Explicating Symbolic Execution (XSYMEXE): An Evidence-Based Verification Framework

John Hatcliff, Robby, Patrice Chalin, Jason Belt

Department of Computing and Information Sciences, College of Engineering, Kansas State University, USA
{hatcliff,robby,chalin,belt}@k-state.edu

***Abstract*—Previous applications of symbolic execution (SYMEXE) have focused on bug-finding and test-case generation. However, SYMEXE has the potential to significantly improve usability and automation when applied to verification of software contracts in safety-critical systems. Due to the lack of support for processing software contracts and ad hoc approaches for introducing a variety of over/under-approximations and optimizations, most SYMEXE implementations cannot precisely characterize the verification status of contracts. Moreover, these tools do not provide explicit justifications for their conclusions, and thus they are not aligned with trends toward evidence-based verification and certification. We introduce the concept of *explicating symbolic execution* (XSYMEXE) that builds on a strong semantic foundation, supports full verification of rich software contracts, explicitly tracks where over/under-approximations are introduced or avoided, precisely characterizes the verification status of each contractual claim, and associates each claim with *explications* for its reported verification status. We report on case studies in the use of Bakar Kiasan, our open source XSYMEXE tool for SPARK ADA.**

I. INTRODUCTION

Over the last decade, software engineering and formal methods research has demonstrated that symbolic execution (SYMEXE) [1] can be an effective technique for automatically checking wide-ranging properties of a program's behavior with little or no developer intervention. Previous applications of SYMEXE have centered around detection of common faults (such as null-pointer dereferencing, buffer overflows, array bounds violations), and test case generation [2]–[6].

SYMEXE tools aim to provide highly-automated and precise reasoning about program states by employing a variety of approximating optimizations. Path sensitive analyses are guided by heuristics that prune, or cutoff, certain paths. For example, to reduce the burden on developers, in SYMEXE, loop invariants are optional; instead, loops are (dynamically & lazily) unrolled up to a developer-configurable bounded depth. Rather than requiring complex "shape properties" for data structures [7], [8] to be declared by developers, data structures are explored up to a bounded size. Finally, rather than requiring manual steps in a theorem prover, logical constraints over program variables are solved by decision procedures (DPs) that may return "don't know" or time-out after a developer-configured time bound. When these path cutoff optimizations occur they can cause SYMEXE to compute an *under-approximation* of the program's behavior; thus, SYMEXE may fail to discover faults in a program that occur beyond path cutoff points. On the other hand, programs often contain data computations that lie outside the theories handled by DPs and the typical strategy for dealing with such cases can cause SYMEXE to introduce an *over-approximation* of a portion of the program's behavior, thus, it may report "false alarms".

While these approximations are effective for automation, their impact on the coverage of a program's state space and the reporting of results of property checking cannot be easily predicted. At a more fundamental level, the unconstrained and unmonitored use of these optimizations means that developers cannot trust the tool to yield a definitive report of the correctness of assertions or other program properties.

In contrast to the emphasis on bug-finding for predefined properties and faults outlined above, we are interested in the *verification* of developer-supplied contracts that specify full functional correctness. While software contract checking is useful in many contexts, we are interested primarily in its application in the development of safety-critical software.

Unfortunately, our experience in several industrial collaborations with companies such as Rockwell Collins, who produce certified safety and security critical software, has shown that current tools for contract checking place too great a burden on developers. For example, the SPARK language and tool framework [9] is one of the premier commercial development frameworks for high-assurance software. SPARK has been used to develop a number of safety/security-critical systems. Even though SPARK and its static analysis components are beneficial and easy to use, its contract language is rarely used due to the burdens the associated tools and methodology impose on developers. In fact, we are not aware of any industrial development effort that makes significant use of the SPARK pre/post-condition notation other than to specify enough context information for procedures to enable absence of run-time errors to be proved.

We believe that a more foundational approach to SYMEXE can significantly improve the usability and effectiveness of contract checking tools by providing a completely automated lazy bounded verification technology that scales to very complex contracts. Our vision is that the highly automated nature of SYMEXE would allow developers to apply contract

This material is based upon work supported by the National Science Foundation under Grant # 0644288 and by the US Air Force Office of Scientific Research (AFOSR) under contract FA9550-09-1-0138.

checking early in the development process with very little effort. We believe that in many cases, SYMEXE can provide complete verification that code conforms to contracts.

In cases where complete verification is not achieved, bounded SYMEXE-based checking can lead to the detection and removal of the vast majority of functional flaws in code and contracts. Moreover, SYMEXE can provide a variety of forms of semantic visualizations, test generation, *etc.*, that can help developers better understand their code, enable more rapid development of contracts, and provide evidence that significantly increases confidence in program correctness.

Our aim is not to replace, *e.g.*, the SPARK Examiner—SPARK's *verification condition generation* (VCGen)-based tool—but to complement it by offering highly automated developer-friendly techniques that can be used directly in the code (specify) / test (check) / debug (understanding feedback) loop of typical developer workflows. Our vision includes providing tool reports that clearly distinguish between contracts that have been proven by SYMEXE from those with remaining proof obligations. Contracts with undischarged proof obligations would be handed off later in the development process to verification engineers who apply less automated tools and proof assistants.

To achieve this vision, we have developed a novel approach to SYMEXE for software contract checking. We call this approach *explicating symbolic execution* (XSYMEXE) because it is designed from the ground up to have robust semantic underpinnings and to produce explicit explanations and justifications about the verification status of each of developer-supplied claim in a program. We make the following contributions:

- Introduce the concept of XSYMEXE as the foundation of an evidence-based framework for justifying the verification status of software contracts checked by SYMEXE, and explain the basic algorithms of XSYMEXE used to explicitly track when under/over-approximations are introduced (Sections II-A, III & V).
- Provide rigorous mathematical definitions for the basic principles of XSYMEXE including notions of soundness that justify the developer interpretation of verification results of SYMEXE when applied to software contracts (Sections II-B, III & IV).
- Implement XSYMEXE in the open source Bakar Kiasan (Kiasan for short) symbolic execution tool for SPARK ADA.
- Summarize the results of publicly available case studies and related artifacts on applying Kiasan to a variety of SPARK examples (Section VI).

This work was motivated by collaborations with engineers at Rockwell Collins and knowledge of how SPARK is being used in Rockwell Collins' security critical projects. We are currently collaborating with AdaCore and Altran Praxis to integrate this technology into their next version of SPARK. Although we illustrate our techniques in the context of SPARK, we believe that the concepts can easily be adapted to other tools, working on different languages, such as the Symbolic Path Finder (SPF) [10] and Klee [6].

II. BACKGROUND

In this paper, we are concerned with XSYMEXE for contract-based languages like SPARK ADA. We will use the term *claim* to be a contract pre-, or post-condition, or inline assert statements and their specialized forms (such as loop invariants).

A. Evidence-based Verif.: Conventional SYMEXE Shortfalls

We carefully illustrate here the ways in which SYMEXE can lead to uncertainty in the verification status of claims while pointing out how XSYMEXE can offer clarity in the conclusions that can be drawn. Examples are in SPARK and the essence of the notation is explained as we go along.

Issue: Under-approximation due to bound (or selective search) cutoffs may mask program faults, one of the more obvious limitations of bounded SYMEXE, is illustrated in **Figure 1(a)**, where an array is used to implement a queue of Size elements. SPARK's contract notation is fairly conventional and should be easy to interpret once one knows that occurrences of E~ in a post-condition refers to pre-state values of E. Execution of this procedure will result in an implicit claim violation due to the array index being out of range in the expression A(K + 1) at Line 20. The Ada standard mandates several kinds of implicit claims like these. The source of the problem is the upper bound of the for loop: it should be Size - 1. When a conventional implementation of SYMEXE is invoked with a loop bound less than Size, then no contract violation is reported because the last iteration of the loop (in which K is equal to Size) is never reached due to the path cutoff from the loop bound exhaustion.

Solution: Because XSYMEXE precisely tracks: (1) when bound exhaustion leads to path cutoffs, and (2) the implicit and explicit claims whose checking may be bypassed due to those particular cutoffs, Kiasan reports that the implicit range claim at Line 20 is VERIFIED$^?$. In Kiasan, we consistently use "?" as an "inconclusiveness" qualifier, read as "maybe". Kiasan also provides an explication that enumerates all the program points in which SYMEXE bound exhaustion occurred which may prevent the full evaluation of the claim.

Issue: Cutoffs in assumption statements, a precondition in the case illustrated in **Figure 1(b)**, can lead to over-approximation and **false alarms**. The procedure contract is actually valid, but the over-approximation introduces a false alarm as we explain next. Consider SYMEXE of this example with a loop bound of 4 and an array bound of 5. SYMEXE begins at the precondition. Since the precondition's universal quantifier is (symbolically) executed as a loop, and since Size can take on the value 5, its evaluation will lead to a symbolic state in which A(K) /= EMPTY for K = 1..4, but will leave A(5) unconstrained since the last iteration of the (quantifier) loop is cutoff due to loop bound exhaustion.[1] SYMEXE continues with the evaluation of the procedure body using (simplifying slightly) a symbolic state built out of the

[1]Terminating the analysis at cutoff points in assumptions would cause the subsequent code/claims to be unanalyzed. In contrast, any claim verified under an over-approximating assumption, is satisfied in any concrete execution.

(a) VERIFIED$^?$ (read inconclusiveness qualifier "?" as "maybe"; *i.e.*, "maybe VERIFIED") due to under-approx. caused by bounding. Claim is actually invalid.

```
subtype Index is Integer range 1..5;
type Vector is array (Index) of Integer;
EMPTY : constant := 0;

procedure dequeue(A : in out Vector;
                  Size : in out Integer; R : out Integer)
  --# pre  (for all K in Index range Index'First..Size =>
  --#          (A(K) /= EMPTY)) and
  --#       Index'First <= Size and Size <= Index'Last;
  --# post (for all K in Index range Index'First..Size =>
  --#          (A(K) /= EMPTY)) and
  --#       R = A~(Index'First) and A(Size~) = EMPTY and
  --#       Size = Size~ - 1 and
  --#       (for all K in Index range Index'First..Size =>
  --#          (A(K) = A~(K + 1)));
is begin
  R := A(Index'First);
  for K in Index range Index'First..Size
    loop A(K) := A(K + 1); end loop;
  A(Size) := EMPTY; Size := Size - 1;
end dequeue;
```

(b) False alarm: REFUTED$^?$ (read "maybe REFUTED") claim, that is actually valid, due to bound exhaustion in assumption (precondition) causing over-approx.

```
procedure dequeue_fixed(...) -- contract same as dequeue
is begin
  R := A(Index'First);
  for K in Index range Index'First..Size - 1
    loop A(K) := A(K + 1); end loop;
-- rest of body like dequeue
```

(c) False alarm: INDETERMINATE claim, reported (due to unhandled theory over-approximation) but claim is actually valid.

```
procedure FAIT(I, J : Integer; R, K, Q : out Integer)
  --# pre I >= -10 and I <= -1 and J >= 1 and J <= 10;
  --# post Q >= -99 and Q <= 0;
is begin R := I * J; K := R + 1; Q := K; end FAIT;
```

(d) UNCOVERED claim at Line 39 if the loop bound is less than 5.

```
procedure B (K : Integer; R : out Integer) is begin
  R := 0;
  for J in Integer range 1..5 loop R := R + K; end loop;
  --# assert P1(K, R);
end B;
```

(e) DEAD claim and code since the product of 2 naturals cannot be negative.

```
procedure DeadEx(J, K : Natural; R : out Integer) is
begin
  R := J * K;
  if(R < 0) then
    --# assert P2(J, K, R);
    R := 0;
  end if;
end DeadEx;
```

(f) FAULTY postcondition as it yields an array index out of bounds violation.

```
procedure InsertionSort (A : in out Vector)
  --# post for all K in Index => (A(K) <= A(K + 1));
-- details of body elided for brevity.
```

(g) VERIFIED claim (fully verified). No bound exhaustion or unhanded theories.

```
procedure shift(n : Index; A : in out Vector)
  --# post for all K in Index range 1..n-1 =>
  --#          (A(K) = A~(K+1));
is begin
  for K in Index range Index'First + 1 .. n
    loop A(K - 1) := A(K); end loop;
end shift;
```

(h) REFUTED claim (post-condition).

```
procedure swap(A : in out Vector; J, K : in Index)
  --# post A(K) = A~(J) and A(J) = A~(K);
is T : Integer; begin
  T := A(K); A(J) := A(K); A(K) := T;
end swap;
```

Fig. 1. Examples used to contrast conventional SYMEXE and XSYMEXE (excerpt, *e.g.*, SPARK derive clauses are elided)

state at the cutoff point. Thus, on the fourth evaluation of Line 28, the unconstrained value of A(5) gets copied into A(4). Being unconstrained, this value can be EMPTY and hence, subsequent evaluation of the post-condition results in a claim violation because the first quantifier's body (namely, A(K) /= EMPTY) is FALSE when K is 4.

Solution: XSYMEXE precisely tracks when over-approximation is introduced, during the evaluation of an assume statement (such as a precondition), due to bound cutoffs, Kiasan qualifies that the postcondition is REFUTED$^?$. It justifies the "?" qualification with an explication consisting of: (1) the assumption expression context where over-approximation was introduced, and (2) a counter-example in support of the claim violation. Of course, due to over-approximation, the counter-example may not satisfy the (full) precondition.

Issue: False alarm from over-approximation due to unhandled theories is illustrated in **Figure 1(c)** where the procedure contract is correct, but (conventional) SYMEXE would report a violation of the post-condition (Line 33) as we explain next. While the precondition sufficiently constrains the values of I and J to ensure that Q is in the stated range in the post-condition, the multiplication I * J at Line 34 introduces non-linear arithmetic which, at best, is partially handled, and generally completely unhandled, by many DP. (While integer non-linear arithmetic is used in this example, there are other theories that are undecidable/unsupported by even the most advanced DPs—*e.g.*, floating point, unbounded strings.) A common SYMEXE strategy in such situations is to introduce uninterpreted functions (abstracting away sub-expressions over unhandled theories) along with basic axioms over them—*e.g.*, for multiplication: a non-negative number multiplied by a non-negative number yields a non-negative number. This introduces an over-approximation which, for this example, would lead a conventional SYMEXE tool to wrongly conclude that the post-condition can be violated.

Solution: XSYMEXE precisely tracks over-approximation for unhandled theories due to the use of uninterpreted functions. In addition, Kiasan can determine if a claim status depends on such uninterpreted functions (*e.g.*, via a form of data and control dependency analysis [11]) and if it does, Kiasan reports the status as INDETERMINATE, indicating that evidence cannot be generated to refute the claim and that, as opposed to the example in Figure 1(b), re-running XSYMEXE with higher bounds will do nothing to help. In situations where uninterpreted functions are introduced and a particular claim violation does not depend on them, then Kiasan can conclude that the claim is REFUTED.

Issue: Uncovered claims. Naive approaches to SYMEXE simply report when a claim is (possibly) violated. In these approaches, the absence of a claim violation could lead the developer to incorrectly conclude that a claim can never be violated. However, it is possible that the bounding strategy used by SYMEXE has simply cutoff some paths—creating an under-approximation that misses evaluating the claims.

For example, when SYMEXE is applied to the example of **Figure 1(d)** with a loop bound of 4, the last iteration of the loop will never be executed, the path to the claim at Line 39 will be cutoff, and nothing will be reported about this claim.

Solution: Kiasan reports this claim as UNCOVERED, and it produces explications that includes bound cutoffs that may be blocking the execution of the claim along with paths from the bound cutoff points to the uncovered assertion.

Issue: Distinguishing dead from uncovered code/claims. Some claims lie along paths for which no concrete execution exists. Even if code coverage is used, conventional SYMEXE algorithms cannot distinguish between claims and code that are uncovered due to cutoffs (as in Figure 1(d)) from claims or code that are semantically unreachable (as in **Figure 1(e)**).

Solution: Because XSYMEXE tracks where paths are cutoff due to bound exhaustion, Kiasan can identify claims and code that are semantically unreachable from claims and code that are merely uncovered. (While the example in Figure 1(e) uses non-linear arithmetic, because of the basic axiom added on non-negative multiplications mentioned previously, Kiasan can conclude that the claim and code are dead.) A claim or code is reported as DEAD if it is uncovered and there are no bound cutoffs along SYMEXE paths from which the code in question is reachable in the procedure control flow graph. Dead code detection is important for critical embedded systems. In fact, in some sectors such as avionics, dead code must be removed to comply with, *e.g.*, DO-178C. Many dead code detection approaches rely on data flow frameworks or syntactic detection; our approach is more powerful (potentially detecting more dead code and generates less number of false alarms) because it considers infeasible path conditions.

Issue: Undefinedness (exceptions) during claim evaluation. Because claims are built from programming language expressions, their evaluation can lead to run-time exceptions. Such claims are "faulty" because their evaluation in a particular state may not return a definite truth value. **Figure 1(f)** illustrates a faulty contract with a post-condition that will generate a range check violation at A(K+1). It is important for developers to understand when their claims are faulty because no useful verification conclusions can be drawn from them; eliminating such exceptions from claims should be one of the first tasks in a contract-based quality assurance methodology.

Solution: Because XSYMEXE: (1) decomposes each contract into primitive claims, and (2) tracks the verification status of both explicit and implicit claims, Kiasan reports that the contract of Figure 1(f) is FAULTY. To explain why the contract is FAULTY, it also provides an explication listing implicit claims (runtime checks) in the contract, and a counterexample for such claims that are REFUTED.

Issue: Is a verified claim really verified? Conventional SYMEXE applied to **Figure 1(g)** may report no violations and offer no clear indication of the actual claim status. Has the post-condition been exhaustively validated for all possible states?

Solution: In analyzing this example, Kiasan relies only on the manipulation of logic constraints that are (completely) supported in the theories of the underlying DP and does not give rise to under-approximations due to bounding as long as the loop and array bounds are greater than Index'Last. If the latter is not true, then VERIFIED$^?$ is reported, hinting to the developer that an increase in the loop and array bounds might enable XSYMEXE to report that the contract has been unequivocally VERIFIED, which is the case for this example. Note that no loop invariant is needed to verify the example.

Issue: Is a refuted claim really refuted? As was alluded to earlier, a common pitfall of static analyses is that they produce too many false alarms that negatively impact one's ability to correct true claim violations. Many SYMEXE tools: (1) introduce over-approximations that can lead to false alarms, and (2) do not precisely track when such approximations have been introduced. Thus, they are unable to distinguish a potential false alarm from an actual claim violation.

Solution: Kiasan does not introduce any over-approximations when run on the example of **Figure 1(h)**. Because XSYMEXE precisely tracks when over-approximations are introduced, Kiasan is able to report that the claim represented by this contract can be REFUTED. Moreover, it yields an explication in the form of a concrete test case and counter-example: *e.g.*, a pre-state having J=2, K=1, A(J)=1, A(K)=0 that leads to violation of the postcondition—the bug is that the first assignment's right-hand side should be A(J).

B. SYMEXE*: Basic Formalization*

In this section, we present a basic formalization of SYMEXE, giving enough mathematical machinery to enable us to rigorously explain claim reporting and its associated semantics.

1) Procedures and Commands: Without loss of generality, we focus our technical definitions on procedural units and execution states for those units that include bindings of the procedure's local variables, parameters, and global variables that are either read or written by the procedure. We will assume (unless stated otherwise), that our formal definitions given below apply to the context of a given procedure P.

As is commonly done in works formalizing program correctness for imperative languages, we assume that our core command language includes the basic `assume` and `assert` statements which are used, in particular, to encode the pre- and post-conditions of procedure contracts. Execution of an assert statement whose expression evaluates to true, has no effect, otherwise, the procedure's execution is said to terminate in *error*. Similarly, execution of an assume statement whose expression evaluates to true, has no effect, otherwise, the procedure's execution is said to terminate due to *infeasibility* [12]. For simplicity, procedure contracts are embedded in the procedure's code as assume (*e.g.*, precondition) and assert (*e.g.*, postcondition) statements via program transformation [13].

2) Concrete and Symbolic Stores: A *store* is a finite partial function relating variables (**Var**) to their values. A *concrete store*, $\sigma^{\mathcal{C}} \in \mathbf{Var} \rightharpoonup \mathbf{Value}^{\mathcal{C}}$, associates variables to concrete

values $\mathbf{Value}^{\mathcal{C}}$. Since symbolic execution manipulates both $\mathbf{Value}^{\mathcal{C}}$ and symbolic values, we define a *symbolic store* $\sigma^{\mathcal{S}} \in \mathbf{Var} \rightharpoonup \mathbf{Value}^{\mathcal{S}}$, where $\mathbf{Value}^{\mathcal{C}} \subseteq \mathbf{Value}^{\mathcal{S}}$.

3) Concrete and Symbolic States: A *concrete state* $s^{\mathcal{C}} = (l\,,\,\sigma^{\mathcal{C}})$ is a pair consisting of a program point (also called a label) l and a concrete store $\sigma^{\mathcal{C}}$, where intuitively, $\sigma^{\mathcal{C}}$ represents the value of the program variables immediately *before* the command at l is executed. We say that $s^{\mathcal{C}}$ is a *state for* l. Similarly, a *symbolic state* $s^{\mathcal{S}} = (l\,,\,\sigma^{\mathcal{S}}\,,\,\phi\,,\,g)$ is a tuple consisting of: a program point, a symbolic store, a *path condition* ϕ, consisting of a finite set of formulae that act as constraints on symbolic values in the store, and a status flag used to indicate, among other things, whether a state is "normal" or "potentially over-approximating" (explained further below). We also say that $s^{\mathcal{S}}$ is a *state for* l.

Let $\Sigma^{\mathcal{C}}_{\mathsf{P}}$ and $\Sigma^{\mathcal{S}}_{\mathsf{P}}$ denote the concrete and symbolic state sets of the procedure P, respectively. When referring to state sets generically, we shall omit the P qualifier on the name.

4) Concrete Execution: A *concrete path* $\pi^{\mathcal{C}} = s^{\mathcal{C}}_1, s^{\mathcal{C}}_2, \ldots$ for procedure P is a possibly infinite sequence of one or more concrete states representing the states generated by executing commands in P from the initial state $s^{\mathcal{C}}_1$. Similar to [12], we have each finite path $\pi^{\mathcal{C}}$ for P end in a final state $s^{\mathcal{C}}_n$ having a special label $l_n \in \{\textsc{Normal}, \textsc{Error}, \textsc{Infeasible}\}$ denoting the path termination status. An INFEASIBLE state results from an assume statement's expression evaluating to $false$. Assume statements are used, *e.g.*, to encode preconditions. We let $\Pi^{\mathcal{C}}_{\mathsf{P}}$ denote the set of all possible concrete paths in P. The program point l in P is said to be *reachable* iff there exists a path in $\Pi^{\mathcal{C}}_{\mathsf{P}}$ containing a state for l; otherwise it is said to be *unreachable* or *dead*.

5) Symbolic Execution: A *symbolic path* $\pi^{\mathcal{S}} = s^{\mathcal{S}}_1, \ldots, s^{\mathcal{S}}_n \in \Pi^{\mathcal{S}}_{\mathsf{P}}$ for procedure P is a sequence of one or more symbolic states representing the states generated by symbolically executing commands in P from the initial state $s^{\mathcal{S}}_1$. Unlike concrete paths, symbolic paths are not required to end in a state for one of the special terminal labels. This models the fact that a symbolic path may be partial due to a SYMEXE *bounds cutoffs*, *e.g.*, array, loop, or call-chain bounds. For simplicity, we often refer to user-configurable analysis bounds (*e.g.*, loop bound, bounding on arrays, timeout) simply as "bounding". We call a symbolic path *complete* iff it ends in a state for one of the special terminal labels, and it is termed *incomplete* otherwise.

6) Soundness: Since we use SYMEXE for formal verification as opposed to just bug finding, we define a binary simulation relation ($\triangleleft$) [14] relating concrete and symbolic states such that $s^{\mathcal{C}} \triangleleft s^{\mathcal{S}}$ when $s^{\mathcal{S}}$ *over-approximates* (*abstracts* or *simulates*) $s^{\mathcal{C}}$. A symbolic state and the concrete states that it abstracts, always agree on their program point. A concrete path $\pi^{\mathcal{C}} = s^{\mathcal{C}}_1, s^{\mathcal{C}}_2, \ldots$ is *over-approximated* by a symbolic path $\pi^{\mathcal{S}} = s^{\mathcal{S}}_1, \ldots, s^{\mathcal{S}}_n$, on its first $n \geq 1$ states, denoted $\pi^{\mathcal{C}} \overset{n}{\triangleleft} \pi^{\mathcal{S}}$, iff $\pi^{\mathcal{C}}$ is at least of length n and $s^{\mathcal{C}}_i \triangleleft s^{\mathcal{S}}_i$ for all $1 \leq i \leq n$. When $\pi^{\mathcal{C}}$ and $\pi^{\mathcal{S}}$ are both of length n and $\pi^{\mathcal{C}} \overset{n}{\triangleleft} \pi^{\mathcal{S}}$, then we simply write $\pi^{\mathcal{C}} \triangleleft \pi^{\mathcal{S}}$. SYMEXE is *sound* iff Figure 2 commutes, where a step of concrete/symbolic execution is denoted by $\mathcal{E}^{\mathcal{C}}$, $\mathcal{E}^{\mathcal{S}}$, respectively.

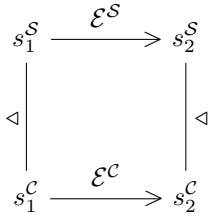

Fig. 2. Commutativity of Concrete Execution and SYMEXE

III. CLAIMS

We address the checking of *developer claims* written in a *formal specification language* that state desired properties of programs written in a particular *programming language*. Several approaches have been proposed for interpreting claims. In the *mathematical interpretation* approach taken in languages such as the current version of SPARK [9], claims are viewed as pure logical formula whose evaluation completes in a single step yielding either *true* or *false* as a result.

In this work, we adopt the *executable interpretation* approach [15] to align with the semantics of specifications to be used in the next generation of SPARK based on Ada 2012. In the executable approach, claims are decomposed to atomic boolean expressions in the programming language. Interpreting a claim amounts to executing the boolean expressions to which it decomposes. Thus, claim evaluation does not in general proceed in a single atomic step; it may have the possibility of terminating abnormally due to ill-formed subexpressions (array accesses with out-of-bound indices, divide-by-zero errors, etc.), *a.k.a.*, undefinedness [15], as illustrated in Figure 1(f). In the subsequent subsections, we give a top-down description of concrete and symbolic evaluation of claims—beginning first with the developer's view of claim evaluation and then drilling down to the details of claim decomposition.

A. Concrete Evaluation of Claims

Because SYMEXE is a path-sensitive analysis, we first address the evaluation of a claim at a particular state along a path and then derive the semantics of a claim by summarizing across all paths. Whenever the command at a program point l is a claim in the form of an assert or assume statement, we use C^l to denote the claim at l. We define $\textsc{CheckClaim}^{\mathcal{C}}(s^{\mathcal{C}}, \mathrm{C}^l) : \textsc{Bool}_{\perp}$ to be a function, returning either TRUE, FALSE or UNDEFINED, that represents the concrete evaluation of C^l in concrete state $s^{\mathcal{C}}$ whose program point is l. (Since $s^{\mathcal{C}}$ is a state for l, providing C^l as a second argument is unnecessary, but including it provides uniformity in the arguments of the semantic functions defined later.) To build towards the developer's intuitive understanding of the meaning of C^l across a set of executions $\Pi^{\mathcal{C}}$, we define the *concrete collecting summary* of C^l, denoted $\textsc{CollectClaim}^{\mathcal{C}}(\Pi^{\mathcal{C}}, \mathrm{C}^l) : \mathcal{P}\,(\textsc{Bool}_{\perp})$, as the union of all possible concrete execution results of C^l across all paths in $\Pi^{\mathcal{C}}$.

The *concrete interpretation* of a claim is the conclusion that a developer can draw about C^l based on its concrete behavior across *all* executions in $\Pi^{\mathcal{C}}$; $\textsc{InterpClaim}^{\mathcal{C}}(\Pi^{\mathcal{C}}, \mathrm{C}^l)$ is defined as follows:

(D) DEAD iff $\textsc{CollectClaim}^{\mathcal{C}}(\Pi^{\mathcal{C}}, \mathrm{C}^l) = \emptyset$, *i.e.*, C^l never appears in the paths $\Pi^{\mathcal{C}}$ (it is unreachable).

(V) VERIFIED iff $\text{COLLECTCLAIM}^{\mathcal{C}}(\Pi^{\mathcal{C}}, \text{C}^l) = \{\text{TRUE}\}$, *i.e.*, C^l is reachable and is TRUE in all states in which it is encountered.
(R) REFUTED when $\text{FALSE} \in \text{COLLECTCLAIM}^{\mathcal{C}}(\Pi^{\mathcal{C}}, \text{C}^l)$, *i.e.*, C^l is reachable and is FALSE on at least one path.
(F) otherwise, when $\text{UNDEFINED} \in \text{COLLECTCLAIM}^{\mathcal{C}}(\Pi^{\mathcal{C}}, \text{C}^l)$, *i.e.*, C^l is reachable and is UNDEFINED on at least one path.

In those situations above where more than one "when" case holds true then the first case is chosen.

B. Symbolic Evaluation of Claims

1) Symbolic Claim Check: Let $\text{CHECKCLAIM}^{\mathcal{S}}(s^{\mathcal{S}}, \text{C}^l) : \text{BOOL}^{?}_{\perp}$ be the result of symbolically evaluating the claim C^l in the symbolic state $s^{\mathcal{S}}$ (for C^l), where $\text{BOOL}^{?}_{\perp} = \text{BOOL}_{\perp} \cup \{\text{DONTKNOW}, \text{FALSE}^{?}, \text{UNDEFINED}^{?}\}$. While this function is generally implemented by a call to a decision procedure (DP), preprocessing and optimization steps may be applied before and/or instead of calling the DP. Details will be given in Section IV, and as will be explained there, DONTKNOW represents the situation where the decision procedure itself returns "don't know" or when a particular theory used in the primitive assertion is not supported by the decision procedure. $\text{FALSE}^{?}$ and $\text{UNDEFINED}^{?}$, read as "maybe false" and "maybe undefined", will be explained in Section III-B3 after the introduction of some essential terminology.

2) SYMEXE *Bound Exhaustion and Cutoff Paths:* As explained in Section II, in bounded SYMEXE, some symbolic paths are terminated prematurely due to bound exhaustion. We refer to these as *cutoff paths*, which are instances of *incomplete* paths mentioned in Section II-B5. We say that C^l is *impacted* by the cutoff path $\pi^{\mathcal{S}}$ of length n terminating in a state $s^{\mathcal{S}}_n$ iff there exists a concrete path $\pi^{\mathcal{C}}$ containing a state $s^{\mathcal{C}}_i$ for C^l, where $i > n$, for which $\pi^{\mathcal{C}} \overset{n}{\lhd} \pi^{\mathcal{S}}$. Intuitively, a claim that is impacted by a cutoff causes inconclusiveness in the analysis results due to the cutoff, because there are behaviors that are not analyzed that might affect the claim's status.

The detection of the potential impact of a claim C^l by a cutoff can be seen as a reachability problem since a cutoff at a program point l_n could prevent the flow of control from continuing to C^l. We define a conservative approximation of cutoff impacts using a Control Flow Graph (CFG) as follows. Let $\pi^{\mathcal{S}}$ be a cutoff path, then we say that C^l is *potentially impacted* by $\pi^{\mathcal{S}}$, and write $\text{C}^l \in \text{CUTOFFIMPACTS}(\pi^{\mathcal{S}})$, iff C^l can be reached from l_n (the program point of the final state in $\pi^{\mathcal{S}}$) in the CFG. (A more precise CUTOFFIMPACTS can be defined using control and data dependencies instead of CFG reachability; we opt to use CFG here for simplicity.)

3) SYMEXE *Bound Exhaustion in Assume Contexts:* A SYMEXE bound exhaustion results in a cutoff in every execution context except that of an assume statement, in which case the assume expression evaluation is stopped (and the symbolic constraints accumulated up until that point are preserved); in such cases, the Boolean sub-expression is assigned a non-deterministic value and SYMEXE continues to the sibling Boolean sub-expression of the assume statement, if any (otherwise, SYMEXE continues to the next statement). Because a non-deterministic value is introduced, the symbolic state status flag g described in Section II-B is set to indicate that the state (and all its successors) are potentially over-approximating. This is how the over-approximation discussed for the example of Figure 1(b) is tracked. In general, over-approximation may result in false alarms, but the alternative is to halt the exploration of the path – which would typically give less feedback to developers about the remaining procedure code and contract clauses. We can now clarify that the previously mentioned $\text{CHECKCLAIM}^{\mathcal{S}}(s^{\mathcal{S}}, \text{C}^l)$ outputs $\text{FALSE}^{?}$ and $\text{UNDEFINED}^{?}$ represent the situations where the DP yields FALSE or UNDEFINED (respectively), but that $s^{\mathcal{S}}$ is flagged as potentially over-approximating (due to a bound exhaustion taking place during the symbolic evaluation of an assume statement at some point earlier in the path).

4) Collecting Summary: Analogous to the concrete case, we define the *symbolic collecting summary* of C^l, denoted $\text{COLLECTCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l) : \mathcal{P}\left(\text{BOOL}^{?}_{\perp} \cup \{\text{CUTOFF}\}\right)$, as the union of: 1) all symbolic execution results of C^l across all paths in $\Pi^{\mathcal{S}}$ as reported by $\text{CHECKCLAIM}^{\mathcal{S}}$, and 2) $\{\text{CUTOFF}\}$, if C^l can be potentially impacted by a cutoff path in $\Pi^{\mathcal{S}}$.

5) Symbolic Interpretation: We now define the *symbolic interpretation* of C^l to represent the conclusions that a developer can draw about this claim from the results of SYMEXE. $\text{INTERPCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l)$ is defined as follows:

(D) DEAD iff $\text{COLLECTCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l) = \emptyset$.
(V) VERIFIED iff $\text{COLLECTCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l) = \{\text{TRUE}\}$.
(R) REFUTED when $\text{FALSE} \in \text{COLLECTCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l)$.
(F) FAULTY when $\text{UNDEFINED} \in \text{COLLECTCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l)$.
(R?) $\text{REFUTED}^{?}$ when $\text{FALSE}^{?} \in \text{COLLECTCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l)$.
(F?) $\text{FAULTY}^{?}$ when $\text{UNDEFINED}^{?} \in \text{COLLECTCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l)$.
(U) UNCOVERED iff $\text{COLLECTCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l) = \{\text{CUTOFF}\}$.
(V?) $\text{VERIFIED}^{?}$ iff $\text{COLLECTCLAIM}^{\mathcal{S}}(\Pi^{\mathcal{S}}, \text{C}^l) = \{\text{TRUE}, \text{CUTOFF}\}$.
(I) INDETERMINATE otherwise.

As before, when more than one "when" case occurs in the interpretation of the above, the first case is taken as defining.

IV. PRIMITIVE CLAIMS AND THEIR SEMANTICS

Thus far, we have explained the semantics of claims from a developer's perspective. In this section, we describe how the concrete and symbolic CHECKCLAIM methods are realized.

A. Primitive Claims and Classical DPs

Programmers have a natural intuitive understanding of the execution of a claim expression yielding either TRUE, FALSE or, in situations where execution cannot terminate normally (*e.g.*, due to an exception) UNDEFINED. This leads to logical formula over a 3-valued logic [15]. Almost all DPs operate over theories expressed in classical 2-valued logic. Hence, we adopt an approach that we developed earlier for use in the Java Modeling Language [15] and explained next.

Generally speaking, a developer claim C consists of n primitive claims A_i, for $1 \leq i \leq n$. We use the term defining primitive claim (A_d) to refer to A_n (the primitive claim that defines the logical meaning of the developer claim after all sub-expressions have been evaluated), and supporting primitive claims (A_s) (*e.g.*, array bounds checks and other range checks

required by Ada on sub-expressions in the developer claim) to mean an A_i, for $1 \leq i < n$. Due to the lack of space, we omit the full details of how a composite claim is decomposed into primitive claims. We do note here that supporting primitive claims are generated, among other reasons, to encode (2-valued) conditions whose truth will guarantee that evaluation of the "rest" of the claim will not result in UNDEFINED—*e.g.*, a claim expression involving division will have a supporting primitive claim asserting that the divisor is not zero.

B. CHECKPRIM$^{\mathcal{S}}$ and Decision Procedure

The checking of the developer claim via CHECKCLAIM$^{\mathcal{S}}$ is defined in terms of the more elementary CHECKPRIM$^{\mathcal{S}}$, which is applied to constituent primitive claims. CHECKPRIM$^{\mathcal{S}}$ is in turn defined via calls to a decision procedure. We now explain how CHECKPRIM$^{\mathcal{S}}(s^{\mathcal{S}}, \mathrm{A})$ can return TRUE, FALSE, DONTKNOW, or FALSE$^{?}$.

Consider the case where SYMEXE reaches a primitive claim A on a state $s^{\mathcal{S}}$. In CHECKPRIM$^{\mathcal{S}}(s^{\mathcal{S}}, \mathrm{A})$, we are interested in determining validity of A under $s^{\mathcal{S}}$. Typically, this is implemented as a call to a decision procedure such as a Satisfiability Modulo Theory (SMT) solver like, *e.g.*, Z3 [16] and Yices [17]. SMT solvers do not directly tell us if an assertion is valid since they are designed to check *satisfiability* of a formula instead of the formula's *validity*. We abstract the means by which we determine validity through satisfiability checkers behind an interface we call DP^{+}, which returns TRUE, FALSE, DONTKNOW.

We require DP^{+} to be sound; that is, it may at most introduce over-approximation (possibly generating false alarms), but not under-approximation (which could result in missed detection of "bugs"). Claim expressions over undecidable theories are the main reason for over-approximation, and unfortunately, most interesting programs deal with theories that are undecidable (*e.g.*, non-linear arithmetic and unbounded string theories). In such cases, a DP^{+} may not be able to give a definite answer; that is, it may give up producing an answer when it tries to reason about state constraints involving undecidable theories and hence return DONTKNOW.

We design our tool to work with multiple SMT solvers, but this is complicated by the fact that the solvers differ in their reasoning strengths and/or the theories that they (directly) support. For example, Yices has no support for non-linear arithmetic, while Z3 tries its best to conservatively solve non-linear arithmetic constraints; neither supports theories of unbounded strings or floating point numbers. The typical workaround for constraints involving theories not supported by the underlying solver is to weaken such constraints by using uninterpreted functions. While this approach is sound, it leads to over-approximation.

Thus, to distinguish conclusive results, we design CHECKPRIM$^{\mathcal{S}}(s^{\mathcal{S}}, \mathrm{A})$ to return TRUE only if the claim is valid (provably true), FALSE only if the claim is invalid (provably false). CHECKPRIM$^{\mathcal{S}}$ can achieve this by a post-processing step after calls to DP^{+} as follows. Let UT$(s^{\mathcal{S}}, \mathrm{A})$ returns $true$ iff $s^{\mathcal{S}}$ or A contain terms from unhandled theories. If DP$^{+}(s^{\mathcal{S}}, \mathrm{A}) =$ FALSE, then CHECKPRIM$^{\mathcal{S}}$: returns FALSE, if $\neg$UT$(s^{\mathcal{S}}, \mathrm{A})$; otherwise, DONTKNOW. A naive UT can be defined as: the path condition in $s^{\mathcal{S}}$ and the constraint in A involves weakening for unhandled theory. If DP^{+} answers TRUE or DONTKNOW, the answer is directly returned by CHECKPRIM$^{\mathcal{S}}$ regardless whether the constraint involves unhandled theory.

Note that this naive UT may sometimes be overly conservative, but can be improved. For example, when using Yices where non-linear arithmetic is unsupported, one can "emulate" multiplication by using an uninterpreted function, but with, *e.g.*, the following sign axiom: multiplication of non-negative numbers yields a non-negative number. In certain cases where such axiom applies, DP^{+} can give a definite answer. This can be detected by further queries to the underlying solver. Moreover, UT can be refined further by using a cone of influence (data dependence) analysis and control dependence analysis (See the technical report version of this paper for more discussion [18].)

In Section III-B3, we described that we may have an over-approximation while evaluating an assumption. In such case, if the DP^{+} returns FALSE, it maybe a false alarm introduced by the over-approximation. Thus, to distinguish this case with the case where we can conclusively determine that the assertion is provably invalid, CHECKPRIM$^{\mathcal{S}}$ examines the over-approximating flag g in $s^{\mathcal{S}}$; it returns FALSE$^{?}$ when g is set, which indicates there was an assumption over-approximation along the path. A test case can be generated to try to convert FALSE$^{?}$ to FALSE; that is, if the test case execution refines the symbolic path (a witness that refutes the claim), then CHECKPRIM$^{\mathcal{S}}$ should return FALSE.

Due to the lack of space, we omit the definitions of COLLECTPRIM$^{\mathcal{S}}(\Pi^{\mathcal{S}}, \mathrm{A}^{l})$ and INTERPPRIM$^{\mathcal{S}}(\Pi^{\mathcal{S}}, \mathrm{A}^{l})$ since they are quite similar to those for developer claims.

C. CHECKCLAIM$^{\mathcal{S}}$

Finally, we can explain how symbolic claim checking is defined in terms of its constituent primitive claims. Thus, for a claim A, CHECKCLAIM$^{\mathcal{S}}(s^{\mathcal{S}}, \mathrm{C}) : \mathrm{BOOL}^{?}_{\perp}$ (where $\mathrm{BOOL}^{?}_{\perp} = \mathrm{BOOL}_{\perp} \cup \{\text{DONTKNOW}, \text{FALSE}^{?}, \text{UNDEFINED}^{?}\}$) is defined in terms of the primitive claims of C and CHECKPRIM$^{\mathcal{S}}$ (whose return values are TRUE, FALSE, DONTKNOW and FALSE$^{?}$) as follows:

- UNDEFINED when CHECKPRIM$^{\mathcal{S}}(s^{\mathcal{S}}, \mathrm{A}_s) =$ FALSE, for any supporting assertion A_s.
- UNDEFINED$^{?}$ when CHECKPRIM$^{\mathcal{S}}(s^{\mathcal{S}}, \mathrm{A}_s) =$ FALSE$^{?}$, for any supporting assertion A_s.
- DONTKNOW when CHECKPRIM$^{\mathcal{S}}(s^{\mathcal{S}}, \mathrm{A}_s) =$ DONTKNOW, for any A_s.
- The value of CHECKPRIM$^{\mathcal{S}}(s^{\mathcal{S}}, \mathrm{A}_d)$ when CHECKPRIM$^{\mathcal{S}}(s^{\mathcal{S}}, \mathrm{A}_s) =$ TRUE for all A_s.

Again, when more than one "when" case occurs in the interpretation of the above, the first case is chosen.

V. CLAIM EXPLICATION

The main goals of XSYMEXE are to be more precise about the conclusiveness of contract checking results and to provide informative evidence for each of the results. XSYMEXE

explications are multi-tiered so that developers can see an initial explication for a developer claim status and then drill down through multiple levels of abstraction for more details—including results of claim checking on individual paths as well as results for the primitive claims that make up a developer claim. Explications for both developer and primitive claims are organized according to the set of status results obtained when collecting the results of claim evaluation across all paths. For example, the top-level report may indicate that a developer claim is REFUTED in a situation where the claim's defining primitive claim is FALSE along some paths and TRUE along other paths. In this case, explication drill-down yields two categories of explications (one for the TRUE paths and one for the FALSE paths). Drill-down through the TRUE category produces a collection of concrete and symbolic paths for when the primitive claim is TRUE (similarly for the FALSE case).

The table below describes the nature of the explications for each status category that can be returned by COLLECTPRIM$^{\mathcal{S}}$.

Collect category	Explication
FALSE	c/s-counter-examples, program location
TRUE	c/s-path
CUTOFF	partial c/s-path, cutoff-location, path chop
DONTKNOW	s-path, unhandled theory program locations
FALSE$^?$	c/s-counter-example, over-approx. location

FALSE: To illustrate an assertion refutation, it is sufficient to generate a counter-example demonstrating that the assertion does not hold under a certain circumstance. It is also helpful to highlight the assertion's source-level program location (region); this is especially valuable for identifying the source of undefinedness of a claim's supporting assertion.

TRUE: As an evidence of an assertion verification, it is useful to generate a (concrete/symbolic) c/s-path (or a test case) demonstrating that the assertion holds.

CUTOFF: To illustrate an assertion impacted by a cutoff, we generate a partial c/s-path demonstrating the program execution leading to the cutoff point; this is coupled with generating a possible path (chop) description/visualization illustrating how program control can transfer from the cutoff program location to the assertion.

DONTKNOW: We can output a symbolic path to illustrate an assertion whose validity cannot be determined; unfortunately, due to the unhandled theory issue described in Section IV-B, it may not always possible to generate a concrete path. Thus, we can only guarantee to output a symbolic path. Hence, it is helpful to also highlight program points that give rise to constraints with unhandled theory.

FALSE$^?$: Recall that FALSE$^?$ can only happen when there is an assumption over-approximation impacting an assertion that is refuted (under that over-approximating assumption). Thus, we can generate c/s-counter-examples illustrating the assertion refutation. As mentioned previously, the counter-examples may not fully satisfy the assumption. Thus, it is helpful to also highlight the program point where the assumption over-approximation occurred.

VI. EVALUATION

To evaluate our approach, we implemented XSYMEXE in Bakar Kiasan and applied it to an extended set of examples including the examples of Section II-A, standard sorting algorithms used for benchmarking, and representatives of data structures used to maintain data packet filtering and transformation in embedded security applications. The latter set is derived from a Rockwell Collins code base and uses arrays to provide a "linked list" set implementation (where links are represented as indices in an auxiliary array) with more efficient additions/deletions. These units are relatively small but generally have rich behavioral contracts since we focus on compositional verification of strong behavioral properties instead of (selective-search) whole program bug-finding. Note that many programs in safety/security-critical embedded applications are relatively small in size.

Bakar Kiasan summarizes the developer claim statuses on a per routine basis, as well as the individual statuses of primitive and defining claims in each developer claim. Status tokens, as described in Section III-B5 (such as `D`, `V`, `V?`, *etc.*), are provided along with claim status evidence like so:

Type	S	Col	Explication
INDEX_LOWER@68	V	T	Paths (0,1,2)
...	...	...	...
INDEX_UPPER@68	V	T	Paths (0,1,2)
POST check@68	R	FT	Failing Path (0@68), Paths (1,2)

Overall Status: R

The above is a condensed report excerpt for the swap procedure of Figure 1(h); the report summary style follows that of SPARK's POGS. The report shows the verification status of swap's postcondition (the Overall Status is R indicating that the claim is refuted), including its supporting primitive claims (*e.g.*, INDEX_LOWER, INDEX_UPPER, which are array range checks) and its defining primitive claim (*i.e.*, POST). For each primitive claim, the S column gives the verification status of the primitive claim given by INTERPPRIM$^{\mathcal{S}}$, the Col column gives the result of COLLECTPRIM$^{\mathcal{S}}$, and the Explication column gives links to the evidence that justifies the status.

As can be observed from the report, there are three distinct paths that soundly abstract *all* the concrete executions of swap's code/contract. The supporting claims are verified for all executions, but the defining claim is refuted by Failing Path 0, while the other two interestingly (perhaps unexpectedly) satisfy the post-condition.

To complement this high-level summary, Bakar Kiasan also generates detailed HTML reports that include source code highlighting (*e.g.*, syntax highlighting, highlighting of problematic areas as described in Section V), and code coverage. The HTML report illustrates each path as a test case with visualization of its pre/post-states. Space constraints prevent us from illustrating the HTML reports in sufficient detail; interested readers can find complete evaluation reports for all our examples online [18].

Figure 3 presents the collective summary (%) of the claim statuses for all examples mentioned at the start of this section. This represents data for 31 explicit claims (recall that a

(%)	D	V	R	F	R?	F?	U	V?	I
Explicit	3.2	58.1	12.9	3.2	3.2	0.0	3.2	12.9	3.2
Implicit	0.4	89.0	0.3	0.0	0.0	0.0	0.2	9.9	0.2
Total	0.5	88.0	0.7	0.1	0.1	0.0	0.3	10.0	0.3

Fig. 3. Claim Status Distribution

claim, such as a post-condition of Figure 1(a), can consist of a complex expression spanning several source lines and calls to helper functions), and 972 implicit claims. For each claim status, the table gives the percentage of claims that have that status. Our goal here is not to suggest that the given distribution is in any way representative of what might be obtained if a different and larger sample of code was analyzed. Rather, we believe that our main achievement is that xSYMEXE is *able* to generate such results. As stated in the introduction, our aim in the creation of xSYMEXE is not to suggest a replacement for, say, VCGen-based verification technology, but to offer a fully-automated complementary alternative that we have found most useful during the early stages of contract and code development in critical systems.

As for the results themselves, to our surprise, the tool pointed out some dead claims in our SPARK suite despite it being used and analyzed in previous studies. The number of VERIFIED claims is high, and the percentage for the other statuses are low. This is what we would expect because our SPARK code suite is fairly mature—the contracts have been repeatedly analyzed and (re-)worked. We anticipate that in practice, developers will more frequently encounter faulty and refuted claims as they write code/contracts, and they will want to eliminate those as early as possible during development (*e.g.*, analogous to experiences when coding in a statically-typed language). Once those are addressed, their verification effort can then focus on the inconclusive statuses (*i.e.*, UNCOVERED, INDETERMINATE and those qualified with "?"), which may require the use of higher-reward, higher-effort techniques (*e.g.*, VCGen tools or using proof assistants).

The proportion of claim statuses that are conclusive vs. those that are not is 4 for explicit claims and 8.5 overall. That is, on average, our tool is able to report 8.5 conclusive claim results for every 1 inconclusive one in our sample. While this is a good ratio for xSYMEXE, more studies are needed to better appreciate what the proportion may be on a larger and perhaps evolving code in production.

VII. RELATED WORK

Early work on the application of SYMEXE to the verification of safety-critical software is described by Coen-Porisini *et al.* [19]. They used SYMEXE to verify properties, expressed in a specialized Path Description Language (PDL), of functions written in Safe-C, a very restricted dialect of C. In their approach, a user must first use tooling to semi-automatically create a *finite* program Execution Model (EM) which is then used as input to the PDL Property Checker which in turn reports the list of EM paths for which given properties hold. Our work differs firstly in that Kiasan directly processes SPARK specifications expressed as contracts as opposed to separate properties written in a specialized language. SPARK contracts are formed from predicates built from standard Ada expressions. More importantly, Kiasan is fully automatic, and does not require, *e.g.*, user intervention in the creation of a finite model in the presence of while loops as is the case for the EM generator. Forcing users to create finite EMs side steps many of the problems we address in xSYMEXE.

Our work was partly inspired by the SPARK [9] Proof ObliGation Summariser (POGS) that: (a) summarizes the verification status of verification conditions as they are processed by different stages in the SPARK tool chain, and (b) uses SPARK's ZombieScope tool to indicate code regions that are semantically dead. Our aim was to see how this concept could be adapted from: (1) the logical interpretation of contracts in the current version of SPARK to the executable interpretation to be used in the upcoming version, and (2) VCGEN to SYMEXE so that developers could profit from the benefits of SYMEXE. Specifically, in contrast to SPARK's VCGEN approach, SYMEXE naturally generates counter-examples and test cases as evidence, enables a number of helpful visualizations, and does not require loop invariants to obtain an initial degree of contract checking. In addition, SPARK's VCGEN tends to yield an all-or-nothing approach when verifying contracts. Paths through a procedure are broken into segments and a VC is generated for each segment. Unless VCs for all segments are discharged (or one is observed to be false), nothing meaningful can be said about the verification status of the contract.

With SYMEXE, a developer's knowledge about the verification status of a contract is much more continuous: immediate feedback with a degree on inconclusiveness is provided with low bounds and conclusiveness is increased as bounds increase. POGS is less discerning. It only characterizes a VC as discharged or undischarged; it does not distinguish between an obligation that can be refuted from one whose status is yet to be determined. Finally, our approach detects both dead code and dead claims. This is beneficial for identifying, *e.g.*, portions of contracts that are not useful. ZombieScope can detect dead code (and thus, detecting dead claims that are inside the dead code regions); however, it does not detect dead claims outside of code such as pre- and post-conditions as it uses the logical contract interpretation.

Our discussion above applies, to some extent, to other VCGEN techniques such as ESC/Java tool family [20], [21], Boogie [22], and Why [23]. Similar to SPARK tools, ESC/Java2 has dead code detection; however, it does not detect dead claims. ESC/Java2, Boogie, OpenJML ESC, and Why do not generate *concrete* counter-examples to illustrate claim refutations. Some of these tools do provide counter-examples, but not in a form that is familiar to developers, *i.e.*, logical formulae instead of test cases (*e.g.*, [21]); thus, it does not scale well (in terms of clarity) to counter-examples involving complex constraints. Similar to the SPARK Examiner, ESC tools and Why do not distinguish provable claim refutations from failed verification attempts.

Some VCGEN-based tools such as Boogie can resort to a form of eager bounded analysis by loop unrolling (hence, loop

invariants are not required), but, in such cases, it diminishes the technique to merely bug-finding. In contrast, xSYMEXE uses dynamic lazy bounding, thus, xSYMEXE can achieve verification when it determines all behaviors are within the bounds. Moreover, we are not aware of their reporting capabilities (*e.g.*, evidence generation and result categorization), especially to the extent of the work presented here. We believe our work can be adapted to a bounded VCGEN-based approach as well.

Verifast [7] and jStar [8] are verification tools based on separation logic for checking claims about heap data. They use an algorithmic approach to SYMEXE that differs from that are used in SPF, Klee, Bakar Kiasan, *etc.*. Instead of relying on bounded checking, these tools require loop invariants and inductive predicates over data structures to create symbolic summaries of heap shapes. While these tools are beneficial, they target a different space, *i.e.*, focusing on full verification of heap properties and require significantly more annotations to be added by developers. For counter-example information, jStar can only generate program locations of interest when it cannot verify claims. In addition, it does not discern undischarged claims (similar to the SPARK Examiner and VCGEN tools above), and it does not address contract undefinedness.

We believe bug finding tools such as SPF [10], Klee [6], and PeX [4] are useful, especially when applied to code as it is being developed. However, because of many engineering compromises such as selective search (*e.g.*, heuristic-based symbolic state-space exploration), context bounding (*e.g.*, test harness), *etc.*, they do not provide guarantees when there is no bug detected. Hence, such approaches are not be able to verify claims or to precisely determine dead claims (or code).

Regardless, we believe our approach can be applied in such under-/over-approximating settings. For example, in the context of selective search, which causes some states to not be explored, the states can be treated as cutoff points. Context bounding is essentially an ad hoc form of assumption which may produce under-/over-approximation; it is typically done for testing certain code behaviors. As the code is geared for verification, it should be codified as a contract.

VIII. CONCLUSIONS AND FUTURE WORK

We have argued that bounded SYMEXE, as commonly implemented in the software engineering community, cannot be applied effectively for verification. Despite the fact that it: (a) offers various usability advantages, and (b) typically employs the same underlying decision procedure packages as verification condition generation (VCGen), the ad hoc approaches taken in SYMEXE for introducing optimizations and over/under-approximations have prevented tools from reporting the precise verification status of contract claims—causing SYMEXE to take a back seat to other deduction based techniques like VCGen in the context of verification.

In this work, we have presented a collection of principles that allow SYMEXE to be used confidently in development contexts that require verification as opposed to just bug-finding. Furthermore, we have presented an approach by which information accumulated during SYMEXE can organized into explications that provide evidence-based justifications for the resulting verification status of claims. Although we have demonstrated our approach in the context of the SPARK framework, the principles that we have introduced can be applied by other bounded SYMEXE tools as well.

The foundation presented here enables a number of interesting future directions. For example, making verification status of claims explicit enables a synergistic combination with other verification tools (*e.g.*, VCGen-based tools): programs are first submitted to highly-automated xSYMEXE based techniques, and then only undischarged claims (*i.e.*, any status but VERIFIED, REFUTED, DEAD) are handed off to downstream verification tools that require more manual effort.

REFERENCES

[1] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," *TACAS*, pp. 553–568, 2003.

[3] K. Sen and G. Agha, "CUTE: A concolic unit testing engine for C," in *ACM SIGSOFT FSE*, 2005, pp. 263–272.

[4] N. Tillmann and J. de Halleux, "Pex–white box test generation for .NET," in *TAP*, ser. LNCS, vol. 4966. Springer, 2008, pp. 134–153.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, 2008.

[6] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX OSDI*, 2008, pp. 209–224.

[7] B. Jacobs, J. Smans, and F. Piessens, "A quick tour of the VeriFast program verifier," in *Asian Symposium on Programming Languages and Systems (APLAS)*, ser. LNCS, vol. 6461. Springer, 2010, pp. 304–311.

[8] D. Naudziuniene, M. Botincan, D. Distefano, M. Dodds, R. Grigore, and M. J. Parkinson, "jStar-Eclipse: an IDE for automated verification of Java programs," in *SIGSOFT FSE*, 2011, pp. 428–431.

[9] J. Barnes, *High Integrity Software—the SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[10] C. S. Pasareanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," in *ASE*, 2010, pp. 179–180.

[11] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[12] M. Barnett and K. R. M. Leino, "Weakest-precondition of unstructured programs," in *PASTE*, 2005, pp. 82–87.

[13] J. Belt, J. Hatcliff, Robby, P. Chalin, D. Hardin, and X. Deng, "Bakar kiasan: Flexible contract checking for critical systems using symbolic execution," in *NASA Formal Methods*, 2011, pp. 58–72.

[14] D. A. Schmidt, "Structure-preserving binary relations for program abstraction," in *The Essence of Computation*, 2002, pp. 245–265.

[15] P. Chalin, "Engineering a sound assertion semantics for the verifying compiler," *IEEE Trans. Software Eng.*, vol. 36, no. 2, pp. 275–287, 2010.

[16] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[17] B. Dutertre and L. de Moura, "The Yices SMT solver," Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[18] J. Hatcliff, Robby, P. Chalin, and J. Belt, "Explicating symbolic execution (xSymExe): An evidence-based verification framework," Kansas State University, Tech. Rep. SAnToS-TR2012-08-01, 2012, santos.cis.ksu.edu/papers/ICSE13-xSymExe.

[19] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé, "Using symbolic execution for verifying safety-critical systems," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 142–151, Sept. 2001.

[20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *PLDI*, 2002, pp. 234–245.

[21] D. R. Cok, "Improved usability and performance of smt solvers for debugging specifications," *STTT*, vol. 12, no. 6, pp. 467–481, 2010.

[22] K. Leino, "This is Boogie 2," *Manuscript KRML*, vol. 178, 2008.

[23] J. C. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus platform for deductive program verification," in *CAV*. Springer, 2007, pp. 173–177.