Model Checker Execution Reports

Rodrigo Castaño Victor Braberman Diego Garbervetsky Sebastian Uchitel
Departamento de Computación, FCEyN, UBA, CONICET, Buenos Aires, Argentina
{rcastano,diegog,vbraber,suchitel}@dc.uba.ar

***Abstract*—Software model checking constitutes an undecidable problem and, as such, even an ideal tool will in some cases fail to give a conclusive answer. In practice, software model checkers fail often and usually do not provide any information on what was effectively checked. The purpose of this work is to provide a conceptual framing to extend software model checkers in a way that allows users to access information about incomplete checks. We characterize the information that model checkers themselves can provide, in terms of analyzed traces, i.e. sequences of statements, and *safe cones*, and present the notion of execution reports (ERs), which we also formalize. We instantiate these concepts for a family of techniques based on Abstract Reachability Trees and implement the approach using the software model checker CPAchecker. We evaluate our approach empirically and provide examples to illustrate the ERs produced and the information that can be extracted.**

I. Introduction

Software model checking [18] constitutes an undecidable problem and, as such, even an ideal tool will in some cases fail to give a conclusive answer. But the limitations are also practical. The vast state spaces can lead to the exhaustion of system resources or impractically long execution times.

Software model checking has been making steady progress during the past decade and today's state-of-the-art software model checkers can handle specific industrial problems particularly well. For instance SDV [1] is highly successful in finding bugs in Windows device drivers.

Unfortunately, some instances take hours of computation, only to inform the user that no counterexample was found within the allotted time or memory limit. A user facing this situation is confronted with several high-level questions about what the verification attempt actually achieved. Should she retry with a longer time limit? How much longer? Is the tool making progress? Maybe she should try another technique?

Our goal is to extend and complement existing work on partial verification by providing a different way for users to observe the work performed by the software model checker. An important step towards our goal is to be able to answer much simpler inquiries about incomplete verification attempts.

We believe answering the following questions would be valuable for a user after an inconclusive verification attempt:

- Can partial safety assurances about the system be extracted from an incomplete verification attempt? For instance, a user that receives a report showing that a whole class of relevant behaviors has been exhaustively checked may use this as part of a dependability case.
- Can behavior that was not analyzed be explored by a user? For instance, a user that can observe that relevant classes of behavior have not even been looked at by the checker, let alone verified, may decide that what seemed like a sufficiently thorough analysis is not such (e.g., an inexperienced user would benefit from knowing that a tool based on BMC with a fixed bound can sometimes give up without ever exploring anything beyond an initialization loop [19]). Moreover, a more experienced user attempting full verification may decide that a drastic change in the verification strategy is needed (e.g., another tool, abstracting the system-under-verification, etc.).

In this paper we explore answering the first question using the notion of *safe cone*. A *safe cone* is a finite trace for which any extension has been analyzed in the incomplete verification attempt. A *minimal feasible safe cone* represents compactly a set of traces that have been successfully verified by the checker. For the second question we build on the notion of a *frontier*. A *frontier trace* is one that was analyzed by the checker but none of its extensions were. Frontier traces represent compactly classes of traces that were not explored by the checker. We will see how ensuring traces are feasible plays a central role in interpreting these results.

Our hypothesis is that execution reports that under-approximate the set of minimal feasible *safe cones* and the set of maximal feasible *frontier traces* can be computed efficiently (with respect to the cost of verification) and can provide non-trivial feedback on incomplete verification attempts.

We will start by illustrating with examples the information we wish to extract and defining a possible formalization of the idealized properties it should have. Subsequently, we define and discuss execution reports (ERs), an under-approximation of the ideal output. Afterwards, we instantiate these concepts for the family of techniques based on Abstract Reachability Trees (ART) [16]. We also briefly discuss a proof-of-concept implementation built as an extension of CPAchecker [6] and an empirical evaluation of our implementation.

We conclude this paper with a discussion of related work and how our approach compares to existing techniques followed by a few concluding remarks.

II. Motivation: What Has and Has Not Been Analyzed?

We frame our work in the context of a verification attempt being interrupted before its completion.

Many verification techniques work by incrementally exploring the state space. This is the case for software model checking techniques like BMC [8], lazy predicate abstraction [16], inlining and unrolling based-techniques like Corral [20],

978-1-5386-2684-9/17 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research - New Ideas

```
1  int min(int a[], int n) {
2     int res = a[0];
3     for (int i=0;i<n;++i)
4       if (a[i] < res) res = a[i];
5     return res;
6  }
7  void init_vector(int a[], int n) {
8     int i = 0;
9     for (i=0;i<n;++i) {
10      a[i] = nondet();
11    }
12 }
13 void test_min(int large) {
14    int n;
15    if (large) {
16       n = 20;
17    } else {
18       n = 1;
19    }
20    int a[20];
21    init_vector(a, n);
22    int min_elem = min(a, n);
23    assert(min_elem <= a[0]);
24 }
```

Figure 1: Harness for method `min`

DSE [9] and, in general, ART-based implementations of various techniques, including explicit value analysis [7] and CEGAR variants of some of the previous among other.

In these techniques one can understand that incremental exploration leads to an incremental but silent increase of *analyzed* traces, i.e. sequences of statements, as depicted in the examples that follow. In fact, some traces might reach a fully verified portion of the behavior space, implicitly defining a *safe cone* containing all possible extensions of such traces.

Understanding that partial explorations provide safety assurances, we will discuss the *frontiers* defined by minimal *safe cone* traces and maximal *analyzed* statement traces.

We illustrate these concepts with a verification attempt of the instance in Figure 1 with bounded model checking.

Example 1 (Analyzed behaviors in BMC)**.** *The code snippet in Figure 1 corresponds to a parametric test harness exercising the method* `min`*. The harness input as well as the result of method* `nondet` *are interpreted by the verification technique as non-deterministic. Our verification attempt, in this example, is not interrupted due to reaching a resource limit but instead due to using a bounded model checker [8] with a bound on the number of loop iterations set to 3.*

Using this configuration, the tool would perform an exhaustive exploration but would disregard any executions involving the fourth loop iteration of the `for`*-loop in* `init_vector`*.*

Any sequence of statements reaching line 18 (`n = 1;`*) will necessarily satisfy the safety property, since the incomplete verification attempt would not find any assertion failures and the loop within* `init_vector` *can be exhaustively analyzed. That means the sequence of statements consisting of lines 14 (*`int n;`*), 15 (*`if (large)`*) and 18 (*`n = 1;`*) defines a* safe cone*, because every continuation of the statement trace is also* `safe`*. The former is also minimal in that if the last statement, line 18, were removed it would not be a* safe cone*.*

Moreover, any execution that carries out line 9 (`for (i=0; i<n; ++i)`*) at most 4 times (3 full iterations and 1 bound check) has also been analyzed. In contrast, any sequence containing line 9 at least 5 times is ignored by BMC and therefore will not be examined. Therefore, any trace exercising line 9 5 times or more will not be considered* analyzed*. That is, the trace composed of lines 14, 15, 16, 20, 21, 8 and then 4 repetitions of lines 9 (*`for (i=0;i<n;++i)`*) and 10 (*`a[i] = nondet();`*) is a maximal* analyzed *trace.* △

We will now revisit these concepts from an entirely different technique: lazy predicate abstraction [16].

Lazy predicate abstraction, the algorithm used by BLAST [16], consists of two alternated phases. The first phase generates, on-the-fly, a reachability tree whose nodes correspond to vertices of the control flow automaton[1] (CFA) of the program. This process goes on until exhaustively exploring the tree or until reaching a node that corresponds to an assertion failure. Each node is associated with a predicate, initially $true$, that must hold for any path reaching that node and prunes some unreachable successors. When a node that represents an error is reached, the second phase deals with analyzing the potential counterexample to determine whether the path reaching the error node is feasible. If the latter phase determines the potential counterexample is infeasible, the tree is refined by strengthening the predicates along the path so that the spurious error node is pruned. Lastly, when a counterexample is produced, it can be checked with a more precise analysis to ensure its feasibility. However, if this additional check fails, the search is abandoned.

Example 2 (Analyzed behaviors in lazy predicate abstraction)**.** *The code presented in Figure 2, reproduced from the papers presenting Conditional Model Checking [4], [5], contains a non-linear safety property. As explained, the construction of the reachability tree will reach the error node and the second phase would attempt to verify the feasibility of the path leading to it. The feasibility check is usually implemented by creating a verification condition to be checked by an underlying SMT solver and, therefore, inherits its limitations. In particular, SMT solvers usually cannot handle non-linear arithmetic and, instead, multiplication is modeled as an uninterpreted function. Concretely, this loss of precision prevents the SMT solver from proving infeasibility. The subsequent, more precise, check would prove the path infeasible and stop exploration.*

In this context, the following is a maximal analyzed *trace: lines 2 (*`int p = nondet();`*), 3 (*`if(p)`*), 8 (*`int x = 5;`*), 9 (*`int y = 6;`*) and 10 (*`int r = x * y;`*).*

On the other hand, the then *branch of the* `if`*-statement can be successfully verified with this technique, since tracking*

[1] A control flow automaton is similar to a control flow graph. Specifically, nodes capture locations and edge labels are statements.

```
1  int main() {
2    int p = nondet();
3    if (p) {
4       int i;
5       for (i = 0; i < 1000000; i++);
6       assert(i >= 1000000);
7    } else {
8       int x = 5;
9       int y = 6;
10      int r = x * y;
11      assert(r >= x);
12   }
13   return 0;
14 }
```

Figure 2: Non-linear arithmetic. Reproduced from CMC papers [4], [5].

the predicate `i < 1000000` *suffices to prove the assertion always holds when execution leaves the* `for`*-loop.*

The successful verification of the then *branch would make the trace composed of lines 2 (*`int p = nondet();`*), 3 (*`if (p)`*) and 4 (*`int i;`*) a* safe cone. △

III. Execution Reports (ERs)

We now formally define ERs as an under-approximation of the set of *safe cones* and $frontier$ traces. These definitions rely on two predicates (*analyzed?* and *isSafeCone?*) whose definition depends greatly on the underlying verification technique used in the incomplete verification attempt. Consequently, in this section we only provide properties that predicates *analyzed?* and *isSafeCone?* ought to satisfy. In the next section we ground the definition of these predicates for ART based verification techniques [16].

Examples 1 and 2 show how the notions of *analyzed* and *safe cone* apply to diverse techniques. We will capture these notions as predicates over traces in the following definitions.

Let the alphabet Σ contain all statements in a program, making $\pi \in \Sigma^*$ a sequence of statements.

Property 1. *The predicate* $analyzed? : \Sigma^* \to Bool$ *satisfies the following property, where* $\cdot$ *stands for concatenation:*

$\forall \pi, \pi' \in \Sigma^*.\ \neg analyzed?(\pi) \to \neg analyzed?(\pi \cdot \pi')$

Property 1 aims to formalize the notion of incrementality that we mentioned. Note that this property is logically equivalent to its contrapositive, that is, $analyzed?(\pi \cdot \pi') \to analyzed?(\pi)$, as expected of an incremental exploration.

Property 2. *The predicate* $isSafeCone? : \Sigma^* \to Bool$ *satisfies the following property:*

$\forall \pi, \pi' \in \Sigma^*.\ isSafeCone?(\pi) \to isSafeCone?(\pi \cdot \pi')$

Property 2 reflects our notion of *safe cone* as a trace reaching a fully analyzed portion of the behavior space. Any trace extension will necessarily also be a *safe cone*.

Property 3. *The predicates* $isSafeCone? : \Sigma^* \to Bool$ *and* $analyzed? : \Sigma^* \to Bool$ *satisfy the following property:*

$\forall \pi \in \Sigma^*.\ isSafeCone?(\pi) \to analyzed?(\pi)$

Property 3 connects the two predicates, capturing how *isSafeCone?*(π) subsumes *analyzed?*(π).

Definition 1. *Given a trace* $\pi \in \Sigma^*$ *and a program* $\mathcal{P}$*, we introduce the following predicate:*

$feasible_{\mathcal{P}}(\pi)$ *holds* iff *there exists a concrete execution of the program* $\mathcal{P}$ *that executes* π*.*

Given that we will always refer to a single program at a time, the system-under-analysis, we will omit the subscript.

Property 4. *Given* $\varphi : \Sigma^* \to Bool$*, a boolean predicate that captures the safety property of interest, the predicate* $analyzed? : \Sigma^* \to Bool$ *satisfies the following property:*

$\forall \pi \in \Sigma^*.\ analyzed?(\pi) \wedge\ feasible(\pi) \to \varphi(\pi)$

Property 4 is at the core of interpreting *analyzed* traces as providing safety assurances. This property also holds for *isSafeCone?* due to Property 3. The predicate $feasible(\pi)$ in the antecedent places the focus of safety assurances on feasible traces, that is, actual behaviors of the system-under-analysis.

Recall that we provide properties that constrain *analyzed?* and *isSafeCone?* but concrete definitions depend on the underlying technique. We now define the set of *safe cones* and *frontier traces* of an incomplete verification attempt.

Definition 2. *The set* $Frontier$ *is defined as follows:*

$Frontier = \{\pi \cdot s | \pi \in \Sigma^*, s \in \Sigma,\ analyzed?(\pi) \wedge feasible(\pi \cdot s) \wedge \neg analyzed?(\pi \cdot s)\}$

Definition 3. *The set* $SafeCone$ *is defined as follows:*

$SafeCone = \{\pi \cdot s | \pi \in \Sigma^*, s \in \Sigma, \neg isSafeCone?(\pi) \wedge feasible(\pi \cdot s) \wedge isSafeCone?(\pi \cdot s)\}$

Definitions 3 and 2 are related to Properties 2 and 1 respectively, since the incrementality of the analysis is key to the search for maximal *analyzed* traces, as in the set $Frontier$, and minimal *safe cone* traces, as in $SafeCone$.

The set $SafeCone$ can provide safety assurances about the system, as in Example 2, where the *then* branch of the `if`-statement has been fully verified.

Conversely, $Frontier$ can suggest shortcomings in the incomplete verification attempt. For instance, in Example 1, the existence of a trace $\pi \in Frontier$ that did not even go past the initialization loop suggests an important part of the test harness was not sufficiently analyzed.

The conclusions obtained from inspecting both sets can help assess the progress of an incomplete verification attempt.

We anticipated the intuitions captured by these definitions in Examples 1 and 2, but there is one important consideration that we omitted so far and only bring up now: feasibility.

Feasibility is relevant because, by definition, infeasible traces do not correspond to system behaviors. Without feasibility guarantees, interpreting each trace would require careful analysis, because it could mislead a user into either increasing or decreasing her confidence in the system-under-analysis.

Definition 4. *An* execution report *is a tuple* (S, F) *where* $S \subseteq SafeCone$ *and* $F \subseteq Frontier$*.*

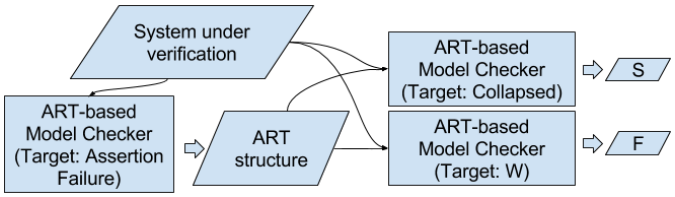

Figure 3: Architecture of ER generation.

Definition 4 defines ERs as under-approximations of the sets $SafeCone$ and $Frontier$, allowing empty sets as valid ERs.

The sets $SafeCone$ and $Frontier$ can grow quickly, making it impractical to compute the full sets. Furthermore, traces can be redundant, in a sense, if they only differ in a few statements, making it sensible to under-approximate.

Fully characterizing these approximations might be too restrictive, but some notion of adequacy seems necessary. We opted for a relaxed notion of *completeness*. For both sets, $SafeCone$ and $Frontier$, we require that if a trace $\pi \in SafeCone$ (resp. $Frontier$) ending in a specific location l exists, then there exists $\pi' \in S$ (resp. F) and π' also ends in l. This guarantee does not force extremely large sets of paths to be reported and loosely resembles a notion of coverage. Our algorithm guarantees this completeness criterion.

IV. Reports for ART-Based Implementations

Throughout this section we will explain how we generate ERs for ART-based [16] techniques.

ARTs constitute a relevant intermediate data structure used in verification. The variety of techniques implemented using ARTs makes them ideal for our proof-of-concept implementation. ART-based implementations comprise a wide range of dissimilar techniques, encompassing lazy abstraction [16], BMC [8], explicit value analysis [7] and CEGAR variants of some of the previous [7], among other.

In order to explain how we generate ERs for ART-based techniques, we first define *analyzed?* and *isSafeCone?* for these techniques in subsection IV-A. Finally, in subsection IV-B we will explain how we compute ERs from an earlier incomplete verification attempt, taking the ART structure and the system-under-verification as input, as depicted in Figure 3.

A. ARTs as Intermediate Data Structures

We have discussed how the concepts of *analyzed* and *safe cone*, captured by predicates *analyzed?* and *isSafeCone?* respectively, apply to example techniques. In this sub-section we will instantiate these concepts to ARTs [16].

An ART is a tree whose nodes correspond to vertices of the CFA and each node is associated to an element of an abstract domain. For example, a lattice of predicates in BLAST.

ART-based algorithms consist of two phases, the first one comprises an incremental generation of the ART and the second phase involves a more thorough counterexample check.

For the purpose of stating which traces can be considered *analyzed* we can ignore the abstract domain elements associated to each node. We can then think of an ART as a tuple $(G, W, q_0, covered)$ where $G = (S, \Sigma, \delta)$ is a graph, $W \subseteq S$ represents a wait list, $q_0 \in S$ is the initial state, $covered : S \to S$ captures subsumption between nodes, and $\delta : S \times \Sigma \to S$ is a transition function. The graph G captures the structure of the partially built ART. W is the wait list that contains elements to be analyzed in order to continue the construction of the ART. The function $covered$ is necessary to represent subsumption between nodes but is not total. We say that a node e is covered by e' *iff* $covered(e) = e'$. Analogously, we consider that a node e is not covered *iff* $covered(e)$ is undefined. Successors of a subsumed node e are not explored because the state space captured by e is also represented by $covered(e)$. Hence, subsumption is relevant to defining which parts of the state space to consider explored.

To make the definitions easier to read, we will assume the following invariant holds for ARTs relative to $covered$:

Property 5. *$\forall e \in S$. such that $covered(e)$ is defined, then $covered(covered(e))$ is not defined.*

In ART-based algorithms, a node e covered by e' need not be further analyzed because any error state found from e will also be found from e'. However, a sequence of statements that reaches e might not be a safe trace if it is possible to reach an assertion failure from e'. In other words, the sub-tree rooted in e cannot be considered exhaustively built unless that is also the case for the sub-tree rooted in e'. This observation is crucial to define what can be regarded as *analyzed* or *safe cone*.

We will extend δ as $\delta' : (S \cup \{None\}) \times \Sigma \to S \cup \{None\}$, with $None \notin S$ to make it total and resolve the $covered$ function transparently as follows:

$$\delta'(q,s) = \begin{cases} \delta(q,s) & \text{if } \delta(q,s) \text{ is defined and} \\ & covered(\delta(q,s)) \text{ is not} \\ covered(\delta(q,s)) & \text{if both } \delta(q,s) \text{ and} \\ & covered(\delta(q,s)) \text{ are defined} \\ None & \text{otherwise} \end{cases}$$

Due to Property 5, $\delta'(q, \pi)$ is never a covered node.

Moreover, we will adapt δ' to traces with $\hat{\delta'} : S \cup \{None\} \times \Sigma^* \to S \cup \{None\}$ as follows:

Given $q \in S \cup \{None\}$, $s \in \Sigma$ and $\pi \in \Sigma^*$:

$$\hat{\delta'}(q, s) = \delta'(q, s)$$
$$\hat{\delta'}(q, s \cdot \pi) = \hat{\delta'}(\delta'(q, s), \pi)$$

We will consider that *analyzed?*(π) holds for a trace π *iff* no prefix of π reaches a node in W from the initial node.

Definition 5. *We define $analyzed?$ for ARTs as:*

$$analyzed?(\pi) \text{ iff } \neg\exists \pi' \in \Sigma^*.isPrefix(\pi', \pi) \wedge \hat{\delta'}(q_0, \pi') \in W$$

Informally, we consider π *analyzed* when no prefix of π reaches a state pending exploration, i.e. one in W. The predicate *analyzed?* is clearly monotonic, satisfying Property 1: $\neg analyzed?(\pi)$ means there exists a prefix that reaches W, therefore any extension $\pi \cdot \pi'$ will also share that prefix.

Similarly, we will consider that *isSafeCone?*(π) holds for a trace π *iff* there exists a prefix of π that leads, from the initial node, to a sub-tree already exhaustively built. Intuitively, a sub-tree is exhaustively built when none of its nodes are in W, the set containing states pending exploration.

Definition 6. *We define* $isSafeCone?$ *for ARTs as:*

$$isSafeCone?(\pi) \text{ iff } \exists \pi' \in \Sigma^*.isPrefix(\pi', \pi) \wedge$$
$$\hat{\delta'}(q_0, \pi') \neq None \wedge \forall \pi'' \in \Sigma^*.\hat{\delta'}(q_0, \pi' \cdot \pi'') \notin W$$

Analogously, the predicate *isSafeCone?* is monotonic, satisfying Property 2: *isSafeCone?*(π) means there exists a prefix from which W is unreachable and consequently any extension $\pi \cdot \pi'$ will also share that prefix.

B. Generating Reports for CPAchecker

We built a proof-of-concept implementation, on top of CPAchecker, capable of generating ERs for ART-based techniques.

The input is an ART, without the abstract domain elements, and the original system-under-analysis. Our output are the sets S and F, composed of statement traces, generated by two independent verification tasks, as shown in Figure 3.

Although it would be possible to evaluate the predicates as stated in Definitions 5 and 6, CPAchecker already outputs a pre-processed representation of the ART structure, collapsing entirely verified sub-trees into a single node, and exports the resulting graph [4], [5]. This way, we generate F by performing successive reachability queries to ART nodes in W and, similarly, to generate S the target nodes are those resulting from the collapsed sub-trees. Informally, we augment an existing ART-based algorithm by composing in parallel the ART being built with the one taken as input.

We define the algorithm in terms of Configurable Software Verification [6], formalizing how different reachability analyses can be used. Due to space constraints we omit the details, which are present in a technical report [10]. It is worth noting that the algorithm used to generate an ER can be any ART-based technique, regardless of what was used for the original verification attempt.

V. Evaluation

In this section we briefly comment the conclusions of a preliminary evaluation we conducted. Due to space constraints we omit the details, available in a technical report [10]. We analyzed the performance of our proof-of-concept implementation with a set of standard benchmark instances. To conduct the experiment, we first analyzed the instances with two different techniques, lazy predicate abstraction and explicit value analysis, and collected the ARTs at the moment of reaching the 900 s time limit. Then we attempted to generate ERs for each of the ART collected.

When using lazy predicate abstraction for verification, it was possible to generate ERs within a fraction of the time originally allotted for verification for the majority of instances. Moreover, when using explicit value analysis, the set F could also be generated within a fraction of the verification time in most cases. Additionally, we manually inspected each of the ERs and gained non-trivial insights in most instances.

VI. Related Work

Conditional model checking [5] (CMC) is an approach where model checkers are extended to produce results even when the verification run could not be completed successfully. The output, in its general form, is a condition under which the program can be safely run. CPAchecker [6] instantiates CMC by generating an assumption automaton, a machine-readable encoding of the ART structure. We use this implementation to produce the ERs. ERs introduce the notions of *frontier* and *safe cone* to characterize the progress of an incomplete verification attempt. The lack of any feasibility guarantees in assumption automata has significant consequences. A user might be misled by the size of the assumption automaton since a vast automaton might correspond to a minuscule number of concrete feasible executions and also the other way around. We overcome this limitation by formally characterizing the properties of our output and adding feasibility guarantees.

A slightly different approach [12], [11] to providing feedback of partial verification results consists of a language extension to be used to annotate assumptions made during verification. This extension can be used to annotate the code and explicitly state conditions under which the program is guaranteed to run safely. These annotations are especially well-suited for local assumptions, sometimes used during manual verification, and for uniform assumptions, such as the absence of integer overflows, which are not affected by the context in which they occur. However, these annotations are not well-suited to state assumptions made by some techniques, such as those based in unrolling loops [12] or those tackled by assumption automata, and as such, they are incomparable to our approach, which can provide value in these cases. One of the use cases of these annotations is to complement the verification efforts and produce a small test suite. This idea of testing to complement earlier verification efforts was later replicated [13] using assumption automata as input.

A recent extension of the Dafny IDE [21], [11] provides, as one of its many features, hints about parts of a specification that might cause timeouts. However, given the modular nature of the tool and the sort of specifications shown as examples, the approach tackles a different problem than ours and the feature might not be applicable in our setting.

There is also previous work on quantifying partial model checker explorations based on usage profiles [22]. These estimations are based on abstract models of behavior and heavily depend on the provided usage profile. A similar approach [15] works applying symbolic execution over the source code of a system, without the need for an abstract model, but in this case, the implementation requires finite domains for all input variables, as well as a usage profile for each of them. Reliability as they define it can be hard to interpret and, once again, could be extremely sensitive to the usage profile provided. Both techniques can be used in conjunction with ours providing different value.

The modeling language Alloy [17] enables its users to specify structural properties. An extension of the Alloy Analyzer [14] highlights parts of the specification that are "problematic" or "hard", in the authors' own words, by monitoring the activity of variables and clauses in the underlying SAT solver. The output is inherently heuristic, whereas our technique provides strong guarantees backed by a formal definition of the properties of the output. The ideas behind the Alloy Analyzer extension could also be applied in conjunction with our techniques to provide additional information.

Our work is heavily influenced by the ideas and tool support of witness validation [3], [2], which we leverage as a machine-readable representation of exploration progress. However, we aim at enabling a richer manual interaction whereas that line of work also attempts to increase tool automation and reduce the need for manual inspection.

VII. Conclusions and Future Work

Software model checking is already capable of handling industrial instances and produce valuable results. However, some instances still remain intractable for full verification. Our work provides a different way to observe the progress of an incomplete verification attempt: by means of an ER.

We formulated the concepts of *analyzed* and *safe cone* traces and formalized the notion of ERs. We also implemented a proof-of-concept implementation and conducted a preliminary evaluation with standard benchmark instances.

We plan to explore ways to abstract the traces included in the ERs for easier visualization and understanding. Furthermore, we would define both existential and universal semantics for abstract traces in the sense that a property applies to the some or every concretization of these abstractions. For instance, guaranteeing that every concretization has been analyzed, could be useful and would pose interesting challenges. In a similar line, to better assess the relevance of an element $\pi \in S$, we need to devise meaningful views and metrics of the possible continuations of π, e.g. coverage metrics of the cone defined by π.

We would also like to analyze how our technique performs for a specific verification technique and varying time limits.

It would be desirable to conduct a user study to assess the effectiveness of our approach once more competing output representations become available.

We intend to look into alternative usages of our output. For example, elements of an ER can be leveraged as additional input to choose the right algorithm [23] to proceed after an incomplete verification attempt.

As mentioned in Section II, the concepts we present are also applicable to verification techniques beyond those already implemented using ARTs: Corral, DSE, inlining or unrolling-based, among other. It seems possible to adapt many of these tools to produce similar intermediate representations, making it possible to evaluate the generality of our approach.

Acknowledgments

This work was partially supported by the projects ANPCYT PICT 2013-2341, 2014-1656, 2015-3638 and 2015-1718, UBACYT 20020130300036BA and 20020130100384BA and CONICET PIP 11220130100688CO and 11220150100931CO.

References

[1] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Communications of the ACM*, 54(7):68–76, 2011.

[2] D. Beyer. Software verification and verifiable witnesses. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–416. Springer, 2015.

[3] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 721–733. ACM, 2015.

[4] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking. *arXiv preprint arXiv:1109.6926*, 2011.

[5] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: a technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 57. ACM, 2012.

[6] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification*, pages 504–518. Springer, 2007.

[7] D. Beyer and S. Löwe. Explicit-state software model checking based on cegar and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013.

[8] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320. ACM, 1999.

[9] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[10] R. Castaño, V. Braberman, D. Garbervetsky, and S. Uchitel. Model checker execution reports. *arXiv preprint arXiv:1607.06857*, 2016.

[11] M. Christakis, K. R. M. Leino, P. Müller, and V. Wüstholz. Integrated environment for diagnosing verification errors. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 424–441. Springer, 2016.

[12] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM 2012: Formal Methods*, pages 132–146. Springer, 2012.

[13] M. Czech, M.-C. Jakobs, and H. Wehrheim. Just test what you cannot verify! In *Fundamental Approaches to Software Engineering*, pages 100–114. Springer, 2015.

[14] N. D'Ippolito, M. F. Frias, J. P. Galeotti, E. Lanzarotti, and S. Mera. Alloy+ hotcore: A fast approximation to unsat core. In *Abstract State Machines, Alloy, B and Z*, pages 160–173. Springer, 2010.

[15] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 622–631. IEEE Press, 2013.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 37(1):58–70, 2002.

[17] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[18] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.

[19] A. Lal and S. Qadeer. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 202–212. ACM, 2014.

[20] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *International Conference on Computer Aided Verification*, pages 427–443. Springer, 2012.

[21] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[22] E. Pavese, V. Braberman, and S. Uchitel. My model checker died!: how well did it do? In *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, pages 33–40. ACM, 2010.

[23] V. Tulsian, A. Kanade, R. Kumar, A. Lal, and A. V. Nori. Mux: algorithm selection for software model checkers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 132–141. ACM, 2014.