \mapsto 0\}, 0\rangle$ is may be synthesized. Note that `fs_process(x)` has 100% accuracy and output coverage according to our metric. However, its output varies from MAX_INT to MIN_INT while process's output values range from 0 to 2. To guard the valid output range for FSs, we employ the static value range analyzer EVA [22], an extension of Frama-C [31], to compute the output ranges of the target function. In Fig. 3 (c), the assume statement[3] in fs_check is the result of applying EVA.

3.3 Selective Abstraction

We decide on the functions to be abstracted by distinguishing auxiliary functions from the main control logic in a program. The main control logic has a significant impact on the entire program, and thus, applying aggressive abstraction is not recommended. As an illustration, consider the function update in Fig. 2, which updates the global variable speed, whose values have a direct impact on the transition condition from line 7 to line 8 or to line 9, and thus on the satisfaction of the property in line 10. We classify update as a component of the main control logic.

3.3.1 Identifying Auxiliary Functions. We start from basic terminologies used in program analysis.

Definition 3. *Terminologies*

(1) *Control Flow Graph*: Given a function f, a control flow graph (CFG) $G_f = (N_f, E_f, s_f, T_f)$ is a directed graph in which N_f is a set of nodes that represent statements of f, E_f is a set of edges that represent control flow between nodes, s_f is a unique entry node, and T_f is a set of end nodes.
(2) *Def-Use*: Given a CFG G_f and a node $n \in N_f$, $def(n)$ is the set of variables defined (updated) at n, and $use(n)$ is the set of variables used at n.
(3) *Data Dependency*: A node n is *data-dependent* on node m (i.e., $m \in dd(n)$) if there exists a variable v such that $v \in def(m) \cap use(n)$, and if there exists a path π from m to n such that for every node $m' \in \pi - \{m, n\}$ and $v \notin def(m')$ [49].

For example, the CFGs of each function in Fig. 2 are illustrated in Fig. 3 (a). Node $3 \in dd(4)$ as in $\in def(3) \cap use(4)$, and there is a path from node 3 to 4 without interleaving definitions.

Definition 4. *Auxiliary Functions.* Let I_f be a set of input variables and O_f be a set of output variables for a function f. We say f is an *auxiliary function* if

(1) $I_f \neq \emptyset$ and $O_f \neq \emptyset$, and
(2) there is no global variable in O_f.

The condition specified in Definition 4 (1) suggests that f should have input and output variables. This condition is necessary to generate a SyGuS-PBE specification as PBE requires I/O relations as semantic constraints. Here, I_f consists of formal parameters, global variables used within f, and variables defined by external function calls (read_sensor()). In Fig. 2, for example, variables a and b are the input variables of check as they are formal parameters. O_f consists of a temporary variable for the return value of f and global variables[4] whose values are defined within f. The condition (2) excludes any functions updating global variables from being auxiliary functions. Functions main and update are not auxiliary functions because they update the global variable speed. Thus, auxiliary functions in Fig. 2 are process and check.

All auxiliary functions are candidates for PBE-based FSs, except when they are only invoked *deterministically* or *independently*. We define them using the notion of *function call context*.

Definition 5. *Function Call Context.* Let $\hat{f}$ be a function call for a function f with a set of actual parameters A invoked at a node n_f, then its call context $ctx(\hat{f})$ is recursively defined as

[3] assume(expr) is a macro of if(!expr) exit(0);.
[4] We note that pointer variables are treated as global variables.

(1) $ctx_0(\hat{f}) = dd(n_f, I_f[A/F]) \cup \{n_f\}$
(2) $ctx_i(\hat{f}) = \{m \mid m \in dd(n),\ n \in ctx_{i-1}(\hat{f})\} \cup ctx_{i-1}(\hat{f})$,

where $dd(n_f, I_f[A/F])$ is a set of nodes on which n_f is data-dependent w.r.t a set of variables $I_f[A/F]$, which represents the replacement of the formal parameters F within I_f with the actual parameters A within $\hat{f}$.

The function call context ctx is the fixed point of ctx_i, which means the set of nodes that contributed to determining the input values of $\hat{f}$. For example, the function call `check(speed,data)` at line 6 in Fig. 2 has the call context $ctx = \{3, 4, 6, 26, 28, 29\}$, which is defined by $ctx_0 = dd(6, \{\texttt{speed}, \texttt{data}\}) \cup \{6\} = \{4, 6, 26, 28, 29\}$ and $ctx_1 = \{3 \in dd(4)\} \cup ctx_0$.

Definition 6. *Deterministic Function Calls.* A function call $\hat{f}$ is *deterministic* if there is no node $n \in ctx(\hat{f})$ such that n uses free variables, where a free variable means a variable whose values are assigned by external objects, e.g., user inputs and external functions.

We do not replace any deterministic function calls with FS calls because there is little room for abstraction. The function call `check(0,0)` at line 1 in Fig. 2 is deterministic because there is no reference to free variables in its call context, but `check(speed,data)` at line 5 can be replaced with its PBE-based FS call because its function call context contains node 4, which uses a free variable *in*.

Definition 7. *Independent Function Calls.* A function call $\hat{f}$ is *independent* if $\hat{f}$ is not deterministic, and there is no node n outside of f and $ctx(\hat{f})$ such that n uses a variable defined in $ctx(\hat{f})$.

If a function call is independent, we can freely explore the input space of the function call without affecting the other parts of the program, thereby speeding up the falsification process using non-determinism without necessarily constructing PBE-based FSs. Therefore, we abstract independent function calls with non-deterministic FSs, which are constructed by (1) assigning a non-deterministic value using function `nondet()` and (2) guarding the output values with an output range analysis using EVA [22]. As an illustration, Fig. 3 (c) shows an non-deterministic FS for the function `process` that non-deterministically returns 0, 1, or 2. Also, there is no concept of accuracy in the non-deterministic FS, and its complexity is always 0.

3.3.2 FS Selection on Target Program. We perform PBE-based FS construction for all the auxiliary functions, and produce a set of FSs $F' = \{f', g', \dots\}$ for the target program P, from which the best subset $S'_{best} \subset F'$ is selected by scoring each subset of FSs.

Scoring is designed with three critical factors to consider: (1) *Complexity*, as FSs with lower complexity than their original functions can reduce falsification cost; (2) *Accuracy*, as FSs with higher accuracy are less likely to generate false alarms; and (3) *Output Coverage*, as FSs with higher output coverage are more effective in falsification.

We also consider the number of times each function is invoked within P because even though a FS f' for f has lower complexity gain than another FS g' for g, i.e., $cmplx(f) - cmplx(f') < cmplx(g) - cmplx(g')$, using f' for abstracting f may result in more overall gain if f is called more often than g within P. Thus, we define our scoring function for each subset of FSs S' as

$$score(S') = \alpha \cdot (1 - \mathtt{norm}(cmplx(P_k^{S'}))) + \beta \cdot \mathtt{norm}(acc(P_k^{S'})) + \gamma \cdot \mathtt{norm}(cov(P_k^{S'})),$$

where $P_k^{S'}$ is P whose loops are unrolled k times after substituting S', and `norm` is a min-max normalization to consider complexity, accuracy, and output coverage on the same scale. α, β, and γ are adjustable constants (and will be assigned in Section 5.3). We now break down these three elements:

$$cmplx(f) = |B(f)| + \sum_{\hat{g} \in down(f)} cmplx(g)$$

$$acc(P_k^{S'}) = \frac{\sum_{f' \in S'} acc(f') \cdot \mid call(f', P_k^{S'}) \mid}{\sum_{f' \in S'} \mid call(f', P_k^{S'}) \mid} \qquad cov(P_k^{S'}) = \frac{\sum_{f' \in S'} cov(f') \cdot \mid call(f', P_k^{S'}) \mid}{\sum_{f' \in S'} \mid call(f', P_k^{S'}) \mid},$$

where $B(f)$ is a set of branches in f, $down(f)$ is a set of function calls invoked in f, and $call(f', P_k^{S'})$ is a set of function calls for f' in $P_k^{S'}$. $cmplx(f)$ is the sum of all branches in f and in functions called by f, and thus, $cmplx(P_k^{S'}) \equiv cmplx(\texttt{main})$ for the entry function of $P_k^{S'}$ `main`. $acc(P_k^{S'})$ (and $cov(P_k^{S'})$) are defined as the sum of the products of each f''s invocation count and its corresponding accuracy (and output coverage), divided by the total FS invocation count within $P_k^{S'}$.

4 COUNTEREXAMPLE-GUIDED ALARM FILTERING AND REFINEMENT

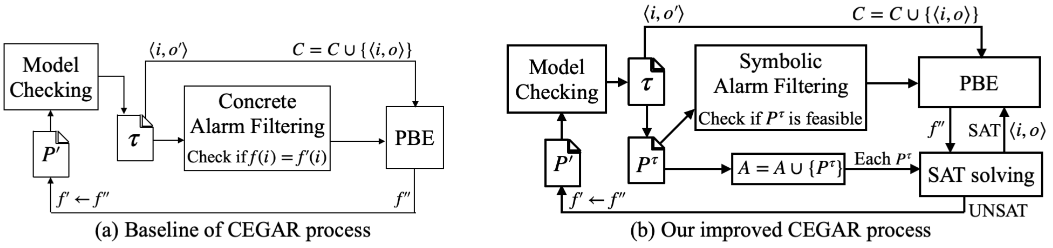


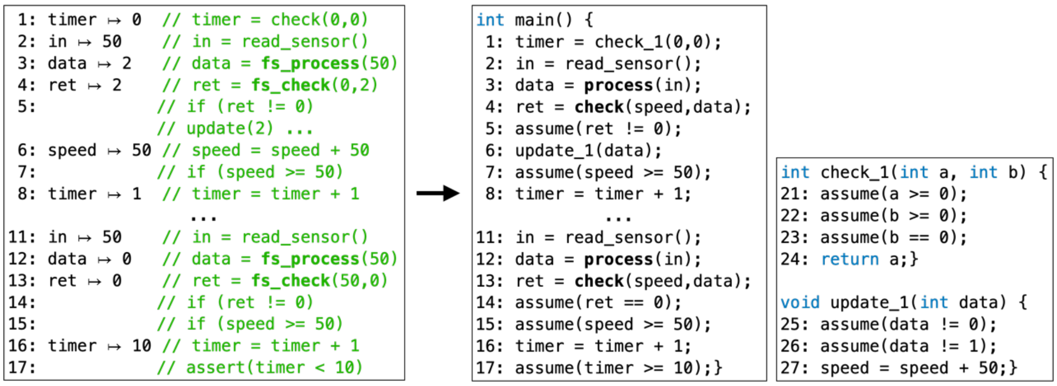
Fig. 6. Difference between the baseline of CEGAR process and our improved process.

After we abstract a target program P into an abstracted program P' using selective abstraction, we employ a bounded model checker to falsify a property of P within a bound k. However, falsification using selective abstraction may produce false alarms, as illustrated in Fig. 3 (d). Identification and removal of false alarms are critical factors in efficient falsification. We address this issue using CEGAR [24], which is specially designed for the efficient use of PBE.

A baseline for PBE-based FS refinement, following the well-known CEGAR approach, is shown in Fig. 6 (a). Given a counterexample trace τ generated from model checking P', which is abstracted by a PBE-based FS f' for a function f, it performs *concrete alarm filtering*, which checks whether the output value of f' (i.e., $f'(i) = o'$) in τ is the same as that of f (i.e., $f(i) = o$). If not (i.e., $o \neq o'$), the corrected I/O pair $\langle i, o \rangle$ in τ is used to refine f' into f''.

However, there are a couple of drawbacks to this baseline. Firstly, concrete alarm filtering only checks FS invocations in a concrete execution trace (computed by a corresponding program input) and might miss the opportunity to find true alarms with different inputs within the same path condition. For example, suppose a function f is replaced with its FS f', where $f(1) = 0$, $f'(1) = 0$, $f(2) = 1$, and $f'(2) = 0$, in a program fragment `[int x; assert(`f'`(x)!=0);]`. Then, there are two input values x = 1 and x = 2 that lead to the assertion failure in the same path, but the case with x = 2 is a false alarm. Secondly, PBE-based FS refinement does not ensure what CEGAR-based refinement approaches, such as interpolation-based refinement, do: avoiding re-exploration of previously explored infeasible paths. Though the refined FS guarantees that a specific set of I/O examples is satisfied, it can still contain other incorrect behaviors that lead to exploring the previously identified infeasible paths.

Thus, we improve the baseline process by employing symbolic alarm filtering [40] to achieve efficient true/false alarm identification as well as effective refinements of FSs with SAT solving the feasibility of symbolic program paths constructed from counterexample traces.



```
 1: timer ↦ 0   // timer = check(0,0)
 2: in ↦ 50     // in = read_sensor()
 3: data ↦ 2    // data = fs_process(50)
 4: ret ↦ 2     // ret = fs_check(0,2)
 5:            // if (ret != 0)
               // update(2) ...
 6: speed ↦ 50 // speed = speed + 50
 7:            // if (speed >= 50)
 8: timer ↦ 1  // timer = timer + 1
                 ...
11: in ↦ 50     // in = read_sensor()
12: data ↦ 0    // data = fs_process(50)
13: ret ↦ 0     // ret = fs_check(50,0)
14:            // if (ret != 0)
15:            // if (speed >= 50)
16: timer ↦ 10 // timer = timer + 1
17:            // assert(timer < 10)
```

(a) Counterexample trace

```
int main() {
 1: timer = check_1(0,0);
 2: in = read_sensor();
 3: data = process(in);
 4: ret = check(speed,data);
 5: assume(ret != 0);
 6: update_1(data);
 7: assume(speed >= 50);
 8: timer = timer + 1;
          ...
11: in = read_sensor();
12: data = process(in);
13: ret = check(speed,data);
14: assume(ret == 0);
15: assume(speed >= 50);
16: timer = timer + 1;
17: assume(timer >= 10);}
```

```
int check_1(int a, int b) {
21: assume(a >= 0);
22: assume(b >= 0);
23: assume(b == 0);
24: return a;}

void update_1(int data) {
25: assume(data != 0);
26: assume(data != 1);
27: speed = speed + 50;}
```

(b) Path program

Fig. 7. Snippet of counterexample trace and path program for Fig. 3 (d).

4.1 Symbolic Alarm Filtering

In our context, a counterexample trace $\tau = \langle a_1, ..., a_n \rangle$ is a sequence of variable assignments that leads to a violation of a given property ψ. Fig. 7 (a) shows a snippet of a counterexample trace generated from model checking P' illustrated in Fig. 3 (c); in each variable assignment, we annotate its corresponding original statement. Variable `ret` at lines 4 and 13 is a temporary variable containing the return value of FS `fs_check`. Empty spaces in lines 5, 7, 14, 15, and 17 exist because they are branches or assertions, so there are no assignments.

From the counterexample trace τ, we construct a *path program* P^τ [9], a finite path through P. Fig. 7 (b) illustrates a snippet of P^τ from Fig. 7 (a), where assignment statements are reverted from their corresponding variable assignments in τ, `assume` statements represent the particular branches taken (particularly, we consider that the violated assertion at line 17 is a branch taken as false), and `check_1` and `update_1` are defined to reflect each execution path after their invocation, respectively. To construct P^τ from τ, we instrument each statement in P' (specifically, we instrument FS call statements into their original function call statements) and execute P' with the program input extracted from τ. Note that the FS calls at lines 3, 4, 12, and 13 are reverted to the original function calls in order to check the feasibility in the original program.

After that, *symbolic alarm filtering* checks whether there is a program input that reaches the end of P^τ (the program point after the last `assume` statement), which is computed by a SAT solver. If we find such program input, though the execution trace may be different from τ, it is a true alarm. Otherwise, we conclude that τ is spurious (false alarm), and P^τ is infeasible in the original program. Symbolic alarm filtering allows us to check the feasibility of a program path by considering all possible program inputs along that path, whereas concrete alarm filtering does not.

For example, concrete alarm filtering only checks whether FS invocations in Fig. 7 (a) are incorrect. `fs_check(50,0)` at line 13 returned an invalid output, thus Fig. 7 (a) is a false alarm. On the other hand, symbolic alarm filtering can determine that Fig. 7 (b) is infeasible, as line 27 within the function `update_1` invoked at line 6 constrains the value of `speed` to be 50, and there is no opportunity of returning 0 from `check(50,data)` at line 13, even though `data` can be freely determined from 0 to 2 as the `process` call at line 12 is independent. Therefore, the `assume` statement at line 14 cannot be satisfied, and thus, the end point of P^τ cannot be reached.

Algorithm 1 Improved PBE-based Function Summary Refinement

```
1: procedure refine($f$: original function, $f'$: function summary, $\tau$: counterexample trace, $P^\tau$: infeasible path program)
2:     $A \leftarrow A \cup \{P^\tau\}$; // accumulating infeasible path program
3:     $R \leftarrow$ correct($f, f', \tau$) // refining I/O constraints
4:     SAT ⟵ true;
5:     while SAT == true do
6:         $C \leftarrow C \cup R$; // existing I/O constraints
7:         $f'' \leftarrow$ runPBE($C$); // PBE-based FS refinement
8:         SAT ⟵ false; // UNSAT
9:         for each $P^\tau \in A$ do
10:            if ($\tau' \leftarrow$ checkSAT($P^\tau[f''/f]$)) $\neq \emptyset$ then
11:                $R \leftarrow R \cup$ correct($f, f'', \tau'$);
12:                SAT ⟵ true;
13:    return $f''$
```

4.2 PBE-Based Function Summary Refinement with SAT Solving

As we have abstracted only auxiliary functions, false alarms must inevitably arise from inaccurate FSs, which is unavoidable when using PBE. Therefore, we focus on the refinement of FS by imposing more I/O constraints identified in the spurious counterexample trace τ. Our improved PBE-based FS refinement approach (Fig. 6 (b)) uses a path program P^τ found to be infeasible through symbolic alarm filtering as well as the I/O information identified from τ. It first finds the incorrect I/O pair $\langle i, o' \rangle$ from τ for each FS f' and corrects it to the valid I/O pair $\langle i, o \rangle$, where $f(i) = o$. Then it re-synthesizes f' to f'' by adding the valid I/O pair to the existing I/O constraints C.

This refined FS f'' guarantees that correct output is produced for the given i, which may be enough for concrete alarm filtering and refinements. However, it does not guarantee that a counterexample generated from checking the program with f'' will not produce the same path program P^τ, with a high risk of having spurious cycles of false alarm filtering. This is because the refined f'' might be a completely different program code than the previous f'. Thus, there is a possibility of re-exploring previously explored infeasible paths when replacing f' with f'' if we do not use the path information identified from symbolic false alarm filtering for the refinements.

Our improved refinement process removes this potential risk by rechecking the feasibility of P^τ after replacing the original function f with f''. Note that we have already proved using symbolic alarm filtering that P^τ is infeasible under arbitrary program inputs. Therefore, if it turns out to be feasible when using f'', the reason must be that f'' is still producing an incorrect output value (for input values different from i). Our refinement process uses this information to further refine f'' before using it for the falsification of the program.

Algorithm 1 is a pseudo code of our improved FS refinements, assuming that a single FS is used for the sake of simplicity; given an original function f, its FS f', a counterexample trace τ, and an infeasible path program P^τ for τ, the algorithm

(1) accumulates the infeasible path program P^τ into a set A (line 2),
(2) collects the wrong I/O pairs of f' invoked in τ and corrects these pairs by executing f (correct). These corrected pairs are added to R, the set of refining I/O constraints (line 3);
(3) updates the set of I/O constraints $C = C \cup R$ that was used to synthesize f' (line 6);
(4) refines f' into f'' using the PBE solver (runPBE) with the updated C (line 7);
(5) checks whether f'' makes any infeasible path $P^\tau \in A$ feasible (checkSAT) after replacing f with f'' in each P^τ, (lines 9 and 10);

  (a) if the replaced path program is feasible (SAT), extracts the incorrect I/O pairs from its witness τ' and adds its corrected I/O to R (line 11);

(6) repeats steps (3)–(5) until all checking P^τ is UNSAT; and

(7) returns f'' if all the path programs in A are infeasible when replaced with f''.

Example 1. Refinement of `fs_check`

(1) Collect and correct the wrong I/O pair $\langle\{\mathsf{a} \mapsto 50, \mathsf{b} \mapsto 0\}, 0\rangle$ from Fig. 7 (a) to $\langle\{\mathsf{a} \mapsto 50, \mathsf{b} \mapsto 0\}, 50\rangle$ by executing `check(50,0)`.

(2) Suppose that `fs_check` was synthesized by the set of I/O constraints C in Fig. 4. Then $C = C \cup \{\langle\{\mathsf{a} \mapsto 50, \mathsf{b} \mapsto 0\}, 50\rangle\}$, thus `fs_check` is refined into $\texttt{fs\_check}(\mathsf{a}, \mathsf{b}) \equiv \texttt{if b < 0 then } -\mathsf{b} \texttt{ else } \mathsf{a} - \mathsf{b}$.

(3) Check whether the refined `fs_check` still reaches the end of the infeasible P^τ (Fig. 7 (b)) by replacing `check` with `fs_check`. Fortunately, even though another wrong I/O pair $\langle\{\mathsf{a} \mapsto 50, \mathsf{b} \mapsto 1\}, 49\rangle$ still exists, it does not make the infeasible P^τ feasible.

(4) Return the refined `fs_check`, which guides the model checker to explore another path.

5 EXPERIMENTS

We implemented our approach named PBEAR (**PBE**-based Selective **A**bstraction and **R**efinement) in Java, using the concolic testing tool CROWN [54] for collecting I/O examples, the SyGuS-PBE solver DUET[5] [44] for synthesizing FSs, the static value analyzer Frama-C EVA [22] for guarding the output value range of FSs, and the bounded model checker CBMC [25] for property falsification and SAT solving. Static analysis such as def-use analysis was performed on the source code pre-processed by CBMC.

5.1 Research Questions

RQ1. Falsification efficiency of PBEAR: How fast does PBEAR find true alarms, or how much less memory does PBEAR use, compared to state-of-the-art abstraction techniques?

For RQ1, we compared PBEAR with:

- CBMC (v5.43) [25]: The baseline (PBEAR also uses CBMC v5.43).
- CBMC-refine (v5.43) [20]: CBMC with SAT-based abstraction. It abstracts floating-point formulas. We chose it due to the presence of several float-typed expressions in our target programs.
- CPAchecker (v2.1) [17]: The winner of *FalsificationOverall* in SV-COMP 2022 [11]. We configured it to use lazy predicate abstraction with Craig interpolants.

RQ2. Effectiveness of our improved CEGAR process: How much time is saved through our improved CEGAR process compared to the baseline process?

For RQ2, we implemented PBEAR$^{\mathrm{b}}$, which is a baseline of PBEAR that follows Fig. 6 (a).

RQ3. Effectiveness of using output coverage for FS synthesis and selection: Does PBEAR effectively identify violations by using output coverage for FS synthesis and selection?

For RQ3, we implemented PBEAR$^{\text{-cov}}$, which is a variant of PBEAR that synthesizes and selects FSs without considering output coverage, prioritizing accuracy instead. In comparison to PBEAR, there is no replacement of I/O constraints in the I/O constraint selection strategy for PBEAR$^{\text{-cov}}$.

5.2 Target Programs and Properties

We used two sets of programs for our experiments: the SV-COMP 2022 benchmarks [11] for fair comparison of PBEAR with those tools known to be the most efficient on the benchmarks, and three typical embedded software with 15 safety properties for evaluating the performance of PBEAR.

[5]Any other SyGuS-PBE tools can be applied, without the need to specifically use DUET.

5.2.1 SV-COMP Benchmarks. We selected 6 categories from the SV-COMP benchmarks and chose 2 or 3 target programs per category (a total of 16 target programs) that share the same set of functions within their respective categories (to reuse synthesized FSs) and have at least one auxiliary function with *ReachSafety* property (i.e., checking reachability of an error state). While 6 out of the 16 target programs are derived from SystemC [19] programs, which are reactive, the others do not show any characteristics of embedded software. Therefore, they may not be the best choice for demonstrating the merits of PBEAR, but we intend to show that PBEAR also performs as well as other tools on general programs[6].

5.2.2 Embedded Software Examples. As PBEAR is specifically designed for efficient falsification of embedded software, we chose three typical embedded programs in the domains of automotive, hardware control, and robotics to evaluate its performance.

P1. An object-following automotive multitasking program [30],
P2. an elevator controller program [27], and
P3. a garbage collector robot program [21].

Each program is written in about 1,000 (P1), 900 (P2), and 600 (P3) lines of code in C. Though their sizes in terms of lines of code are smaller compared to the SV-COMP benchmarks, they operate within an infinite loop using multiple global variables and external inputs, showing typical characteristics of embedded software. Furthermore, safety or liveness properties for these programs require exploring multiple time steps, which increases their verification complexity. Table 1 is a partial list of properties for each program we used for experiments[7]. These properties are specified as monitoring code in the target program that is checked at the end of every infinite loop iteration.

Table 1. Examples of properties for each program.

P	ψ	Description
P1	ψ_{12}	It shall not suddenly stop at a speed less than 10 when it drives at a speed exceeding 70.
	ψ_{16}	It shall be backed up within 10 iterations if the distance from the car in front is 84 or more.
P2	ψ_{22}	It shall stop at that floor within 600 iterations when the `UP_BUTTON` is pressed on any floor.
	ψ_{24}	It shall stop at the target floor within 600 iterations when any `FLOOR_BUTTON` is pressed in the elevator.
P3	ψ_{31}	It shall reach all 10 states (at least 10 iterations are required).
	ψ_{32}	It shall throw that garbage away within 15 iterations when it captures any garbage.

5.3 PBEAR Setup

- Collection of I/O Examples: We collected up to 10,000 I/O examples for each auxiliary function using the concolic testing tool CROWN [54] and mutating the test inputs generated by the tool. As CROWN generates test inputs for maximum branch coverage, the generated inputs may initiate diverse behaviors of the auxiliary functions.
- Options for FS Synthesis: We used DUET [44] for FS synthesis with the timeout for each DUET run set to 300 seconds, and the total timeout for the iterative synthesis set to 1 hour. We extended the timeout for each DUET run to 600 seconds during PBE-based FS refinement, intended to help DUET address additional I/O constraints.
- FS Evaluation: We scored subsets of FSs with at most five PBE-based FSs each due to their vast number of possible combinations. We selected the highest-scoring subset of FSs for replacement

[6]In fact, it was the best choice we could make because of the lack of benchmarks in embedded software.
[7]These are artificial properties imitating functional requirements we may encounter in practice.

by increasing the bound k ($k = 1, 2, 4, \ldots$) until the score rankings ($score(P_k^{S'})$) from 1st to 5th place stabilized. We assigned $\alpha = 2$, $\beta = 1$, and $\gamma = 3$, as we prefer high output coverage.

- Falsification: We started BMC with a bound $k = 10$ and incremented it by 10 for each run up to $k = 1000$. This setup also aligned with CBMC and CBMC-refine.
- Reuse of FSs: We reused FSs for the same function because, in our target programs within the same category in SV-COMP, multiple auxiliary functions are identical. We also used the same FSs when falsifying different properties for each embedded software.

All experiments were performed on a 3.3-GHz Intel Xeon Gold 6234 CPU with 200 GB RAM, running the Ubuntu 20.04 64-bit version. Each experiment was terminated if it timed out, encountered an out-of-memory situation or runtime errors, or found a true alarm.

We note that the time and memory usages for selective abstraction were not considered in our measurement because it needs to be conducted once before the iterative falsification process and can be reused for checking multiple properties.

5.4 RQ1. Falsification Efficiency of PBEAR

Table 2. Property falsification results on SV-COMP benchmarks.

P	$\|F\|$	PBEAR					CBMC			CBMC-refine				CPAchecker			
		$\|S'\|$	#R	T(s)	M(GB)	V?	T(s)	M(GB)	V?	#R	T(s)	M(GB)	V?	#R	T(s)	M(GB)	V?
gcd_1+newton_1_6		1	2	79.0	0.05	O	7.0	0.07	O	32	5.9	0.06	O	4	26.6	0.24	O
gcd_3+newton_1_4	1	1	0	39.9	0.09	O	7.4	0.07	O	35	77.6	0.10	O	4	102.8	0.25	O
gcd_3+newton_3_7		1	0	126.6	0.22	O	61.6	0.19	O	47	229.6	0.38	O	4	195.8	0.44	O
email_spec4		2	0	3.9	0.18	O	8.0	0.36	O	0	7.6	0.36	O	6	172.2	1.95	O
email_spec7	8	2	0	5.2	0.22	O	8.9	0.38	O	0	8.5	0.38	O	7	281.1	1.93	O
email_spec27		2	0	61.7	0.32	O	31.3	0.50	O	0	30.7	0.50	O	3	>900s	1.37	X
Fibonacci04	1	1	0	0.1	0.01	O	3.5	0.26	O	0	3.3	0.26	O	0	1.1	0.16	O
Fibonacci05		1	0	4.2	0.29	O	3.8	0.28	O	0	3.5	0.29	O	0	1.1	0.15	O
floppy_simpl3.cil-1	8	8	0	0.2	0.01	O	0.1	0.01	O	0	0.1	0.01	O	2	2.3	0.25	O
floppy_simpl4.cil-1		8	0	0.8	0.02	O	0.2	0.03	O	0	0.2	0.03	O	2	2.7	0.25	O
pals_floodmax.5.1		2	1	3.0	0.06	O	0.9	0.07	O	0	0.9	0.07	O	5	413.7	0.85	O
pals_floodmax.5.4-10	2	2	3	110.8	0.06	X	1.1	0.07	O	0	1.0	0.07	O	6	433.8	0.74	O
pals_floodmax.5.4		2	3	8.2	0.06	O	2.0	0.07	O	0	1.9	0.07	O	6	>900s	0.80	X
token_ring.01.cil-2		1	0	2.1	0.12	O	1.4	0.11	O	0	1.3	0.11	O	3	2.1	0.15	O
token_ring.05.cil-2	1	1	0	7.2	0.35	O	5.5	0.34	O	0	5.1	0.35	O	3	9.2	0.26	O
token_ring.14.cil		1	0	32.9	1.06	O	34.4	1.07	O	0	31.9	1.07	O	2	>900s	0.44	X
Comparison with PBEAR on time and memory usage[1]							0.47x	1.24x	-	-	1.09x	1.31x	-	-	4.45x	3.27x	-

[1] We only compared the time usage for the 'O' properties. Timeouts (>900s) are not factored into the comparison.

5.4.1 *Results on Benchmark Programs.*

Table 2 shows the falsification result on 16 SV-COMP programs[8] using PBEAR, CBMC, CBMC-refine, and CPAchecker. From left to right, the target program (P), the number of auxiliary functions in P ($|F|$), and the name of each tool are listed. Each column under the tool name represents the number of selected auxiliary functions ($|S'|$, only for PBEAR) using the scoring function, the number of refinements (#R) performed, time usage for falsification (T) in seconds, peak memory usage (M) in GBytes, and the result of falsification (V?, where 'O' indicates a true alarm found, and 'X' indicates unknown).

Overall, CBMC performed best for these 16 programs as it was the fastest in finding all true alarms. All the other tools were less efficient than CBMC because their overhead for refinement

[8]In certain instances, names are shortened. For example, "email_spec" corresponds to "email_spec_productSimulator.cil", and "pals_floodmax" was shortened by omitting post-fixes such as ".ufo.UNBOUNDED.pals" and ".ufo.BOUNDED-10.pals".

outweighed the benefits of abstraction. PBEAR used the least amount of memory among all the tools but failed to identify a true alarm for pals_floodmax.5.4-10 after three iterations of FS refinements and falsification attempts up to bound $k = 1000$ without finding any violations. This is an example showing that PBEAR may always not able to find true alarms due to the use of PBE-based FSs which is neither sound nor complete. We note that the experiment using CPAchecker was conducted using the `-predicateAnalysis` option instead of the `-svcomp22` option because the former performed faster in finding true alarms. If we had used the latter option, it would have falsified two more programs but would have been 1.3 times slower with 2.9 times more memory consumption.

Table 3. Property falsification results on the three embedded software programs.

P	$\|F\|$	ψ	k	PBEAR					CBMC			CBMC-refine				CPAchecker			
				$\|S'\|$	#R	T(s)	M(GB)	V?	T(s)	M(GB)	V?	#R	T(s)	M(GB)	V?	#R	T(s)	M(GB)	V?
P1	5	ψ_{11}	60	1	0	2,198	23.4	O	11,595	26.0	O	726	2,657	24.1	O	49	>3d	33.9	X
		ψ_{12}	110	1	0	33,915	70.2	O	13,976	OOM	X	1,536	29,822	75.2	O	238	>3d	6.5	X
		ψ_{13}	60	1	0	2,674	23.3	O	11,739	25.9	O	591	2,517	24.1	O	50	>3d	25.1	X
		ψ_{14}	50	5	1	1,605	14.3	O	2,994	15.8	O	289	875	14.9	O	49	>3d	9.7	X
		ψ_{15}	70	1	0	3,859	32.3	O	27,675	36.5	O	780	4,998	33.7	O	230	>3d	13.6	X
		ψ_{16}	120	1	0	47,591	80.0	O	14,741	OOM	X	0	36,574	OOM	X	248	>3d	15.9	X
		ψ_{17}	50	1	0	648	14.4	O	2,934	15.9	O	288	954	14.9	O	51	>3d	9.1	X
		ψ_{18}	60	1	0	2,273	23.3	O	8,330	25.8	O	588	2,391	24.1	O	64	>3d	70.5	X
		ψ_{19}	110	1	0	50,982	70.3	O	20,349	OOM	X	1,555	39,763	75.2	O	56	>3d	OOM	X
P2	3	ψ_{21}	310	3	0	1,533	4.4	O	22,682	66.7	O	7	23,183	65.8	O	426	>3d	8.3	X
		ψ_{22}	610	3	30	39,091	12.4	O	30,056	OOM	X	0	31,326	OOM	X	418	>3d	7.1	X
		ψ_{23}	610	3	27	40,881	13.9	O	29,205	OOM	X	0	30,631	OOM	X	418	>3d	6.7	X
		ψ_{24}	610	3	19	>3d	26.0	X	105,882	OOM	X	1	173,323	OOM	X	68	>3d	2.8	X
P3	1	ψ_{31}	20	1	0	151	3.9	O	3,357	13.2	O	0	3,488	13.2	O	32	27	0.9	O
		ψ_{32}	30	1	0	751	11.4	O	43,156	43.1	O	0	43,737	43.1	O	159	1,022	4.8	O
Comparison with PBEAR on time and memory usage[1]									8.57x	1.77x	-	-	1.53x	1.72x	-	-	1.16x	0.70x	-

[1] OOM means that a tool ran out of memory. We consider OOM as 80 GB when comparing memory usage.

5.4.2 Results on Embedded Software. Table 3 shows the falsification results for the three embedded software programs. Similar to Table 2, this table includes additional columns where ψ represents properties, and k denotes bounds for loop unwinding when a violation of ψ occurs. Compared to the results on the SV-COMP benchmark, CBMC did not perform well on these three programs, with falsification time being 8.57 times longer and 6 OOM occurrences. This means that the characteristics of embedded software and their properties require abstraction and refinement. CBMC-refine outperformed PBEAR in P1, as the falsification time was 1.17 times faster after 6,353 refinements. However, it still suffered from scalability issues. OOM occurred for properties ψ_{16}, ψ_{22}, ψ_{23}, and ψ_{24}. Overall, CBMC-refine took 1.53 times more falsification time and 1.75 times more memory compared to PBEAR. CPAchecker used less memory compared to PBEAR (0.7 times). However, it found only 2 true alarms after a total of 2,556 refinements.

On average, PBEAR took 16,297 seconds and 28 GB to find each true alarm (except for ψ_{24}). The fastest case among them was ψ_{31}, where by replacing independent function calls with non-deterministic FS calls. However, ψ_{24} took more than 3 days of falsification time without finding a true alarm using PBEAR, even though the number of refinements was smaller than ψ_{22} and ψ_{23}. This is because of the complexity of the property itself as shown in Table 1; ψ_{24} requires to explore all the execution paths making the elevator reach the floor, pick up passengers, and then reach another floor where any `FLOOR_BUTTON` is pressed, to reason about it anyways, whereas ψ_{22} only requires the elevator to reach the floor to which the `UP_BUTTON` is pressed. Due to this complexity, none of the tools were able to falsify this property.

5.5 RQ2. Effectiveness of Improved CEGAR Process

Table 4. # of refinements, time usage (seconds), and # of detected true alarms of PBEAR and PBEAR^b.

P and ψ		PBEAR			PBEAR^b		
		#R	T(s)	V?	#R	T(s)	V?
gcd_1+newton_1_6		2	79.0	O	38	225.4	O
Fibonacci05		0	4.2	O	0	0.1	X
floppy_simpl4.cil-1		0	0.8	O	0	0.2	X
pals_floodmax.5.1		1	3.0	O	1	1.7	O
pals_floodmax.5.4-10		3	110.8	X	3	102.5	X
pals_floodmax.5.4		3	8.2	O	3	3.8	O
P	ψ	#R	T(s)	V?	#R	T(s)	V?
P1	ψ_{11}	0	2,198	O	3	7,004	O
	ψ_{12}	0	33,915	O	6	175,218	O
	ψ_{13}	0	2,674	O	15	24,920	O
	ψ_{14}	1	1,605	O	1	1,637	O
	ψ_{16}	0	47,591	O	2	93,761	X
	ψ_{17}	0	648	O	53	38,714	O
	ψ_{18}	0	2,273	O	7	11,900	O
	ψ_{19}	0	50,982	O	2	92,889	O
P2	ψ_{22}	30	39,091	O	30	36,681	O
	ψ_{23}	27	40,881	O	27	40,762	O
	ψ_{24}	19	>3d	X	19	>3d	X
P3	ψ_{31}	0	151	O	0	155	X
	ψ_{32}	0	751	O	0	750	X

Table 5. # of refinements, time usage (seconds), and # of detected true alarms of PBEAR and PBEAR$^{\text{-cov}}$.

P and ψ		PBEAR				PBEAR$^{\text{-cov}}$			
		$\lvert S'\rvert$	#R	T(s)	V?	$\lvert S'\rvert$	#R	T(s)	V?
gcd_1+newton_1_6		1	2	79.0	O	1	1	9.7	O
gcd_3+newton_1_4		1	0	39.9	O	1	0	16.7	O
gcd_3+newton_3_7		1	0	126.6	O	1	0	355.4	O
email_spec4		2	0	3.9	O	2	0	4.0	O
email_spec7		2	0	5.2	O	2	0	5.2	O
email_spec27		2	0	61.7	O	2	0	62.4	O
Fibonacci04		1	0	0.1	O	1	0	0.1	O
Fibonacci05		1	0	4.2	O	1	0	4.2	O
pals_floodmax.5.1		2	1	3.0	O	2	1	>900s	X
pals_floodmax.5.4-10		2	3	110.8	X	2	1	98.7	X
pals_floodmax.5.4		2	3	8.2	O	2	1	>900s	X
P	ψ	$\lvert S'\rvert$	#R	T(s)	V?	$\lvert S'\rvert$	#R	T(s)	V?
P1	ψ_{11}	1	0	2,198	O	2	0	3,036	O
	ψ_{12}	1	0	33,915	O	2	0	15,917	X
	ψ_{13}	1	0	2,674	O	2	0	1,786	O
	ψ_{14}	5	1	1,605	O	6	3	2,310	O
	ψ_{15}	1	0	3,859	O	2	0	5,089	O
	ψ_{16}	1	0	47,591	O	2	0	2,739	X
	ψ_{17}	1	0	648	O	2	0	633	O
	ψ_{18}	1	0	2,273	O	2	0	2,104	O
	ψ_{19}	1	0	50,982	O	2	0	2,747	X
P2	ψ_{21}	3	0	1,533	O	3	30	26,492	O
	ψ_{22}	3	30	39,091	O	3	33	71,196	O
	ψ_{23}	3	27	40,881	O	3	3	12,382	O
	ψ_{24}	3	19	>3d	X	3	13	180,643	O

Table 4 presents a comparison between PBEAR, with symbolic alarm filtering and our improved PBE-based FS refinement with SAT solving, and PBEAR^b, with the baseline approach for concrete alarm filtering and PBE-based FS refinement. We omitted ψ_{15}, ψ_{21}, and several benchmarks in SV-COMP where symbolic alarm filtering was not conducted. Overall, PBEAR found 17 true alarms considerably faster with a lower number of refinements, whereas PBEAR^b found only 12.

In P1, PBEAR^b required additional 88 refinements to falsify 9 properties (and failed to falsify ψ_{16} due to the OOM issue after two refinements) compared to PBEAR, requiring an extra 71.7 hours. In the case of gcd_1+newton_1_6, PBEAR^b explored the same infeasible path during 38 refinement iterations, whereas PBEAR conducted only two refinements. In the case of Fibonacci05, PBEAR^b was unable to refine a FS due to an impractically large input value, leading to refinement failure with the overflow of its output value. In the cases of floppy_simpl4.cil-1 and ψ_{31} and ψ_{32} in P3, where independent function calls were replaced with non-deterministic FSs, we did not refine false alarms identified through concrete alarm filtering in PBEAR^b as it is obvious that it would take a long time to iteratively refine these FSs with concrete counterexamples.

5.6 RQ3. Effectiveness of Using Output Coverage for FS Synthesis and Selection

Table 5 shows the impact of using output coverage in determining falsification ability by comparing the results with (PBEAR) and without (PBEAR$^{\text{-cov}}$) using output coverage in the FS evaluation. We omitted P3 and two benchmarks in SV-COMP that did not use PBE-based FSs, but non-deterministic FSs. Overall, PBEAR found two more true alarms than PBEAR$^{\text{-cov}}$ with almost no overhead for the SV-COMP benchmark programs, three more true alarms for P1, but missed one true alarm for P2.

In the pals_floodmax.XX category, for example, a function commonly used in that category was synthesized and replaced with a FS with 98.95% accuracy and 100% output coverage by PBEAR, while PBEAR$^{-\text{cov}}$ synthesized the same function with 99.66% accuracy and 50% output coverage. In the case of P1, a FS for the function `fisqrt`, which was not selected by PBEAR due to its low output coverage (21.74%), was selected by PBEAR$^{-\text{cov}}$, resulting in missing true alarms for ψ_{12}, ψ_{16}, and ψ_{19}. In the case of P2, however, PBEAR$^{-\text{cov}}$ found a true alarm after only 13 refinements, whereas PBEAR encountered timeouts. PBEAR$^{-\text{cov}}$ did not explicitly consider output coverage, but it coincidentally achieved 100% output coverage in this case. Thus, with the same output coverage, it could achieve better effectiveness by improving accuracy solely without controlling the number of I/O constraints.

5.7 Discussion

As PBEAR is specialized in abstracting auxiliary functions, its application to SV-COMP benchmarks did not show better performance compared to that of CBMC; It could not find a true alarm for the pals_floodmax.5.4-10 benchmark in Table 2 and showed slower performance than CBMC, the fastest on those benchmark programs, for most of the cases. This shows that our approach implemented in PBEAR can be an overhead for checking (relatively) small-scale programs with simple properties. On the other hand, as shown with complex embedded software programs in Table 3, PBEAR solved all but one property, whereas CBMC encountered memory issues, and CPAchecker did not finish within 3 days in most cases. It shows that existing best-performing model checking tools are not quite suitable for falsifying timing-related safety properties typically found in embedded software. The empirical results indicate that our approach successfully supports our hypothesis, which explores the potential of PBE-based selective abstraction in achieving a well-balanced combination of falsification precision and efficiency.

Our approach has a couple of drawbacks. Firstly, our FS selection strategy is exponential, as it involves selecting the best subset by scoring each subset of FSs (Section 3.3.2). Though this issue was not prominent as our target programs used in experiments contain only 1 to 8 auxiliary functions, it could be a major performance bottleneck for more complex systems. This may be addressed by introducing heuristics for limiting the maximum number of selected FSs. Secondly, our approach has to limit its applicability to auxiliary functions that do not contain float-typed or dynamic input/output variables due to the limitation of state-of-the-art PBE techniques. We may apply more aggressive data abstractions to broaden the applicability of PBEAR; For example, dynamic objects can be abstracted to static objects and flattened out before applying PBE. Float-typed auxiliary functions can be abstracted using well-known abstraction techniques [8, 14]. A simple example is abstracting a statement `if(x > y)` into `if(B)`, where `B` is a Boolean abstraction of `(x > y)`, `x` and `y` are float-typed input variables, but the output variable should be integer-typed.

The performance of PBEAR depends on how we use the PBE tool, which is quite sensitive to the choice of I/O constraints. While PBEAR aims to optimize the performance of the PBE tool by balancing the number of I/O constraints during iterative synthesis, it may not always produce the best solution. In Table 5, PBEAR$^{-\text{cov}}$ was able to falsify ψ_{24} because it does not attempt to control the number of I/O constraints, enabling higher accuracy. In contrast, PBEAR failed to achieve the same result. However, increasing I/O constraints also raises complexity, as illustrated in the failure of PBEAR$^{\text{b}}$ in Table 4 for ψ_{16}. Thus, there is a room for improvements by identifying more effective I/O constraint selection strategies.

5.8 Threats to Validity

Constructing FSs using PBE produces neither sound nor complete abstractions. Therefore, there is no guarantee that our approach will succeed in falsification even when faults exist in a target program. Additionally, the metrics proposed for FS, including accuracy, output coverage, and complexity, do not always ensure falsification performance. Depending on the searching heuristics, a model checker may find violations more quickly, influenced by luck, even if the complexity of FS is higher than that of the original function. Similarly, accuracy only indicates a possibility of returning more precise output values.

There may be a question regarding the generality of our approach as the experiments were performed on a limited number of benchmarks. We do not claim that PBEAR works better than existing approaches in general. However, as it is specifically aimed, the experimental results show the potential benefits of our approach for embedded software, where multiple auxiliary functions exist with complex temporal properties. Turning to internal validity, the possibility of errors within our implementation exists. To address this threat, we manually checked each step of the PBEAR process. It is also possible that our experimental setup was not optimal, e.g., we might not have used the best options for other tools. We tried to find the best options by communicating with tool developers.

6 RELATED WORK

6.1 Abstraction using Function Summaries

FSs have been used in various contexts, e.g., in static analysis [6], model checking [2, 52], and concolic testing [33, 41]. For BMC, FunFrog [52] and HiFrog [2] construct over-approximate FSs for target functions in a SSA form and iteratively refine them using counterexamples to reduce false alarms. SMART [33] uses FSs in the context of efficient dynamic test generation. It constructs an under-approximate FS by collecting path formulas in pre- and post-conditions after each dynamic execution for a function. FOCAL [41] enhances the approach by introducing counterexample-guided refinements in the context of extended unit testing. Our approach is in the same line of these approaches but uses PBE for constructing function summaries and with details on how to select the target functions to be abstracted.

BESTER [47] migrated from *all-or-nothing* paradigm, whether PBE either is successful (i.e., satisfying *all* given I/O constraints) or failed, to *best-effort* paradigm, which presents a ranked list of candidate programs that satisfy some *part* of I/O constraints for the better user interaction. BESTER is similar to PBEAR w.r.t considering the number of satisfied I/O examples (i.e., accuracy) and program size (i.e., complexity) for ranking the best result. However, PBEAR additionally considers output coverage for effective property falsification.

6.2 Selective Abstraction

The notion of selective abstraction has been used in various approaches; Bjesse [18] performed selective abstraction of memories for word-level model checking. Armando et. al [7] abstracts only for arrays in C programs. Yin et al. [56] proposed BMC for concurrent C programs by abstracting only scheduling constraints. Oil-CEGAR [38] is a CEGAR-based approach for embedded software with an OS model program to reduce the non-determinism of multitasking. It performs selective predicate abstraction, abstracting only those statements that use or define global variables. These approaches uniformly apply selective abstractions to the target program, while PBEAR explicitly selects functions to be abstracted using evaluation criteria.

The approach by Sato et al. [51] is the closest to ours as it performs predicate abstraction to a set of functions that have cyclic function calls in the functional language ML. In their experiments, out of 18 programs, the traditional predicate abstraction required 59 refinements, whereas selective predicate abstraction required just 28 refinements. PBEAR also selectively abstracts functions, but with different criteria and using PBE.

6.3 Compositional Model Checking

Compositional model checking [26] reduces verification complexity by decomposing a target system into several components and verifying each component separately, instead of verifying the entire system. Assume-guarantee reasoning [1, 32, 46] enables it more efficiently by generating assumptions about each component's environment. Our approach can be applied in this context by synthesizing *auxiliary* components using PBE instead of generating assumptions.

Conditional model checking [15], a variant of compositional model checking, reasons about a condition ψ such that a program satisfies a property under ψ. This enables partial verification, as some parts of the program that have already been verified under condition ψ are not explored again in the next model checking process. This approach has been further developed with verification strategy selection [12], cooperation of different verification tools [16], and decomposition of verification tasks [13]. These approaches are useful for finding more efficient strategies *during* verification process, while our approach focuses more on engineering aspects for efficient falsification using bounded model checking through PBE-based selective abstraction and refinement.

7 CONCLUSION AND FUTURE WORK

We have presented PBEAR, a novel property falsification technique using PBE-based function summaries for selective abstraction and refinement. PBEAR abstracts auxiliary functions so that they can be replaced with FSs, while leaving the main control logic so that we can easily localize alarm filtering and refinement to FSs. It synthesizes FSs using the PBE solver, filters true/false alarms symbolically, and refines PBE-based FSs with SAT solving for efficient property falsification. Our experiments showed that PBEAR outperformed the bounded model checking (CBMC), SAT-based abstraction (CBMC-refine), and predicate abstraction (CPAchecker) techniques in property falsification on three embedded software programs written in C. To our best knowledge, it is the first work utilizing PBE for abstraction and refinement.

Though our experiments were conducted only on the source code level, our approach can be generalized for falsifying embedded software systems with distributed architectures or with multiple external black-box components, such as third parties or platform-dependent hardware, which can be automatically abstracted using PBE as long as we can collect the I/O examples for those components. Believing that it is far more realistic to abstract using PBE than to assume the existence of specifications for external components, we plan to further investigate the applicability of PBEAR to the composition of embedded software components. We also believe that our approach is not only applicable to embedded software but is also likely to be suitable for programs in other domains with control-oriented structures and auxiliary functions.

DATA AVAILABILITY

PBEAR (Docker image) is publicly available at [39] for reproduction. All experimental results are also publicly available at https://figshare.com/projects/PBEAR/188904.

ACKNOWLEDGMENTS

We thank the reviewers for their extensive comments and constructive suggestions. We are also grateful to Woosuk Lee, the developer of DUET, for his help in utilizing DUET. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2021R1A5A1021944).

REFERENCES

[1] Karam Abd Elkader, Orna Grumberg, Corina S Păsăreanu, and Sharon Shoham. 2018. Automated circular assume-guarantee reasoning. *Formal Aspects of Computing* 30, 5 (2018), 571–595. https://doi.org/10.1007/s00165-017-0436-0

[2] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti EJ Hyvärinen, and Natasha Sharygina. 2017. HiFrog: SMT-based function summarization for software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, Axel Legay and Tiziana Margaria (Eds.). Springer, Berlin, Heidelberg, 207–213. https://doi.org/10.1007/978-3-662-54580-5_12

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD '13)*. IEEE, Portland, OR, USA, 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[4] Rajeev Alur, Dana Fisman, Saswat Padhi, Andrew Reynolds, Rishabh Singh, and Abhishek Udupa. 2019. The 6th Syntax-Guided Synthesis Competition (SyGuS-Comp). Retrieved September, 2023 from https://sygus-org.github.io/comp/2019/

[5] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, Axel Legay and Tiziana Margaria (Eds.). Springer, Berlin, Heidelberg, 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

[6] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Copenhagen.

[7] Alessandro Armando, Massimo Benerecetti, and Jacopo Mantovani. 2014. Counterexample-guided abstraction refinement for linear programs with arrays. *Automated Software Engineering* 21 (2014), 225–285. https://doi.org/10.1007/s10515-013-0132-0

[8] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 203–213. https://doi.org/10.1145/378795.378846

[9] Thomas Ball and Sriram K Rajamani. 2002. *Generating abstract explanations of spurious counterexamples in C programs.* Technical Report. Technical Report MSR-TR-2002-09, Microsoft Research.

[10] Cinzia Bernardeschi, Andrea Domenici, and Paolo Masci. 2018. A PVS-Simulink Integrated Environment for Model-Based Analysis of Cyber-Physical Systems. *IEEE Transactions on Software Engineering* 44, 6 (2018), 512–533. https://doi.org/10.1109/TSE.2017.2694423

[11] Dirk Beyer. 2022. Progress on Software Verification: SV-COMP 2022. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)*, Dana Fisman and Grigore Rosu (Eds.). Springer, Cham, 375–402. https://doi.org/10.1007/978-3-030-99527-0_20

[12] Dirk Beyer and Matthias Dangl. 2018. Strategy Selection for Software Verification Based on Boolean Features. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, Cham, 144–159. https://doi.org/10.1007/978-3-030-03421-4_11

[13] Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. 2022. Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, USA, 536–548. https://doi.org/10.1145/3510003.3510064

[14] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer* 9 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[15] Dirk Beyer, Thomas A Henzinger, M Erkan Keremoglu, and Philipp Wendler. 2012. Conditional model checking: A technique to pass information between verifiers. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 57, 11 pages. https://doi.org/10.1145/2393596.2393664

[16] Dirk Beyer and Sudeep Kanav. 2022. CoVeriTeam: On-Demand Composition of Cooperative Verification Systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)*, Dana Fisman and Grigore Rosu (Eds.). Springer, Cham, 561–579. https://doi.org/10.1007/978-3-030-99524-9_31

[17] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *International Conference on Computer Aided Verification (CAV '11)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, Berlin, Heidelberg, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[18] Per Bjesse. 2008. Word-Level Sequential Memory Abstraction for Model Checking. In *Formal Methods in Computer-Aided Design (FMCAD '08)*. IEEE, Portland, OR, USA, 1–9. https://doi.org/10.1109/FMCAD.2008.ECP.20

[19] David C Black and Jack Donovan. 2004. *SystemC: From the ground up.* Springer, New York, NY, USA.

[20] Angelo Brillout, Daniel Kroening, and Thomas Wahl. 2009. Mixed abstractions for floating-point arithmetic. In *Formal Methods in Computer-Aided Design (FMCAD '09)*. IEEE, Austin, TX, USA, 69–76. https://doi.org/10.1109/FMCAD.2009.5351141

[21] Brobot. 2016. Garbage Colletor Robot Program. Retrieved September, 2023 from https://github.com/stvhwrd/Brobot/

[22] David Bühler. 2017. *EVA, an evolved value analysis for Frama-C: structuring an abstract interpreter through value and state abstractions.* Ph. D. Dissertation. University of Rennes 1.

[23] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph Csallner. 2018. Automatically Finding Bugs in a Commercial Cyber-Physical System Development Tool Chain With SLforge. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 981–992. https://doi.org/10.1145/3180155.3180231

[24] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *International Conference on Computer Aided Verification (CAV '00)*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Springer, Berlin, Heidelberg, 154–169. https://doi.org/10.1007/10722167_15

[25] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, Berlin, Heidelberg, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[26] Edmund M Clarke, David E Long, and Kenneth L McMillan. 1989. Compositional model checking. In *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science*. IEEE, Pacific Grove, CA, USA, 19 pages. https://doi.org/10.1109/LICS.1989.39190

[27] Elevator controller. 2020. Elevator Controller Program. Retrieved September, 2023 from https://github.com/Sandbergo/elevator-controller/

[28] William Craig. 1957. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic* 22, 3 (1957), 269–285.

[29] Mattia Fazzini, Alessandra Gorla, and Alessandro Orso. 2020. A framework for automated test mocking of mobile apps. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. ACM, New York, NY, USA, 1204–1208. https://doi.org/10.1145/3324884.3418927

[30] Object follower. 2013. Object-Following Automotive Multitasking Program. Retrieved September, 2023 from https://github.com/addud/object-follower/

[31] Frama-C. 2023. Framework for Modular Analysis of C programs. Retrieved September, 2023 from https://frama-c.com/

[32] Mihaela Gheorghiu Bobaru, Corina S Păsăreanu, and Dimitra Giannakopoulou. 2008. Automated assume-guarantee reasoning by abstraction refinement. In *International Conference on Computer Aided Verification (CAV '08)*. Springer, Berlin, Heidelberg, 135–148. https://doi.org/10.1007/978-3-540-70545-1_14

[33] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '07, Vol. 42)*. ACM, New York, NY, USA, 47–54. https://doi.org/10.1145/1190215.1190226

[34] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[35] Daniel Conrad Halbert. 1984. *Programming by example.* Ph. D. Dissertation. University of California, Berkeley.

[36] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. ACM, New York, NY, USA, 1159–1174. https://doi.org/10.1145/3385412.3386027

[37] Yu Jiang, Han Liu, Houbing Song, Hui Kong, Rui Wang, Yong Guan, and Lui Sha. 2018. Safety-Assured Model-Driven Design of the Multifunction Vehicle Bus Controller. *IEEE Transactions on Intelligent Transportation Systems* 19, 10 (2018), 3320–3333. https://doi.org/10.1109/TITS.2017.2778077

[38] Dongwoo Kim and Yunja Choi. 2019. Model checking embedded control software using OS-in-the-loop CEGAR. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. IEEE, San Diego, California, 565–576. https://doi.org/10.1109/ASE.2019.00059

[39] Yoel Kim and Yunja Choi. 2023. Reproduction Package (Docker Image) for the FSE 2024 Article 'PBE-based Abstraction and Refinement for Efficient Property Falsification of Embedded Software'. figshare. https://doi.org/10.6084/m9.figshare.24798264.v4

[40] Yunho Kim, Yunja Choi, and Moonzoo Kim. 2018. Precise concolic unit testing of C programs using extended units and symbolic alarm filtering. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/3180155.3180253

[41] Yunho Kim, Shin Hong, and Moonzoo Kim. 2019. Target-driven compositional concolic testing with function summary refinement for effective bug detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. ACM, New York, NY, USA, 16–26. https://doi.org/10.1145/3338906.3338934

[42] Adrian Baruta Kyle Foss, Ivo Couckuyt and Corentin Mossoux. 2022. Automated Software Defect Detection and Identification in Vehicular Embedded Systems. *IEEE Transactions on Intelligent Transportation Systems* 23, 7 (2022), 6963–6973. https://doi.org/10.1109/TITS.2021.3065940

[43] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software*

Engineering (ESEC/FSE '17). ACM, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309
[44] Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434335
[45] Kenneth L. McMillan. 2006. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification (CAV '06)*, Thomas Ball and Robert B. Jones (Eds.). Springer, Berlin, Heidelberg, 123–136. https://doi.org/10.1007/11817963_14
[46] Corina S Păsăreanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M Cobleigh, and Howard Barringer. 2008. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design* 32 (2008), 175–205. https://doi.org/10.1007/s10703-008-0049-6
[47] Hila Peleg and Nadia Polikarpova. 2020. Perfect is the enemy of good: Best-effort program synthesis. In *34th European Conference on Object-Oriented Programming (ECOOP '20) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.2
[48] Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*. ACM, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310
[49] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. 2007. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems* 29, 5 (2007), 43 pages. https://doi.org/10.1145/1275497.1275502
[50] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. 2019. Get Rid of Inline Assembly through Verification-Oriented Lifting. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. IEEE, San Diego, CA, USA, 577–589. https://doi.org/10.1109/ASE.2019.00060
[51] Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. 2013. Towards a Scalable Software Model Checker for Higher-Order Programs. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM '13)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/2426890.2426900
[52] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2011. Interpolation-based function summaries in bounded model checking. In *7th International Haifa Verification Conference (HVC '11)*, Kerstin Eder, João Lourenço, and Onn Shehory (Eds.). Springer, Berlin, Heidelberg, 160–175. https://doi.org/10.1007/978-3-642-34188-5_15
[53] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7
[54] VPlusLab. 2023. CROWN 2.0. Retrieved September, 2023 from https://www.vpluslab.kr/crown2/
[55] Yu Wang, Fengjuan Gao, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. 2022. Automatic Detection, Validation, and Repair of Race Conditions in Interrupt-Driven Embedded Software. *IEEE Transactions on Software Engineering* 48, 1 (2022), 346–363. https://doi.org/10.1109/TSE.2020.2989171
[56] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2018. Scheduling constraint based abstraction refinement for weak memory models. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. ACM, New York, NY, USA, 645–655. https://doi.org/10.1145/3238147.3238223

Received 2023-09-28; accepted 2024-01-23