Using Symbolic Execution for Verifying Safety-Critical Systems

Alberto Coen-Porisini
Dipartimento di Ingegneria dell'Innovazione
Università di Lecce
via per Monteroni
I-73100 Lecce, Italy
coen@ultra5.unile.it

Giovanni Denaro, Carlo Ghezzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci, 32
I-20133 Milano, Italy
{denaro,ghezzi}@elet.polimi.it

Mauro Pezzè
Dipartimento di informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
via Bicocca degli Arcimboldi 8
I-20126 Milano, Italy
pezze@disco.unimib.it

ABSTRACT

Safety critical systems require to be highly reliable and thus special care is taken when verifying them in order to increase the confidence in their behavior. This paper addresses the problem of formal verification of safety critical systems by providing empirical evidence of the practical applicability of symbolic execution and of its usefulness for checking safety-related properties. In this paper, symbolic execution is used for building an operational model of the software on which safety properties, expressed by means of a Path Description Language (PDL), can be assessed.

Keywords

Symbolic execution, safety-critical system, verification, formal methods.

1. INTRODUCTION

Safety critical systems such as aircraft avionics, nuclear power plant control and patient monitoring require to be highly reliable, since failures in this kind of systems can have catastrophic consequences.

Formal methods can successfully enhance the quality of software [3, 5, 23, 25, 29] and therefore increase the confidence in the system behavior. Unfortunately, formal methods are often neglected in practice: in industrial projects software verification usually relies on techniques such as code inspection and testing, while correctness is very seldom formally proved.

Symbolic execution is a well known formal technique that has been proposed for many activities such as symbolic debugging [20], test data generation [7, 26, 27], verification of program (partial) correctness [12, 19] and program reduction [8, 10]. However, symbolic execution is not currently used in industrial environments although several prototypes have been built. The main reasons for this are:

1. the inadequacy of existing symbolic executors in dealing with dynamic data structures, loops, etc;
2. the necessity of coupling symbolic executors with expression simplifiers and/or theorem provers, which, in general, are difficult to use and require *ad hoc* skills;
3. the (alleged) low cost effectiveness of introducing any formal technique (and symbolic execution in particular) into a real-life software development process.

Although, in general, the above reasons may be true, we argue that symbolic execution can become an applicable technique for proving important safety properties of a significant class of industrial safety critical systems. In fact, several important producers of safety critical systems, in particular in the field of avionics and nuclear power plant systems, develop software using subsets of the C language to avoid linguistic features, such as dynamic allocations of memory, that are likely to cause failures difficult to detect during verification. The subsets of C used in most of such applications closely correspond to the subset of C that can be effectively analyzed with symbolic execution. In addition, the implementation of such systems usually leads to software whose structure is simplified by limiting, for instance, the way in which loops are used, by avoiding recursion, and by bounding the complexity of control conditions. Modern expression simplifiers can automatically simplify most of the conditions and constructs used in such applications. Finally, a more rigorous verification of safety critical system is becoming increasingly important in order to assess the strong safety requirements of these systems, thus increasing the economic convenience of formal verifications over a pure testing based verification approach [6].

This paper addresses the problem of formal verification of safety critical software systems implemented using a subset of the C language referred to as *Safer-C* [17]. The paper provides empirical evidence of the practical applicability of symbolic execution for checking safety-related properties for this kind of systems. In particular, symbolic execution is used for building operational models of the software on which safety properties can be assessed. Such properties are expressed as predicates of execution paths by means of a Path Description Language (PDL). Therefore, the combination of symbolic execution and safety property assessment can effectively increase the confidence in the behavior of safety critical applications.

The paper is organized in the following way: Section 2 provides an overview of symbolic execution. Section 3 discusses the proposed analysis methodology, including the definition of PDL. Section 4 presents an environment that supports the described analysis framework, and discusses its current limitations. Section 5 reports on the experiments that have been carried out. Finally, Section 6 discusses the related works found in literature, while Section 7 draws some conclusions.

2. SYMBOLIC EXECUTION

The main idea behind symbolic execution is to use symbols instead of *numbers* as input values, and to represent the values of program variables with symbolic expressions. As a result, the output values computed by a program are expressed as function of the program input symbols.

In order to describe the state of a symbolically executed program the instruction counter and the set of pairs <variable, value> that usually defines the state of a *numeric execution*, is not sufficient. As an example, let us consider the program Ex1 shown in Figure 1, in which each statement is labeled for the sake of clarity.

```
L1:  int example(int data) {
L2:  int out;
L3:  if (data > 0)
L4:     out = data;
L5:  else
L6:     out = -data;
L7:  return(out);
L8:  }
```

Figure 1: The sample program Ex1.

Let the symbol α represent the value of the input parameter `data`. When executing the conditional statement, the value of `data` does not allow one to choose along which branch the computation has to continue. Hence, one has to assume either $\alpha > 0$ to execute the *then* branch or $\alpha \leq 0$, to execute the *else* branch. Such assumptions are represented by means of a first order predicate referred to as *path condition* (PC). Thus, the state of a symbolic execution is represented by a triple <IP, PC, V>, where

- IP is the instruction pointer referring to the statement to execute next;
- PC is the path condition;
- V is the set of pairs <variable, expression>.

Initially, IP points to the first statement, the input parameters are initialized using new symbols while other variables are initialized to the special value *Undef*. PC is set to *true*, that is no assumption is made on the values of variables[1].

Symbolic execution can be formalized by means of a function representing the execution of a single program statement. Let *Exec* denote such a function:

$$\text{Exec: SymbolicState} \times \text{Program} \rightarrow 2^{\text{SymbolicState}}$$

Exec takes as input a symbolic state and a program and returns the symbolic states[2] resulting from the execution of the current statement (i.e., the statement referred to by IP). The semantics of Exec can be informally described as follows:

- *Assignment statement*. The right hand side is evaluated. The resulting expression is then assigned to the left hand side variable in the output state. IP points to the next statement of the code and PC remains unchanged.
- *Conditional statement* (`if (C)` S_{then} `else` S_{else}). If PC implies neither C nor $\neg$C, Exec returns two symbolic states, corresponding to the two different branches, in which both IP and PC have been modified. IP points to the first statement of the `then` or the `else` branch, respectively, while PC is obtained as the logical *and* of the previous PC and the evaluation of C or $\neg$C, respectively. Conversely, if PC implies C ($\neg$C) only the `then` (`else`) branch can be executed. Thus, Exec returns a single symbolic state in which only IP has been modified to point to the first statement of the `then` (`else`) branch.
- *Loop statement* (`while (C)` S_{loop}). This statement is handled as the following equivalent statement:

```
if(C) {
        Sloop;
        while (C) Sloop
}
```

  Thus, the loop is unfolded and Exec is recursively applied, so that at each iteration new symbolic states are constructed.
- *Procedure calls*. Procedure calls are managed by means of macro-expansion. The symbolic execution jumps to the invoked sub-program and executes it, making the necessary assignments to represent parameters passing and return.

The interested reader can find details on how symbolic execution can be formally defined in [7, 20], while Section 6 discusses the state of the art to the best of our knowledge.

3. VERIFYING SAFETY CRITICAL SYSTEMS

The software embedded in safety critical systems is often written in a subset of the C programming language, with the aim of reducing failures that are caused by potentially hazardous linguistic features. For instance, critical software for avionics and nuclear power plants is written in a subset of the C language known as Safer-C [17]. The most significant limitations of Safer-C are:

- pointers can be used only for *by-reference* parameter passing and they must be limited to only one level indirection;
- memory cannot be handled dynamically;
- recursion is not allowed;
- union types, implicit type casting, and goto statements are forbidden;
- files are not used.

The reasons behind the classification of some C constructs as hazardous are out of the scope of this paper. The interested reader can refer to [17] for a thorough discussion.

[1] Any possible pre-condition on the input parameters can be defined by initializing PC.

[2] Branching statements such as conditional statements, loops etc. can return more than one symbolic state.

Symbolic execution can effectively handle only a subset of programming constructs. In particular, the main limitations concern dynamic handling of memory. Since Safer-C forbids dynamic handling of memory, symbolic execution can be effectively applied to Safer-C programs, and thus it represents a promising analysis method for such programs. In this paper we provide some experimental evidence that symbolic execution is indeed an effective technique for analyzing important properties of programs written in Safer-C.

In what follows, we introduce first the concept of *execution model* that represents the execution space of a program under analysis, produced through symbolic execution, and then the PDL language, which is used to describe properties concerning execution models.

3.1 Execution Models

An *execution model* of a program P describes all the (symbolic) states that can be reached during a computation. It is a direct graph whose nodes are symbolic states, and it is formally defined as a tuple:

EM(P) = <N, n_0, E, T> where:

- N is the set of nodes, i.e., symbolic states reached during the symbolic execution of P;
- $n_0 \in N$, is the initial node, i.e., the initial symbolic state;
- $E = \{(n_i, n_j) \in N \times N \mid n_j \in Exec(n_i, P)\}$ is the set of edges connecting the different nodes.
- $T \subseteq N$ is the set of terminal nodes, i.e., final states of the symbolic execution.

Notice that an execution model, in general, can contain an unlimited number of states and therefore the corresponding graph can have paths of unlimited length. The main reason for this being the way in which loops are handled, that is by unfolding them. For example, Figure 2 shows the execution model of program Ex1 of Figure 1.

An *execution path* is a sequence of nodes $(n_1;\ldots;n_k)$ such that:

- $n_i \in N$, $1 \le i \le k$;
- n_1 equals n_0;
- $(n_i, n_{i+1}) \in E$, $1 \le i \le k-1$;
- $n_k \in T$.

Thus, an execution path represents a single terminating computation belonging to an execution model. For instance, in the execution model of Figure 2, it is possible to identify two different execution paths corresponding to executing the *then* and *else* branches of program Ex1, respectively.

3.2 The Path Description Language

PDL is a formal language that allows one to express structural program properties, that is properties concerning reachable statements and execution paths of a program. Let us consider a program P and suppose that every statement therein is labeled: let $Lab_1, \ldots, Lab_n$ denote such labels. In what follows first PDL syntax is defined in a classical inductive way by introducing PDL *terms* and *expressions* and then semantics is informally discussed.

3.2.1 PDL Terms

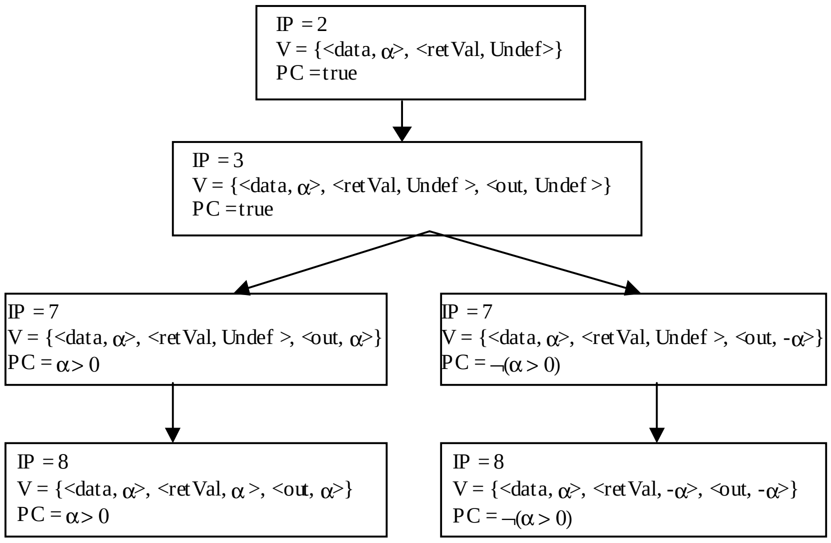
PDL terms are defined in a classical inductive way:

- Given a label Lab_i, $1 \le i \le n.$, Lab_i is a PDL term;
- Given two labels Lab_i, Lab_j, $1 \le i, j \le n$ and a PDL term t, $Lab_i \rightarrow Lab_j$, $Lab_i \rightarrow (t) \rightarrow Lab_j$ and $\neg(t)$ are PDL terms;
- Nothing else is a PDL term.

3.2.2 PDL Expressions

PDL expressions are defined in the following inductive way:

- Given a PDL term t, $\forall_p$ (t) and $\exists_p$ (t) are PDL expressions;
- Given a PDL expression e, $\neg$(e) is a PDL expression;
- Nothing else is a PDL expression.



IP = 2
V = {<data, α>, <retVal, Undef>}
PC = true
IP = 3
V = {<data, α>, <retVal, Undef >, <out, Undef >}
PC = true
IP = 7
V = {<data, α>, <retVal, Undef >, <out, α>}
PC = α > 0
IP = 7
V = {<data, α>, <retVal, Undef >, <out, -α>}
PC = ¬(α > 0)
IP = 8
V = {<data, α>, <retVal, α >, <out, α>}
PC = α > 0
IP = 8
V = {<data, α>, <retVal, -α>, <out, -α>}
PC = ¬(α > 0)

Figure 2 The execution model for program Ex1

3.2.3 PDL Semantics

A PDL term and/or expression must be evaluated over an execution model and can be either true or false. The main difference between a term and expression being that the value of the former refers to a given execution path belonging to the execution model, while the value of the latter refers to the execution model as a whole. In other words a PDL expression can be viewed as a closure over all execution paths existing in an execution model.

Let EM(P) denote an execution model of program P. The semantics of PDL terms can be informally defined in the following way:

- Lab_i evaluates to *true* on a given execution path $p \in EM(P)$ if and only if p contains the node corresponding to statement Lab_i;
- $Lab_i \rightarrow Lab_j$ evaluates to *true* on a given execution path $p \in EM(P)$ if and only if p contains the node corresponding to Lab_j followed by node corresponding to Lab_j. Nodes Lab_i and Lab_j may be separated by zero or more nodes on path p;
- $Lab_i \rightarrow (t) \rightarrow Lab_j$ evaluates to *true* on a given execution path $p \in EM(P)$ if and only if term $Lab_i \rightarrow Lab_j$ evaluates to *true* on p and t evaluates to *true* on q, where q is a sub-path contained in p;
- $\neg(t)$ evaluates to *true* on a given execution path $p \in EM(P)$ if and only if term t evaluates to *false* on p.

The semantics of PDL expressions can be informally defined as follows (for the formal definition see [21]):

- $\forall_p$ (t) evaluates to *true* on EM(P) if and only if for all execution paths $p \in EM(P)$ t evaluates to *true*;
- $\exists_p$ (t) evaluates to *true* on EM(P) if and only if there is at least one execution path $p \in EM(P)$ on which t evaluates to *true*;
- $\neg(e)$ evaluates to *true* on EM(P) if and only if e evaluates to *false* on EM(P)[3].

For example, the following PDL expressions referring to program [Ex1] evaluates to *true* on the execution model of Figure 2:

$$\forall_p (L8),\ \exists_p (L1 \rightarrow L4) \text{ and } \exists_p (L1 \rightarrow \neg L4 \rightarrow L8).$$

The first expression states that all execution paths reach the *return* statement (L8). The second one states that there exists at least one execution path that reaches first statement L1 (beginning of the procedure) and then the assignment of the *then* branch (L4). Finally the last expression states that there exists at least one execution path that reaches the *return* statement starting from the beginning of the procedure and that does not reach the assignment of the *then* branch.

Properties assessment for a given program can be done by combining symbolic execution and PDL. The former is used to build the execution model of the program, while the latter is used to verify which properties hold and which do not. Moreover, besides checking whether a given property is true, one can also identify on which execution paths the property holds. Thus, a PDL term can be used to select the execution paths belonging to an execution model that satisfy it. In this way execution paths representing behaviors particularly relevant for the program can be selected and analyzed by looking at the way in which the symbolic state evolves.

[3] It can be easily proved that $\exists_p$ (t) is equivalent to $\neg(\forall_p (\neg t))$.

4. THE TOOL

This section describes the Symbolic execution Aided Verification Environment (SAVE) that has been designed and implemented to support the analysis of safety critical application. SAVE has been used to validate the approach proposed in this paper and is characterized by the following main functionalities:

- **Execution Model Generator.** Given a program written in Safer-C, the Execution Model Generator allows the user to build the corresponding execution model. Users can provide an initial precondition in term of logical constraints on the values of the input variables to restrict the analysis to a subset of the whole execution model. When a branching statement is reached the user can choose along which branches the symbolic computation has to continue, or else can decide to automatically follow all the branches. By allowing branch selection, one can avoid the infinite execution problem that may arise because of loop unfolding. All reached symbolic states are recorded in the execution model.
- **PDL Property Checker**. It allows users to check properties expressed in PDL over the execution model built with the Execution Model Generator. The PDL Property Checker identifies all paths of the execution model for which the given property is satisfied.
- **Execution Model Inspector**. Execution models can be visualized as a graph whose nodes represent symbolic states and are labeled with the program statements to be executed next. Users can inspect the content of each node, i.e., the symbolic values associated with the program variables and the path condition.

Figure 3 illustrates, in UML style, the logical components of SAVE and their relative dependencies.

From an architectural viewpoint, the tool is composed of two main subsystems, namely the *symbolic execution subsystem* and the *model handler subsystem* that provide the above mentioned functionalities. A graphical user interface enables users to interact with the tool. In what follows the subsystems are presented in more detail.

4.1 The Symbolic Execution Subsystem

The symbolic execution subsystem provides capabilities for loading a program, initializing the symbolic state, symbolically executing a statement, managing the symbolic state and accessing its content. This subsystem contains a component, which we refer to as *symbolic executor*, implementing the symbolic execution algorithms. The symbolic executor works on an intermediate representation of the input C program, obtained by means of the publicly available LCC compiler [30]. Although LCC processes any C program, the symbolic executor only accepts programs written in Safer-C.

The symbolic executor uses an algebraic simplifier when evaluating a branching point condition in order to determine along

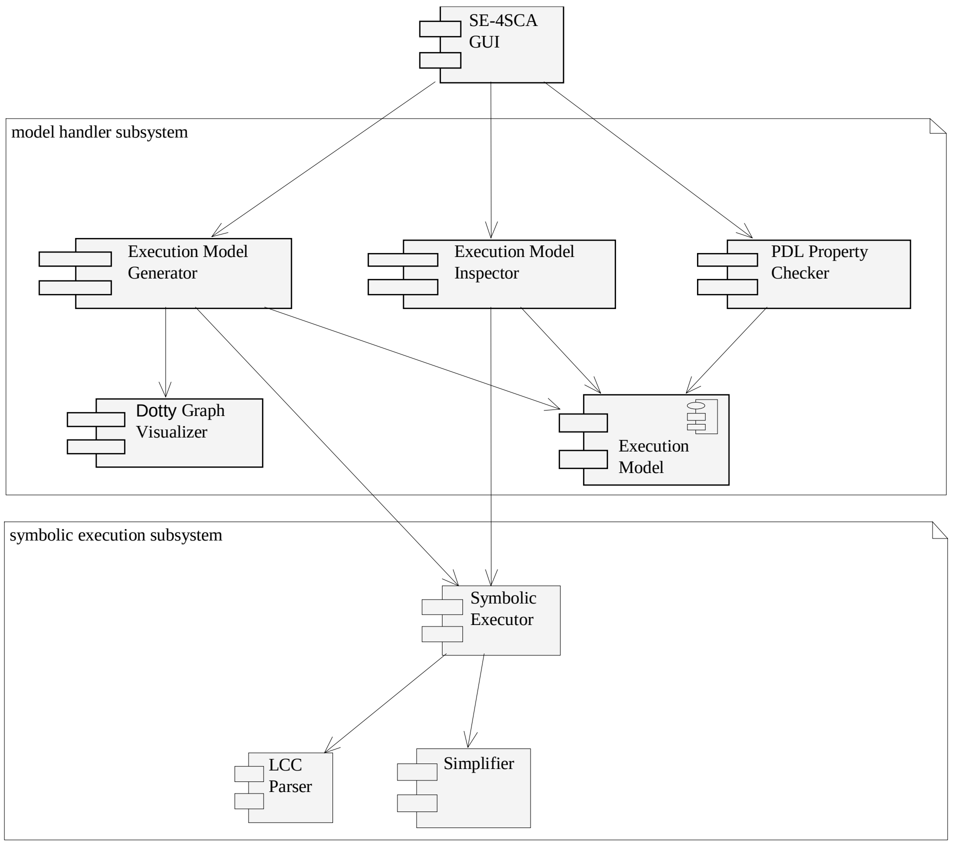


Figure 3: Component diagram for SE-4SCA

which branch the execution continues. Whenever the algebraic simplifier is not able to decide which branch has to be followed, the decision is taken by an external component suitably invoked. Such component must support a pre-defined interface and must have registered as *decision-maker* using the *registration service* provided by the symbolic execution subsystem.

In this way, different components can act as decision-makers using different policies. This point will be further discussed in the next sub-section.

4.2 The Model Handler Subsystem

The model handler subsystem uses the symbolic execution subsystem to build and handle the execution model. Moreover, it contains a specific package for handling the data structure used to store an execution model. Since storing the full execution model can require an enormous amount of memory, only the structure of the execution model (i.e., the underlying graph with references to the executed program statements) is memorized.

The model handler interacts with the user through the graphical interface and provides capabilities for

1. *building the execution model* (execution model generator). The execution model generator uses the symbolic executor to compute the different symbolic states composing the execution model. In this case, it is registered as decision-maker for the symbolic execution subsystem and thus, it is invoked by the symbolic execution subsystem every time the algebraic simplifier fails in evaluating a condition. The execution model generator, in turn, asks the user which branch he/she wants to follow. If the user wants to follow all the branches, the execution model generator takes care of backtracking the different symbolic states in order to build the execution model. The execution model generator uses the public Dotty graph layout manager [31] for visualizing the execution model. The graphical visualization of the execution space can be very useful for understanding the behavior of small portions of the code, but it is less useful for large spaces. Thus, it can be used for inspecting the execution space of small programs or small portions of large programs that are particularly relevant for the safety of the system;
2. *checking PDL properties* (PDL property checker). The PDL property checker evaluates a given a PDL expression over the execution model built by means of the execution model generator. Beside providing the resulting boolean value, this component actually identifies all the execution paths belonging to the execution model that satisfy the PDL expression;
3. *inspecting symbolic states* (execution model inspector). The execution model inspector allows the inspection of symbolic states of computed execution models. Since symbolic states are transparently re-computed before inspecting their content, this component is registered as decision-maker and it is invoked by the symbolic executor when re-computing a

given symbolic state. In this case it automatically provides the requested answer since it knows which branch has to be followed when re-computing the symbolic state.

4.3 Performance Issues

In the current implementation each symbolic state is represented with 36 bytes of data, and thus the tool is able to memorize execution models of 10^4 states with 1MB of memory and of 10^7 states with 1GB of memory. For the medium-size industrial examples we have analyzed, the number of states of the computed models ranges between 10^3 and 10^4 states (see Section 5 for more details on the experiments). Since the core of large safety critical applications does not usually exceed some tenths of thousands of lines of code, an up-to-date server can easily provide the resources required for analyzing such systems.

The time required to transparently re-compute a symbolic state is linear in the length of the path from the initial state to the considered state. In our experiments, the re-computations of a symbolic state never exceeded few seconds on a Sparc workstation equipped with Solaris operating system.

Reducing the number of symbolic states stored for checking a PDL property can further increase the (already acceptable) size of the applications analyzable with the current prototype. Currently, the prototype builds the execution model of the relevant execution space before starting to check a given property. This may cause the construction of many states that are not relevant to check the property. Therefore, the tool can be improved by first statically identifying the relevant execution paths on the control-flow graph of the program, and then by building only the symbolic states for selected paths.

5. THE CASE STUDY

SAVE has been developed as part of a European project and has been validated on benchmarks provided by our industrial partners. In particular, it has been used to investigate important safety and liveness properties of avionics and nuclear plant applications. Due to confidentiality reasons, we cannot report on these cases. Thus, we selected a well-known and publicly available application from one of these application domains to illustrate the approach proposed in this paper: the Traffic Alert and Collision Avoidance System (TCAS). TCAS is an on-board aircraft conflict detection and resolution system used by all US commercial aircraft, whose specification can be found in [28]. TCAS has been studied by both the industry and the academia [1], [18], [24] and can be considered as a benchmark for safety critical applications. In what follows, TCAS is informally described and then the different experiments carried out using SAVE are presented.

5.1 TCAS

TCAS continuously monitors the radar information to check whether there is any neighbor aircraft that could represent a potential threat by getting too close. In such a case, it is said that an *intruder* aircraft is entering the *protected zone*. Whenever an intruder aircraft enters the protected zone, TCAS issues a *Traffic Advisory* (TA) and estimates the time remaining until the two aircraft reach the closest point of approach (and then –hopefully– begin to fly away from each other). Such estimate is used to calculate the vertical separation between the two aircraft assuming that the controlled aircraft either maintains its current trajectory or performs immediately an upward (downward) maneuver. Depending on the results obtained, TCAS may issue a *Resolution Advisory* (RA) suggesting the pilot either to climb or to descend.

The experimental work described in this section focuses on the component of TCAS that is responsible for finding the best RA. The component, which is made up of 120 lines of code, comes from a set of programs used by Hutchins et al. in a previous experiment [34]. Figure 4 shows the code of procedure `alt_sep_test()`, which is referred to in all the examined

```
      int alt_sep_test() {
         bool enabled, tcas_equipped, intent_not_known;
         bool need_upward_RA, need_downward_RA;
         int alt_sep;
ASTBeg: enabled = High_Confidence &&
                  (Own_Tracked_Alt_Rate <= OLEV) &&
                  (Cur_Vertical_Sep > MAXALTDIFF);
         tcas_equipped = (Other_Capability == TCAS_TA);
         intent_not_known = Two_of_Three_Reports_Valid &&
                            Other_RAC == NO_INTENT;
         alt_sep = UNRESOLVED;
         if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
ASTEn: {
         need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
         need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
         if (need_upward_RA)
ASTUpRA:    alt_sep = UPWARD_RA;
         else if (need_downward_RA)
ASTDownRA:       alt_sep = DOWNWARD_RA;
              else
ASTUnresRA:     alt_sep = UNRESOLVED;
         }
         return alt_sep;
      }
```

Figure 4 - The code of `alt sep test()`

PDL properties.

The vertical separation between the two aircraft is represented by means of the global variable `Current_Separation`, while `Up_Separation` and `Down_Separation` represent the estimated vertical separation after a climbing maneuver and a descending maneuver, respectively. These variables can be regarded as input data for the analyzed sub-system. The vertical separation at the closest point of approach is considered *adequate* if it is greater than a threshold value (`Positive_RA_Thresh_Value`), which can be considered as a system constant.

The code contains five labels that identify the core execution states, and precisely:

- *ASTBeg* identifies the beginning of the actual code.
- *ASTEn* identifies the statement where the analyzed subsystem starts the computation for selecting the best escape maneuver.
- *ASTUpRA* identifies the statement where a climbing RA is selected.
- *ASTDownRA* identifies the statement where descending RA is selected.
- *ASTUnresRA* identifies the statement where no RA is selected.

5.2 Experiments

In what follows the effectiveness of the approach is illustrated by showing how SAVE can be used to prove some important properties of TCAS identified in [35]. Other proven properties are reported in [21].

All properties considered in this section refer to the possibility of issuing either an upward or a downward RA. The PDL expressions describing the issuing of an upward and a downward RA are respectively:

❑ $\exists_p$ (ASTEn → ASTDownRA) [PrA]
i.e., there exists at least an execution path such that the starting of the computation for selecting the best escape maneuver (*ASTEn*) is followed by the selection of a descending RA (*ASTDownRA*)

❑ $\exists_p$ (ASTEn → ASTUpRA) [PrB]
i.e., there exists at least an execution path such that the starting of the computation for selecting the best escape maneuver (*ASTEn*) is followed by the selection of a climbing RA (*ASTUpRA*)

The properties taken into account are proved by evaluating the above two PDL expressions over the execution models built using different initial conditions.

5.2.1 P1: Safe advisory selection

This property states that whenever TCAS can issue either a safe or a non-safe advisory, the latter never occurs.

P1. *If one maneuver produces adequate separation and the other one does not, then the RA corresponding to the maneuver that does not produce adequate separation is never issued.*

This property is verified if

a) expression [PrA] is not satisfied when the following initial condition holds:

```
Up_Separation ≥ Positive_RA_Alt_Thresh and
Down_Separation < Positive_RA_Alt_Thresh
```

i.e., the separation resulting from climbing is adequate, while the separation resulting from descending is not,

and

b) expression [PrB] is not satisfied when the following initial condition holds:

```
Up_Separation < Positive_RA_Alt_Thresh and
Down_Separation ≥ Positive_RA_Alt_Thresh
```

i.e., the separation resulting from descending is adequate, while the separation resulting from climbing is not.

The “safe advisory selection” property has been proved by building the execution models for both cases. Each of the two models comprises 4228 symbolic states and 58 execution paths.

5.2.2 P2: Best advisory selection

The “best advisory selection” property states that whenever no safe advisory exists, the worst unsafe advisory is never selected.

P2. *Let neither climb nor descend maneuvers produce adequate separation. Then, the RA corresponding to the maneuver that produces less separation is never issued.*

This property is verified if

a) expression [PrA] is not satisfied when the following initial condition holds:

```
Up_Separation < Positive_RA_Alt_Thresh and
Down_Separation < Positive_RA_Alt_Thresh and
Up_Separation > Down_Separation
```

i.e., neither separations are adequate, but the separation resulting from descending is worse than the one resulting from climbing,

and

b) expression [PrB] is not satisfied when the following initial condition holds:

```
Up_Separation < Positive_RA_Alt_Thresh and
Down_Separation < Positive_RA_Alt_Thresh and
Up_Separation < Down_Separation
```

i.e., neither separations are adequate, but the separation resulting from climbing is worse than the one resulting from descending.

Proving the “best advisory selection” property led to building the execution models for both cases, each of which comprising 4100 symbolic states and 62 execution paths.

5.2.3 P3: Avoid unnecessary crossing

The “avoid unnecessary crossing” property states that whenever both climbing and descending are safe a *crossing* advisory is never selected. A crossing occurs when the two aircraft switch their relative altitudes, and thus at a certain moment during the maneuver the altitudes of the two aircraft become equal. A crossing is risky and thus it should be always avoided.

Property	Execution model			Property checking	
	assumptions	Num. of states	Num. of paths	PDL property	Num. of satisfying paths
P1	`Up_Separation≥Positive_RA_Alt_thresh and Down_Separation<Positive_RA_Alt_Thresh`	4228	58	∃p(ASTEn→ ASTDownRA)	0
	`Up_Separation<Positive_RA_Alt_thresh and Down_Separation≥Positive_RA_Alt_Thresh)`	4228	58	∃p(ASTEn→ ASTUpRA)	0
P2	`Up_Separation<Positive_RA_Alt_Thresh and Down_Separation<Positive_RA_Alt_Thresh and Up_Separation>Down_Separation`	4100	62	∃p(ASTEn→ ASTDownRA)	0
	`Up_Separation<Positive_RA_Alt_Thresh and Down_Separation<Positive_RA_Alt_Thresh and Up_Separation<Down_Separation`	4100	62	∃p(ASTEn→ ASTUpRA)	0
P3	`Up_Separation>=Positive_RA_Alt_Thresh and Down_Separation>=Positive_RA_Alt_Thresh and Own_Tracked_Alt>Other_Tracked_Alt`	2212	38	∃p(ASTEn→ ASTDownRA)	0
	`Up_Separation>=Positive_RA_Alt_Thresh and Down_Separation>=Positive_RA_Alt_Thresh and Own_Tracked_Alt<Other_Tracked_Alt`	2212	38	∃p(ASTEn→ ASTUpRA)	0
P4	`Own_Tracked_Alt>Other_Tracked_Alt`	3576	54	∃p(ASTEn→ ASTDownRA)	16
	`Own_Tracked_Alt<Other_Tracked_Alt`	3576	54	∃p(ASTEn→ ASTUpRA)	16
P5	`Up_Separation>Down_Separation`	6988	90	∃p(ASTEn→ ASTDownRA)	0
	`Up_Separation<Down_Separation`	6988	90	∃p(ASTEn→ ASTUpRA)	0

Table 1 Summary of the Experimental Results

P3. *If both climbing and descending produce adequate separation then a crossing RA is never issued.*

To formalize this property, we need to refer to the variables `Own_Tracked_Alt` and `Other_Tracked_Alt` that denote the altitude of the two aircraft.

This property is verified if:

a) expression [PrA] is not satisfied when the following initial condition holds:

```
Up_Separation ≥ Positive_RA_Alt_Thresh and
Down_Separation ≥ Positive_RA_Alt_Thresh and
Own_Tracked_Alt > Other_Tracked_Alt
```

i.e., both separations are safe and the controlled aircraft is at a higher altitude than the intruder aircraft,

and

b) expression [PrB] is not satisfied when the following initial condition holds:

```
Up_Separation ≥ Positive_RA_Alt_Thresh and
Down_Separation ≥ Positive_RA_Alt_Thresh and
Own_Tracked_Alt > Other_Tracked_Alt
```

i.e., both separations are safe and the controlled aircraft is at a lower altitude than the intruder aircraft.

Also this property has been proved by building the execution models for both cases. Each model comprised 2212 symbolic states and 38 execution paths.

5.2.4 *No crossing advisory selection*

The "no crossing advisory" selection property restricts the "avoid unnecessary crossing" property by stating that a crossing advisory can never be issued, regardless of the upward and downward separations.

P4. *A crossing RA is never issued.*

This property is verified if

a) expression [PrA] is not satisfied when the following initial condition holds:

```
Own_Tracked_Alt > Other_Tracked_Alt
```

i.e., the controlled aircraft is at a higher altitude than the intruder aircraft,

and

b) expression [PrA] is not satisfied when the following initial condition holds:

```
Own_Tracked_Alt < Other_Tracked_Alt
```

i.e., the controlled aircraft is at a lower altitude than the intruder aircraft.

The "no crossing advisory selection" property cannot be proved. In fact, the property checker identified some execution paths for which either [PrA] or [PrB] is satisfied under the stated initial conditions. Inspecting the symbolic states belonging to such execution paths, we can identify counterexamples. For example, in the case (a), a crossing advisory is issued if the descend maneuver does not produce adequate separation while the climb maneuver produces more separation than the descend maneuver.

5.2.5 P5: Optimal advisory selection

The “optimal advisory selection” property states that TCAS never issues the RA that leads to the smaller separation. This property is interesting because it is stronger than (and subsumes) P1 and P2.

P5. *The RA that produces less separation is never issued.*

This property is verified if

a) expression [PrA] is not satisfied when the following initial condition holds:

```
Up_Separation > Down_Separation
```

i.e., the separation obtained by climbing is greater than the one obtained by descending,

and

b) expression [PrB] is not satisfied when the following initial condition holds:

```
Down_Separation > Up_Separation
```

i.e., the separation obtained by descending is greater than the one obtained by climbing.

The optimal advisory selection property has been proved by building the execution models for both cases. Each of the two models comprises 6988 symbolic states and 90 execution paths.

Table 1 summarizes the results of the experiments. Each property corresponds to two rows, one for each assumption to be analyzed. Each row indicates (1) the number of states and paths of the execution model built for checking the assumption and (2) the number of paths produced by the property checker when trying to satisfy the PDL expressions. An empty number of paths indicates that the PDL expression is not satisfiable, while a non empty number of paths indicates the number of paths for which the PDL property is satisfied.

6. RELATED WORKS

Many environments supporting symbolic execution have been built, some of which are reviewed by Corward in [9]. Girgis experimentally evaluated a system for testing FORTRAN77 programs [15]. Such system uses symbolic execution for generating test cases for all the program paths satisfying the criterion “each loop in the program is iterated zero, one and two times” (ZOT path subset). Offutt and Seaman used symbolic execution for re-writing program internal variables as functions of the input values and then applying constraint-based testing for generating test cases [26]. Dillon, alone in [12] and together with Kemmerer and Harrison in [13], used symbolic execution to formally verify safety properties of Ada tasking programs. Coen-Porisini, DePaoli, Ghezzi and Mandrioli in [11] and Cimitile, De Lucia and Munro in [8] focused on symbolic execution based approaches for re-use issues. The former paper addresses specialization of generalized Ada software components via symbolic execution, while the latter paper proposes symbolic execution for reverse engineering and for qualifying reusable parts of existing software modules.

Lately, more work has been carried out in order to provide effective analysis techniques that can be applied in industrial environments.

Bush, Pincus and Sielaff [32] presented a tool, called PREfix, for finding dynamic programming errors that is based on a technique very similar to symbolic execution even though the latter is not explicitly mentioned. The tool is meant for identifying errors in memory handling, such as dangling pointers, memory leaks etc. PREfix analyzes programs written in C without any major restriction, since the aim is not at proving any property, but signaling some, although not all, real defects in dynamic memory management [32]. Conversely, our approach is limited to programs written in Safer-C, but can capture all program behaviors, thus dealing with all potential defects.

According to the authors, PREfix has been used on several large commercial products providing evidence of the effectiveness of analysis based on symbolic execution. It must be noticed that in order to be applicable in practice the proposed technique “selects a representative sample of achievable paths to simulate” [32]. Thus, this approach is similar to ours when dealing with loops.

Model checking represents an important alternative to symbolic execution for the analysis of programs. However, model checking requires a finite model of the program, which can be obtained either from software specifications or (seldom) directly from code. In the latter case, the program under analysis must be simplified to enable automatic analysis. Simplifications focus on the specific properties of interest. For example, San Pietro and Martena [33] present a technique for analyzing C code. The technique consists of encoding C code in Promela and then using Spin to carry out the analysis. The kind of analysis that one can carry out using this approach depends on the chosen abstraction that is the way in which encoding is defined. In [33] the focus is on identifying the so-called “may-alias” relation and aims at improving compiler optimization. Thus, encoding is meant for handling variables addresses rather than values.

7. CONCLUSIONS

This paper shows the usefulness of symbolic execution for the analysis of an important class of systems, namely software for safety critical applications written in Safer-C, and illustrates the possibility of automatically proving important safety properties through the analysis of TCAS. Additional experiments not reported in this paper for confidential reasons and observations based on the experimental results and information on avionic and nuclear application domains indicate that the approach can scale to industrial size software critical systems.

8. ACKNOWLEDGMENTS

The authors want to thank Lucio Casini, Maurizio Clerici Lorenzo Mera, and Sergio Silva for their invaluable help in the early phases of the project and in the implementation of the prototype. The authors would like to thank also Siemens for making publicly accessible the code of TCAS used to illustrate the approach.

9. REFERENCES

[1] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon D. Reese. “Model checking large software specifications,” *IEEE Transactions on Software Engineering*, 24(7):498-520, 1998.

[2] P. Asirelli, P. Degano, G. Levi, A. Martelli, U. Montanari, G. Pacini, F. Sirovich, and F. Turini. “An extensible environment for program development based on a symbolic interpreter,” *Proceedings of the 4th International*

Conference on Software Engineering, pages 251-263. IEEE Computer Society Press, 1979.

[3] Stephane Barbey and Didier Buchs. "Testing of Ada abstract data types using formal specifications," *Eurospace AdaEurope'94 Symposium Proceedings*, Copenhagen, 1994.

[4] R. S. Boyer, B. Elspas, and K. N. Levitt. "SELECT-A formal system for testing and debugging programs by symbolic execution," *ACM SIGPLAN Notices*, 10(6):234-245, June 1975.

[5] J. P. Bowen and M. G. Hinchey. "Ten commandments of formal methods," *IEEE Computer*, 28(4):56-63, 1995.

[6] David W. Binkley. "C++ in safety critical system," *Annals of Software Engineering*, 4:223-234, 1997.

[7] Lori A. Clarke. "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, 2(3):215-222, September 1976.

[8] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. "Qualifying reusable functions using symbolic execution," *Proceedings: Second Working Conference on Reverse Engineering*. IEEE Computer Society Press, 1995.

[9] P. David Coward. "Symbolic execution systems - a review," *Software Engineering Journal*, pages 229-239, 1988.

[10] Alberto Coen-Porisini, Flavio De Paoli, Carlo Ghezzi, and Dino Mandrioli. "Software specialization via symbolic execution," *IEEE Trans. Software Engineering*, SE-17(9):884-899, September 1991.

[11] A. Coen-Porisini and F. De Paoli. "SESADA: An environment supporting software specialization," *ESEC '91*, volume 550 of Lecture Notes in Computer Science, pages 266-289. Springer-Verlag, 1991.

[12] Laura K. Dillon. "Using symbolic execution for verification of Ada tasking programs," *ACM Transactions on Programming Languages and Systems*, 12(4):643-669,1990.

[13] Laura K. Dillon, Richard A. Kemmerer, and Linda J. Harrison. "An experience with two symbolic execution-based approaches to formal verification of Ada tasking programs," *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, 1988, pages 114-122, 1988.

[14] E. Allen Emerson and Edmund M. Clarke. "Using branching time temporal logic to synthesize synchronization skeletons," *Science of Computer Programming*, 2(3):241-266, December 1982.

[15] M. R. Girgis. "An experimental evaluation of a symbolic execution system," *IEEE Software Engineering Journal*, 7(4):285-290, July 1992.

[16] Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. "TRIO: A logic language for executable specifications of real-time systems," *The Journal of Systems and Software*, 12(2):107-121, May 1990.

[17] Les Hatton. *Safer C: Developing Software for High-integrity and Safety-critical Systems*. McGraw-Hill International Ltd., 1995.

[18] Mats P. E. Heimdahl. "Experiences and lessons from the analysis of TCAS II," *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 79-83, 1996.

[19] Richard A. Kemmerer and Steven T. Eckman. "UNISEX: A UNIX-based symbolic EXecutor for Pascal," Software - Practice and Experience, 15(5):439-458, May 1985.

[20] James C. King. "Symbolic execution and program testing," *Communications of the ACM*, 19(7):385-394, July 1976.

[21] Giovanni Denaro, "Using Symbolic Execution for Verifying Safety-Critical Systems", Tech. Rep., Polimi.2000.39, 2000

[22] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall software series. Prentice-Hall, Englewood Clis, NJ 07632, USA, 1978.

[23] Luqi and Joseph A. Goguen. "Formal methods: Promises and problems," *IEEE Software*, 14(1):73-85, 1997.

[24] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. "Requirements specification for process-control systems," *IEEE Transactions on Software Engineering*, 20(9):684-707, September 1994.

[25] Pascal Manoury and Marianne Simonot. "Automatizing termination proofs of recursively defined functions," *Theoretical Computer Science*, 135(2):319-343, 1994.

[26] A. Jeerson Out and E. Jason Seaman. "Using symbolic execution to aid automatic test data generation," *Compass '90: 5th Annual Conference on Computer Assurance*, pages 12-21, Gaithersburg, Maryland, 1990.

[27] C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. "On the automated generation of program test data," *IEEE Transactions on Software Engineering*, 2(4):293-300, 1976.

[28] RTCA. *Minimum operational performance standards for traffic alert and collision avoidance system (TCAS) airborne equipment consolidated edition*. Guideline DO-185, Radio Technical Commission for Aeronautics, September 1990.

[29] Jeffrey M. Voas. "Testing formal methods," *Proceedings of the First Workshop on the Foundations of Component-Based Systems*, , pages 240-245, Zurich, Switzerland, 1997.

[30] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co. ISBN 0-8053-1670-1. 1995

[31] Dotty web site, www.research.att.com/sw/tools/graphviz/

[32] William R. Bush, Jonathan D. Pincus and David J. Sielaff. "A static analyzer for finding dynamic programming errors," *Software Practice and Experience*, 30;775-802, 2000

[33] Vincenzo Martena and Pierluigi San Pietro. "Alias Analysis by means of a Model Checker," *10th International Conference on Compiler Construction (CC 01)*, 2001.

[34] M. Hutchins, H. Foster, T. Goradia and T. Ostrand. "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," *Proceedings of the 16th International Conference on Software Engineering*, pages 191-200, Sorrento, Italy, 1994

[35] C. Livadas, J. Lygeros and N. A. Lynch. "High-level modelling and analysis of TCAS," *Proceedings of the IEEE Real-Time Systems Symposium*, pages 115-125, 1999