

Euclide: A Constraint-Based Testing framework for critical C programs*

Arnaud Gotlieb

*INRIA Rennes - Bretagne Atlantique
Campus Beaulieu, 35042 Rennes Cedex, France
Arnaud.Gotlieb@irisa.fr*

Abstract

Euclide is a new Constraint-Based Testing tool for verifying safety-critical C programs. By using a mixture of symbolic and numerical analyses (namely static single assignment form, constraint propagation, integer linear relaxation and search-based test data generation), it addresses three distinct applications in a single framework: structural test data generation, counter-example generation and partial program proving. This paper presents the main capabilities of the tool and relates an experience we had when verifying safety properties for a well-known critical C component of the TCAS (Traffic Collision Avoidance System). Thanks to Euclide, we found an unrevealed counter-example to a given anti-collision property.

1 Introduction

Context. Safety-critical systems must be thoroughly verified before being exploited in commercial applications. In these systems, software is often considered as the weakest node of the chain and many efforts are deployed in order to reach a satisfactory testing level. A challenge in this area is the automation of the test data generation process for satisfying functional and structural testing requirements. For example, the standard document which currently governs the development and verification process of software in airborne system (DO-178B) requires the coverage of all the statements, all the decisions and MC/DC at the highest level of criticality and it is well-known that DO-178B structural coverage is a primary cost driver on avionics project.

In addition, the verification process of critical systems often requires the verification of safety properties, as people's life may rely on these properties. For airborne systems, some safety properties can be extracted from specifi-

cation documents that describe the so-called anti-collision theory regulating the controlled airspace. Checking these safety properties is mandatory and is usually performed by manual code reviews. Although they are widely used, most of the existing testing tools on the market are currently restricted to test coverage monitoring and measurements. Coverage monitoring answers to the question: what are the statements or branches covered by the test suite ? while coverage measurements answers to: how many statements or branches have been covered ? But these tools usually cannot find the test data that can execute a given statement, branch or path in the source code. In most industrial projects, the generation of structural test data is still performed manually and finding automatic methods for this problem remains a holly grail for most testers. Nevertheless, several experimental tools exist for C programs including INKA [20], PATHCRAWLER [32, 27], CUTE [30] or PEX [31], but none of them can also check safety properties or generate counter-examples that invalidate safety properties. Software model-checking tools such as SAVE [9], MAGIC [6], BLAST [23] or CBMC [8] have been proposed for checking properties over a piece of C code. But, these tools usually cannot generate a test suite that covers selected structural criteria. Finally, proof-based environments such as WHY/CADUCEUS [18] can automatically prove properties for C programs. But these tools cannot generate test cases or counter-examples.

Euclide. In this paper, we propose Euclide a constraint-based testing tool that features three main applications: structural test data generation, counter-example generation and partial program proving for critical C programs. The core algorithm of the tool takes as input a C program and a point to reach somewhere in the code. As a result, it outputs either a test datum that reaches the selected point, or an “unreachable” indication showing that the selected point is unreachable. Optionally, the tool takes as input additional safety properties that can be given under the form of pre/post conditions or assertions directly written in the

*This work is partially supported by ANR through the RNTL CAT and the CAVERN projects under the reference ANR-07-SESUR-003

code. In this case, Euclide can either prove that these properties or assertions are verified or find a counter-example when there is one. As these problems are undecidable in the general case, Euclide only provides a semi-correct procedure (when it terminates, it provides the right answer) for them. Hopefully, by restricting the subset of C that the tool can handle (no dynamic memory allocation, no recursion) these non-termination problems remain infrequent in practice. In addition, Euclide implements several procedures that combine atomic calls to the core algorithm. For example, by selecting appropriate points to reach in the source code, the tool can generate a complete test suite able to cover the `all_statements` or the `all_decisions` criteria.

Providing a tool able to deal with these three applications (structural test data generation, counter-example generation and partial program proving) in a single framework offers several advantages:

- For the developers having to maintain code they did not write, using a tool able to generate a failure-causing test datum that reaches a given point facilitates the debugging process. In fact, the test datum can easily be submitted as input to a symbolic debugger that will drive the computation towards the failure-causing point in the code ;
- In the unit testing phase, achieving high coverage with a test set that satisfies safety assertions improves the quality of the test selection process. The issued test set favorably enriches the set of tests to replay for future versions of the software (Regression Testing) ;
- For certification purposes, it is convenient to work only on a single certification product, namely the source code along with its annotations (assertions and pre/post conditions). Showing that the program satisfies all the required safety properties and that all parts of the program are executable and have been tested with respect to these properties is certainly a good way to convince a certification authority that the developed software is correct and reliable.

The underlying technology of Euclide is Constraint-Based Testing (CBT). Constraint-Based Testing is a two-stage process consisting first to generate a constraint system that corresponds to the testing objective we want to reach (for example, a selected point in a source code) and then, second to solve the constraint system by using well-recognized constraint programming techniques. CBT received considerable attention these latter years as constraint programming emerged as a worthwhile programming paradigm and solving techniques have been much improved.

Contributions. The originality of Euclide comes from its unique way of combining symbolic and numerical analyses such as static single assignment form, constraint prop-

agation, integer linear relaxation and search-based test data generation. Static single assignment form (SSA) relieves the tool from using costly and path-oriented symbolic evaluation techniques for generating the constraint system. Indeed, SSA allows considering several paths going through the selected point to reach at the same time. Thanks to constraint propagation, Euclide nicely handles non-linear operations such as multiplication between unknown variables, division, conditional and loop statement within C programs. Thanks to integer linear relaxation, the tool handles efficiently linear operations over integer variables. It also detects some unsatisfiable (possibly non-linear) constraint systems which were unbearable without this technique. Finally, thanks to its search-based test data generator that cooperatively labels the variables according to distinct heuristics, Euclide can generate test data or counter-examples in very efficient way. In this paper, we do not claim that Euclide is better than other more specialized test data generators or software model-checkers, but we show that this is its combination of symbolic and numerical techniques that offer the opportunity to get results outside of the scope of other tools. We exemplify this statement by our recent experience on using Euclide to prove safety properties for a well-known critical C component of the TCAS (Traffic Collision Avoidance System). Thanks to Euclide, we found an unrevealed counter-example to a given anti-collision property.

Plan of the paper. The rest of the paper is organized as follows: Section 2 reviews the main technologies used in Euclide. Section 3 presents its architecture and implementation while Section 4 relates our experience in using Euclide for generating test data and checking safety properties of a critical module of the TCAS. Section 5 presents the related work and finally, Section 6 concludes and draws some perspectives to this work.

2 Constraint generation and solving

2.1 Critical ISO/IEC compliant C programs

Our approach is dedicated to the testing of safety-critical (and ISO/IEC compliant) C programs. These programs share some characteristics such as being written in a restricted subset of the C language that excludes recursion and dynamic memory allocation among other things. The C language, as defined by the ISO/IEC standard [33], has also the considerable advantage to be well defined in terms of syntax and semantics, even if several operations have still an undefined behavior¹ or a behavior defined by the imple-

¹Exact behavior which arises is not specified by the standard, and exactly what will happen does not have to be documented by the C implementation.

mentation (in particular for floating-point computations).

Euclide handles a subset of C that includes integer and floating-point computations, pointers towards named locations, arrays of statically-allocated size, structures, function calls, bit-to-bit operations such as masks, all control structures (including loops) and almost all operators (34 over 42). But, it also has some restrictions: it does not deal accurately with unstructured statements such as `gotos`, unconstrained pointer arithmetic (such as using a physical address of a memory segment or adding two unrelated addresses as if they were integers), function pointers, functions with an unknown number of parameters, `volatiles`, `unions`, memory type casting (such as reading an integer as it was an address), library and external function calls (unavailable source code).

2.2 Generating Euclide programs

Euclide is based on a constraint model of C programs. This model, expressed in a dedicated language, is extracted from the source code by several transformational passes: parsing, normalization, pointer analysis, Static Single Assignment form and constraint model generation. In this section, we briefly review all these passes and discuss the main technologies used in Euclide to generate and solve constraint systems corresponding to testing objectives.

Parsing and normalization. This pass consists in building a symbol table and an abstract syntax tree for each compilation unit (preprocessed program). The symbol table keeps track of the type, scope, memory allocation class of each variable of the program while the abstract syntax tree captures the syntax of all the (non-declarative) statements of each function. Normalization is a process that permits to break complex statement into simpler ones. The rationale behind this pass is to simplify other passes by considering a smaller set of statements to analyze. Complex control structures are rewritten into simpler ones, function calls and arguments are isolated as well as side-effect expressions, multi-operators statements are decomposed. For example, thanks to the introduction of new temporary variables, a complex assignment statement such as `e=v1*v2*f()+v3;` is decomposed into `t0=f(); t1=v1*v2; t2=t1*t0; e=t2+v3;` because the function call has a higher priority than `*` and `+` and operands are evaluated from left to right. Note that such decomposition correctly handles multi-occurrences in C expressions. In the presence of floating-point computations, special attention must be paid to preserve the semantics. In particular, the decomposition requires that intermediate results of an operation conform to the type of storage of its operands². In the previous example, if `v1` and `v2` are of

²This property is not a requirement of IEEE-754 which is the standard that governs floating-point computations and consequently it is not always

single-format, then the temporary variable `t1` must also be single-format. For floating-point computations, this process has been extensively presented in a dedicated paper [4].

Points-to analysis. Euclide implements a *points-to analysis* that statically collects a set of variables that may be pointed by the pointers of the program and determines the set of memory locations that can be accessed through a dereference [21]. We selected a flow-sensitive points-to analysis previously introduced by *Emami et al.* [16] where each points-to relation is a triple: $pto(p, a, definite)$ or $pto(p, a, possible)$ where a denotes a variable pointed by p . In the former case, p points definitely to a on any control flow path that reaches the statement where the pointing relation has been computed. In the latter case, p may points to a only on some control flow paths. In a flow-sensitive analysis, the order on which the statements are executed is taken into account and the analysis is computed on each statement of the program.

Single Static Assignment form (SSA). A key-feature of Euclide concerns its use of the SSA form to avoid the usual costly path exploration phase of other tools. The SSA form is a semantics-preserving transformation of a program where each variable has a unique definition and every use of this variable is reached by the definition. Performing this transformation requires to rename uses and definitions of the variables. For example `i=i+1; j=j*i` is transformed into `i2=i1+1; j2=j1*i2`. At the junction nodes of the control structures, SSA introduces special assignments called ϕ -functions, to merge several definitions of the same variable: `v3 = ϕ (v1, v2)` assigns the value of `v1` in `v3` if the flow comes from the first branch of the decision, the value of `v2` otherwise. SSA provides special expressions to handle arrays: `access(a, k)` which evaluates to the k^{th} element of `a`, and `update(a0, j, v)` which evaluates to an array `a1` which has the same size and the same elements as `a0`, except for `j` where value is `v`. In the presence of pointers, special care must be taken when expliciting the possible hidden definitions of variables. We therefore defined a special form called Pointer SSA that captures hidden definitions through the usage of new special assignments exploiting the results of the flow-sensitive points-to analysis. The interested reader can consult [21] to get more details on our implementation of the so-called *Pointer SSA form* which accurately captures hidden definitions due to dereferences.

Constraint generation. Finally, statements under SSA form are converted into constraints in a dedicated intermediate language (not very inventively called Euclide). Relations, which are units of the language, can be either user-defined or primitive. User-defined relations correspond to functions defined in the C program while primitive relations

true. For example, on Intel's architectures extended formats are used by default to store intermediate results

are relations provided by the language itself. A relation can call other relations, allowing so to capture the C function calling mechanism. Examples of primitive relations include the ITE relation that models a conditional statement or the W relation modeling iterative statements. We will discuss the ITE relation in details in Sec.2.3 while details on W can be found in [13]. Evaluating an Euclide program yields either to true (= 1), or false (= 0) or suspend (= 0..1), corresponding to the truth value of the last evaluated relation of the program. Evaluation is incremental and relations can be awoken by additional relations. Fig.1 contains a simple Euclide program that implements a relational version of the *greatest common divisor* algorithm. Note that this Euclide program has been automatically generated from the imperative version of the gcd program.

```

rel GCD (X, Y, Z) iff          % true iff Z = gcd(X, Y)
{
  [X, Y, Z] in integers(unsigned, 32),
  X > 0, Y > 0, Z > 0,
  W(X > 0, [X, Y], [X4, Y2], [X5, Y3],
  {
    ITE (X < Y, [X, Y], [X2, Y1], [X3, Y2],
    {
      locals [X1],          % X1 is local to the current bloc
      X1 = X + Y,
      Y1 = X1 - Y,
      X2 = X1 - Y1,
    },
    { }                    % There is no Else_part
  ),
  X4 = X3 - Y2
  } )
}

```

Figure 1. The Euclide GCD program

On the request $GCD(X, Y, Z)$, X in $1..10$, Y in $10..20$, Z in $1..1000$, the constraint solvers of Euclide reduce the bounds of Z to $1..10$. Furthermore, if we add the relation $X=2*Y$, then Euclide automatically deduces that Z must be equal to Y to satisfy the request, which is a strong deduction usually outside the scope of other constraint solvers. In addition, the Euclide language includes a *reach* directive that is used to specify testing objectives. By inserting a *reach* directive in an Euclide program, the user unambiguously selects a location to reach within the source code and constrains the solutions of the program to satisfy this objective. For example, in the program of Fig.1, adding a *reach* directive in the Then-part of the conditional relation, permits to generate a test datum (values for X, Y) that reaches this part through an executable path. This *reach* directive is a key point of Euclide as it permits to specify various problems of reachability, including structural test data generation and counter-example generation.

An error-free semantics. The Euclide program captures an error-free relational semantics of its correspond-

ing C program. In other words, executions that yield errors such as dividing-by-zero or null pointer dereferencing are not considered when solutions of the Euclide program are sought. In fact, Euclide aims at finding functional faults and not runtime errors (i.e. errors that cause exceptions at runtime). Typically, a functional fault occurs in a program P when P returns the value 3 when 2 was expected. Detecting functional faults is crucial in the context of safety-critical program verification as people's life may rely on it. Note that functional faults cannot be detected by existing static analyzers as there is no oracle in these tools. By focusing on functional faults only, our constraint model is also simpler to implement and more efficient, as it does not have to maintain spurious erroneous states.

2.3 Constraint solving

The most innovative part of Euclide concerns its constraint solving engine. As said previously, Euclide implements constraint propagation, dynamic linear relaxation and search-based test data generation in order to satisfy testing objectives. A testing objective can be either 1) to generate a test datum that passes through a *reach* directive, 2) to generate a counter-example (i.e. a complete path that invalidates a property) or 3) to prove that a given property is satisfied by all executions of the program. Both former cases correspond to find a solution of a constraint system while the latter corresponds to show that a certain constraint system is unsatisfiable. In the latter case, the proof is only partial because all the domains on which the proof holds are bounded.

Constraint Propagation (CP). Roughly speaking, CP considers each constraint in isolation as a filter for the variation domain of the constraint variables. Once a reduction is performed on the domain of a variable, CP is awaking the other constraints that hold on this variable in order to propagate the reduction. Technically, CP is incrementally introducing constraints into a propagation queue. Then, an iterative algorithm is managing each constraint one by one into this queue by filtering the domains of their inconsistent values. When the variation domain of variables is too large, filtering algorithms consider usually only the bounds of the domains for efficiency reasons: a domain $D = \{v_1, v_2, \dots, v_{n-1}, v_n\}$ is approximated by the range $v_1..v_n$. When the domain of a variable is pruned then the algorithm reintroduces in the queue all the constraints that hold on this variable. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed. When selected in the propagation queue, each constraint is added into a constraint-store which memorizes all the considered constraints. The constraint-store is contradictory if the domain of at least one variable becomes empty. In this case the corresponding

testing objective is shown as being unsatisfiable.

Efficiency and completeness of CP. In the worst case, constraint propagation runs in $O(mn)$ where m denotes the number of constraints and n denotes the size of the largest domain. But constraint propagation alone does not guarantee satisfiability, as it just prunes the variation domains without looking at potential solutions. And it must be coupled with other mechanisms in order to find solutions or to show inconsistency³

Dynamic Linear Relaxations (DLRs). In [14], we introduced DLRs to relax dynamically all the constraints of an Euclide program, including the non-linear ones, within a Linear Programming framework. Linear Programming techniques such as the simplex procedure can solve huge instances of linear constraint systems very efficiently. Linear relaxation can be understood as a systematic way to over-approximate Euclide’s relations by linear constraints. We integrated linear relaxations within the constraint propagation process, yielding to an optimized cooperation scheme of the constraint solving process. For control structures (conditionals, loops) we proposed specific DLRs based on case-based reasoning and abstract interpretation techniques [13]. For example, the DLR of the ITE relation uses the following principles: given an ITE relation modeling a disjunction between two subpaths (Then-part and Else-part), first try to prove that one of the two disjuncts is unsatisfiable with the rest of the constraints and, thus, replace the overall disjunction by the other disjunct. Second, when this case-based reasoning fails, compute the union of both domains as in the following example: from the disjunctive constraint $X = Y \vee X = 5$ with domains $D_X = -1000..1000$, $D_Y = 0..1$, one can deduce that $D_X = 0..5$, $D_Y = 0..1$. In Euclide, we extended the union principle with linear relations. For example, considering $X = Y + 10 \vee X = Y - 10$ with domains $D_X = D_Y = 0..20$ we deduce that $-10 \leq X - Y \leq 10$ while the above reasoning over domains would not have deduce anything new on the domains.

Test Data Generation. CP and DLRs cannot guarantee satisfiability on their own as they both computes over-approximations of the sets of solutions. Hence, it is necessary to combine these processes with a labeling step in order to exhibit a solution (a test data satisfying the testing objective or a counter-example to a given property) or to demonstrate unsatisfiability (a partial proof of the property). Such a labelling step consists in exploring the input search space. One remarkable feature of modern labelling procedures is their ability to awake constraint propagation. Once a value a is assigned to a variable v , a constraint $v = a$ is added to the constraint system and awakes other constraints holding on v . Thanks to CP, the input search space is likely to be pruned before having to enumerate all the val-

³Proving a property over a piece of code in Constraint-Based Testing requires showing that a constraint system is unsatisfiable.

ues of the variables domain. In Euclide, we implemented and experimented several heuristics to choose the variable and the value to enumerate first. Finally, we depicted a labelling procedure that enchains several heuristics: *domain constraints*, *domain splitting*, *exhaustive search*, and *random choices*. *Domain constraints* consists in exploring subdomains of the input search space by iteratively increasing the size of the explored subdomains, while *domain splitting* consists in dividing the subdomains by propagating division constraints. For example, if $x \in 0..2^{32} - 1$ then domain splitting first adds the division constraint $x \in 0..2^{31} - 1$ which will be propagated throughout the constraint system and second it adds $x \in 2^{31}..2^{32} - 1$. *Exhaustive search* is the process that will enumerate all the values in the increasing or decreasing order of a given single dimension subdomain while *random choices* will pick up values at random within a domain. Thanks to these heuristics, search-based test data generation allows to find solutions in most cases. However, as the problem of finding solutions of a non-linear constraint system over finite domain is NP-hard [22], it may happen that the search fails in a reasonable amount of time. For these reasons, we implemented a parameterized timeout process to the search.

3 Architecture and implementation

Euclide features three main applications: structural test data generation, counter-example generation and partial program proving. The tool architecture shown in Fig.2 and its implementation were thought with these applications in mind.

3.1 Architecture

The tool takes a set of C files as input, optionally annotated by pre/post conditions and assertions (input column). For each C function of the files, an intra-procedural control flow graph is built and can be displayed through a graphical user interface (Control flow graph generator and CFGs component of the output column). In addition, an Euclide program is generated through the passes that have been presented above (parsing, normalization, points-to analysis, SSA form, constraint generation). Selecting either a node or a branch to reach yields to add a *reach* directive within the intermediate Euclide program (testing objectives of the input column). From there, constraint solving is launched according to some parameterization through an evaluator component. When a test data is generated, the flow is monitored either on the control flow graph or on a textual view of the Euclide program. The value of each individual input is shown and recorded when agreed by the user. Optionally, the linear relations that over-approximate each intermediate state of the analysis are printed within an interme-

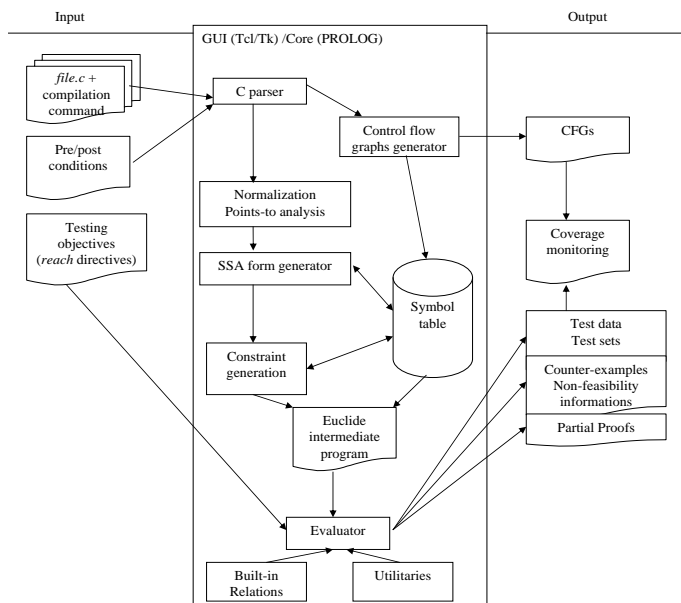


Figure 2. Euclide’s architecture

diate file. When the testing objective is unsatisfiable (non-feasible point or unsatisfiable property), then this is reported to the user. In addition, several automatic structural test data generation procedures are available such as generating a test set that covers all the executable statements or decisions. These procedures use several algorithms that add *reach* directives in appropriate locations.

3.2 Implementation

The Euclide’s implementation includes 9 internal components (inside the box of Fig.2) and two additional interface components. The tool is mainly developed in Prolog (~10 KLOC), C (~0.3 KLOC) and Tcl/Tk (~0.5 KLOC). The internal components include a backtrackable C parser written with the Definite Clause Grammar of Prolog, a SSA form generator based on the single-pass generator of Brandis and Mossenbock [5], an Euclide program generator and parser, a built-in relations library that implements most of the C operations (conditionals, loops, bit-to-bit operators, logical operators, function call operator, access/update, memory operations,...) and an utility component. The additional interface components implement the graphical user interface in Tcl/Tk and the batch mode in Prolog. Floating-point low-level representation and operations are implemented in C.

The evaluator component implements several constraint solvers that make use of the two following libraries: the *clpfd* library of Sicstus Prolog which implements a finite domains constraint solver ; and the *clpq* library that imple-

ments a linear programming solver based on simplex over the rationals. We made the two solvers cooperate by implementing our own constraint propagation queue and by building a dedicated constraint propagation solver.

4 Case study

Euclide is a Constraint-Based Testing tool dedicated to the validation of critical C programs and besides the traditional validation on academic examples, we wanted to evaluate the capabilities of Euclide on a real-world program. A typical (but small) example is the well-documented TCAS component of the Siemens suite. This suite was initially provided by Thomas Ostrand and its colleagues at Siemens Corporate Research Unit for an experimental study of the fault detection capabilities of coverage criteria [24]. It was then exploited by both Industry and Academia to evaluate testing strategies. Each component of the suite comes with a set of test cases and a set of mutants that exemplify typical faults. Recently, the suite was made publicly and freely available through the Software-artifact Infrastructure Repository [15].

TCAS (Traffic Alert and Collision Avoidance System) is an on-board aircraft conflict detection and resolution system embedded on all commercial aircrafts. The system is intended to alert the pilot to the presence of nearby aircraft that pose a mid-air collision threat and to propose maneuvers so as to resolve these potential conflicts. In cases of collision threats, the TCAS enters some levels of alertness. As shown on Fig.3, when an intruder aircraft enters a protected zone, the system issues a Traffic Advisory (TA) to inform the pilot of potential threat. In addition, TCAS estimates the time remaining until the two aircrafts reach the closest point of approach (CPA). If the danger of collision increases then a Resolution Advisory (RA) is issued, providing the pilot with a proposed maneuver that is likely to solve the conflict. The RAs issued by TCAS are currently restricted to the vertical plane only (either climb or descend) and their computation depends on time-to-go to CPA, range and altitude tracks of the intruder.

Implementation. The main component (*tcas.c*), extracted from the Repository is responsible of the Resolution Advisories issuance. It is made up of 173 lines of C code and contains nested conditionals, logical operators, type definitions, macros and function calls. Fig.4 shows the call graph of the program while Fig.5 shows the code of the highest-level function *Alt_sep_test* which computes the RAs. This function takes 14 global variables as input, including *Own_Tracked_Alt* the altitude of the TCAS equipped airplane, *Other_Tracked_Alt* the altitude of the “threat”, *Positive_RA_Alt_Thresh* an adequate separation threshold, *Up_Separation* the estimated separation altitude resulting from an upward maneu-

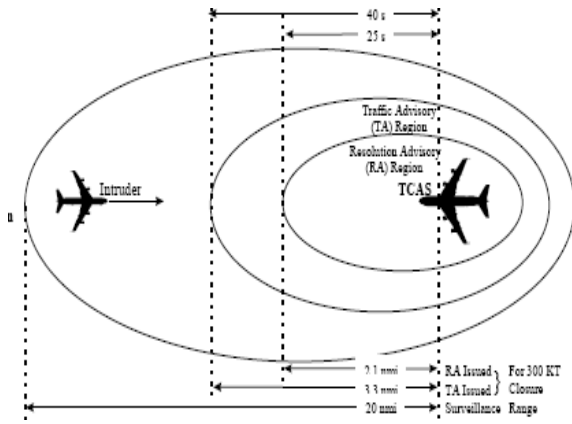


Figure 3. TCAS alarms

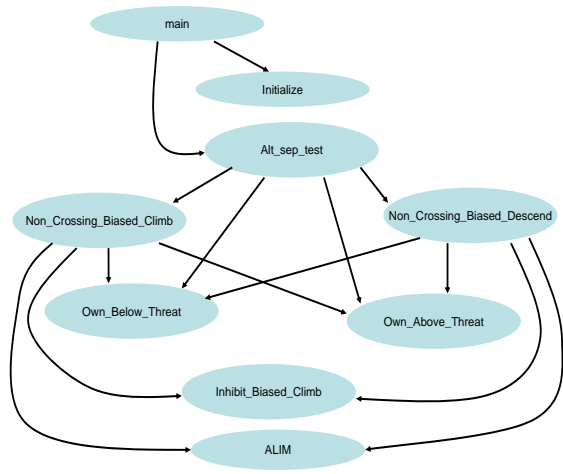


Figure 4. Call graph of `tcas.c`

ver and `Down_Separation` the estimated separation altitude resulting from a downward maneuver.

Interestingly, any TCAS implementation should be certified under level B of the DO-178B standard⁴. This has several implications w.r.t. the testing level required for certifying the TCAS. In particular, all the statements and decisions of the source code must be executed at least once during the testing process and any statement and decision must be shown as being executable, because non-executable elements do not trace to any software requirements and do not perform any required functionality.

Safety properties. In addition to these requirements, any TCAS implementation should verify safety properties that come from the aircraft anti-collision theory [28]. For the considered component, several properties referring to the possibility of issuing either an upward or a downward RA have been previously formalized [26, 9]. Tab.1 shows the five double properties extracted from [9]. For example, property P1b says that if an upward maneuver does not produce an adequate separation while a downward maneuver does, such as in Fig.6, then an upward RA should not be produced.

Results and analysis. We conducted several experiments on this program to evaluate the capabilities of Euclide to serve as an aid for certification purposes. Firstly, we evaluated structural test data generation for the coverage of the `all_decisions` criterion. On an Intel Core Duo 2.4GHz clocked PC with 2GB of RAM, Euclide generated a test set covering all the executable decisions of the `tcas` program in 16.9 seconds, including time spent garbage col-

⁴The standard classifies systems under 5 criticality levels: from the highest critical level A to the least critical E

```

int alt_sep_test()
{
1.  bool enabled, tcas_equipped, intent_not_known;
2.  bool need_upward_RA, need_downward_RA;
3.  int alt_sep;

4.  enabled = HighConfidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
5.  tcas_equipped = Other_Capability == TCAS_TA;
6.  intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;

7.  alt_sep = UNRESOLVED;

8.  if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
9.  {
10.     need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
11.     need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
12.     if (need_upward_RA && need_downward_RA)
13.     /* unreachable: requires Own_Below_Threat and Own_Above_Threat
14.     to both be true */
15.     alt_sep = UNRESOLVED;
16.     else if (need_upward_RA)
17.     alt_sep = UPWARD_RA;
18.     else if (need_downward_RA)
19.     alt_sep = DOWNWARD_RA;
20.     else alt_sep = UNRESOLVED;
21. }
22. return alt_sep;
}

```

Figure 5. Function `alt_sep_test` from `tcas.c`

lecting, stack shifting, or in system calls. It also showed that the decision of line 11-12 of Fig.5 was non executable in less than 0.2 second. Secondly, we evaluate partial program proving on the safety properties of Tab.1. Results are shown in Tab.2. Finding counter-examples to safety properties on a TCAS implementation could appear as being dramatic. But, the reader should be warned that this TCAS implementation probably corresponds to a preliminary version and that it has probably never been used in operational conditions.

Surprisingly, we found that properties P2B, P3A and P5B were not proved w.r.t. the implementation and, thanks to Euclide, we exhibited verified counter-examples. These counter-examples satisfy the preconditions but invalidate the postconditions of the properties when submitted to the

Table 1. Safety properties for tcas.c

Num.	Property	Explanation	Specifications
P1a	Safe advisory selection	An downward RA is never issued when an downward maneuver does not produce an adequate separation	assumes $Up_Separation \geq Positive_RA_Alt_Tresh$ & $Down_Separation < Positive_RA_Alt_Tresh$; ensures $result \neq need_Downward_RA$;
P1b	Safe advisory selection	An upward RA is never issued when an upward maneuver does not produce an adequate separation	assumes $Up_Separation < Positive_RA_Alt_Tresh$ & $Down_Separation \geq Positive_RA_Alt_Tresh$; ensures $result \neq need_Upward_RA$;
P2a	Best advisory selection	A downward RA is never issued when neither climb or descend maneuvers produce adequate separation and a downward maneuver produces less separation	assumes $Up_Separation < Positive_RA_Alt_Tresh$ & $Down_Separation < Positive_RA_Alt_Tresh$ & $Down_Separation < Up_Separation$; ensures $result \neq need_Downward_RA$;
P2b	Best advisory selection	An upward RA is never issued when neither climb or descend maneuvers produce adequate separation and an upward maneuver produces less separation	assumes $Up_Separation < Positive_RA_Alt_Tresh$ & $Down_Separation < Positive_RA_Alt_Tresh$ & $Down_Separation > Up_Separation$; ensures $result \neq need_Upward_RA$;
P3a	Avoid unnecessary crossing	A crossing RA is never issued when both climb or descend maneuvers produce adequate separation	assumes $Up_Separation \geq Positive_RA_Alt_Tresh$ & $Down_Separation \geq Positive_RA_Alt_Tresh$ & $Own_Tracked_Alt > Other_Tracked_Alt$; ensures $result \neq need_Downward_RA$;
P3b	Avoid unnecessary crossing	A crossing RA is never issued when both climb or descend maneuvers produce adequate separation	assumes $Up_Separation \geq Positive_RA_Alt_Tresh$ & $Down_Separation \geq Positive_RA_Alt_Tresh$ & $Own_Tracked_Alt < Other_Tracked_Alt$; ensures $result \neq need_Upward_RA$;
P4a	No crossing advisory selection	A crossing RA is never issued	assumes $Own_Tracked_Alt > Other_Tracked_Alt$; ensures $result \neq need_Downward_RA$;
P4b	No crossing advisory selection	A crossing RA is never issued	assumes $Own_Tracked_Alt < Other_Tracked_Alt$; ensures $result \neq need_Upward_RA$;
P5a	Optimal advisory selection	The RA that produces less separation is never issued	assumes $Down_Separation < Up_Separation$; ensures $result \neq need_Downward_RA$;
P5b	Optimal advisory selection	The RA that produces less separation is never issued	assumes $Down_Separation > Up_Separation$; ensures $result \neq need_Upward_RA$;

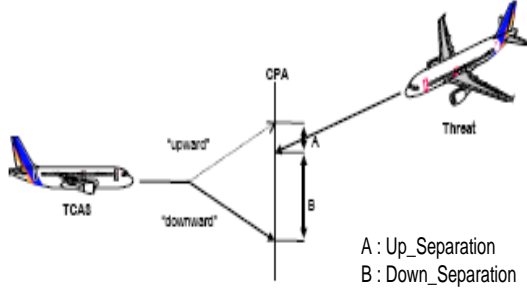


Figure 6. Resolution Advisories

Table 2. Verification of safety properties

Num	Results	Time (sec.)	Mem. (MB)
P1a	Property proved	0.7	4.6
P1b	Property proved	0.7	4.6
P2a	Property proved	0.6	4.6
P2b	Counter-example found	0.7	4.6
P3a	Counter-example found	5.4	6.3
P3b	Property proved	1.2	4.6
P4a	Counter-example found	6.8	6.9
P4b	Counter-example found	2.7	5.9
P5a	Property proved	0.6	4.6
P5b	Counter-example found	1.0	4.6

implementation. All the material of these experiments, including the test data corresponding to counter-examples, is available online⁵. In addition, counter-examples to properties P5B were not reported in the literature [9, 8, 6]. Moreover, we got these counter-examples and proofs very quickly (all the counter-examples and proofs are generated in less than 20s on our standard machine) which is encouraging for a future comparison with other more dedicated tools.

5 Related work

Euclide addresses three distinct applications for C programs, namely test data generation for structural testing, counter-example generation and partial program proving. We are not aware of any tool having the same capabilities for C programs. However, many tools exist for one or two of these tasks.

Partial program proving. These tools usually apply Floyd-Hoare logic or Dijkstra’s weakest preconditions calculus to the formal verification of the so-called *verification conditions (VCs)* extracted from programs and annotations.

⁵www.irisa.fr/lande/gotlieb/resources.html

Caduceus [18], which was pioneering deductive verification of C programs, concurrently launches several interactive proof assistants or theorem provers to prove a given assertion. Spec# [25] infers loops invariants by using abstract interpretation and infers VCs, even in the presence of dynamic allocated objects on the heap. More recently, Dash [2] exploits lightweight symbolic execution techniques and a single call to a theorem prover to show that a given property is satisfied on several paths of the implementation. Euclide implements its own automated constraint solving procedures while Caduceus, Spec# and Dash exploit existing interactive proof assistants and automated theorem provers. As a result, Euclide deals more accurately with floating-point computations [4] and more efficiently with integer-based computations as it supposes every integer variable to belong to a finite domain and implements its own dedicated constraint techniques. But Euclide is also harder to develop and is less general because its proofs are only valid for bounded integer variables.

Automatic test data generators. The test data generator Godzilla was proposed very early [12] for Fortran programs in the context of mutation testing. In a subsequent paper, the *dynamic domain reduction procedure* was developed to

enrich the constraint solving capabilities of this approach [29]. This procedure mimics the constraint propagation step described in Sec.2.3. Constraint propagation is an old idea that dates back to the beginning of the seventies and its use has been proposed very early for test data generation [3]. InKa [20] was a pioneer in the use of *Constraint Logic Programming* for generating test data for C programs. It was able to generate test case for programs containing dynamic allocated structures as its memory model was rich enough [7]. Euclide can be seen as a successor of InKa as it shares many technical features with it (both are based on SSA and Constraint Propagation). However, several distinct choices have been made for efficiency reasons. Using some a priori restrictions (no dynamic memory allocation, no recursion), the Euclide's memory model is simpler and permits to deal more efficiently with integer computations. PathCrawler [32], Dart [19] and CUTE [30] are three modern path-oriented structural test data generators. These three tools rely on path selection, symbolic execution and concolic execution. On the contrary, Euclide rely on statement or decision selection (goal-oriented approach [17]), static single assignment form and a mixture of symbolic and numeric constraint solving procedures. The treatment of loops is very different: while these path-based tools unfold the control flow structure of loops to select a path, Euclide handles a loop structure as a whole. By abstracting the behavior of the loop structure (as done with abstract interpretation techniques), the tool can deduce properties outside the scope of any path-based test data generator. For example, Euclide can (sometimes) determine that a given point, positioned after a loop structure, is unreachable. This is impossible with a path-based tool that will enumerate indefinitely all the paths through the loop structure. Recently, Bardin and Herrmann performed a remarkable work on the OSMOSE tool which aims at covering all executable paths of a binary program by using constraint solving techniques [1]. By addressing low-level binary-code, they opened a door that we could benefit from for improving the coverage of our own tool. In fact, C code often presents low-level features that we cannot currently deal with (unconstrained pointer arithmetic, dynamic jumps, ...).

Counter-example generation. Software model-checkers such as Save [9], Blast [23], Magic [6] or Cbmc [8] permit to find counter-examples to temporal properties over C programs. These tools explore the paths of a bounded model of programs in order to find a counter-example path to the property. Some of them exploit *predicate abstraction* and counter-example refinement to boost the exploration. Euclide contrasts with SAT-based or SMT-based model-checkers as it does not abstract the program and does not generate spurious counter-example paths. In particular it builds a high-level constraint model of C program by capturing an error-free semantics without considering a

boolean abstraction of the program structure. Our approach has more similarities with the CPBPV tool of Collavizza, Rueher and Van Hentenryck [10, 11] that call several constraint solvers in sequence. Recently, its authors showed that CPBPV could outperform the best model-checkers on several classical benchmarks. As Euclide, CPBPV tool is based on deductive constraint programming techniques. However, research and experimental work remains to confirm these results obtained on a small set of academic programs.

6 Conclusion

In this paper, we introduced Euclide, a Constraint-based testing platform for C programs. The capabilities of the tool include structural test data generation, counter-example generation and partial program proving and it combines numerical and symbolic techniques, namely SSA, constraint propagation, dynamic linear relaxations and search-based test data generation. Euclide handles a large subset of C, even if some a priori restrictions have been done (no recursion, no dynamic allocation). The tool was applied to the verification of a critical component of the TCAS, which yields an unrevealed counter-example to a safety property. However, the tool could be improved in many ways. Function calls are currently handled by inlining which prevents Euclide from using efficient modular constraint-based analysis. Summaries of function calls could be exploited in the test data generation process. Search-based test data generation currently exploits only complete heuristics that explore the whole search space in the worst case. We could also exploit local search techniques that are sometimes very efficient. Other similar improvements are possible and requires additional research works in order to increase the efficiency of the tool.

7 Acknowledgment

Much of the choices and decisions taken within the development of Euclide were discussed with other people, and I am indebted to all of them. I would like to thank especially Tristan Denmat who investigated the role of Abstract Interpretation in the linear relaxation techniques we employed. Many thanks also to Bernard Botella, Benjamin Cama, Florence Charretier, Nadjib Lazaar, Bruno Marre, Matthieu Petit and Pierre Rousseau.

References

- [1] Sebastien Bardin and Philippe Herrmann. Structural testing of executables. In *11th Int. Conf. on Software Testing, Verification and Validation (ICST'08)*, pages 22–31, 2008.

- [2] N. Beckman, A. Nori, S. Rajamani, and R. Simmons. Proofs from tests. In *Proc. of ISSTA'08*, pages 3–14, 2008.
- [3] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. Miller. SMOTL - a system to construct samples for data processing program debugging. *IEEE Transactions on Software Engineering*, 5(1):60–66, January 1979.
- [4] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability Journal*, 16(2):pp 97–121, June 2006.
- [5] M.M. Brandis and H. Mössenböck. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Transactions on Programming Language and Systems*, 16(6):1684–1698, Nov. 1994.
- [6] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.
- [7] F. Charretier, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. In *TAIC-PART (Testing: Academic and Industrial Conference)*, Windsor, UK, Sep. 2007.
- [8] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proc. of ASP-DAC'03*, pages 308–311, Jan. 2003.
- [9] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'01)*, pages 142–150, Vienna, Austria, September 2001. ACM.
- [10] H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 182–196, 2006.
- [11] H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpy: A constraint-programming framework for bounded program verification. In *Proc. of CP2008*, LNCS 5202, pages 327–341, 2008.
- [12] R.A. DeMillo and J.A. Offut. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [13] T. Denmat, A. Gotlieb, and M. Ducasse. An abstract interpretation based combinator for modeling while loops in constraint programming. In *Proceedings of Principles and Practices of Constraint Programming (CP'07)*, Springer Verlag, LNCS 4741, pages 241–255, Providence, USA, Sep. 2007.
- [14] T. Denmat, A. Gotlieb, and M. Ducasse. Improving constraint-based testing with dynamic linear relaxations. In *18th IEEE International Symposium on Software Reliability Engineering (ISSRE' 2007)*, Trollhättan, Sweden, Nov. 2007.
- [15] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.
- [16] E. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of PLDI'94*, Orlando, FL, Jun. 1994.
- [17] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering Methodology*, 5(1):63–86, Jan. 1996.
- [18] J.C. Filliâtre and C. Marché. Multi-prover verification of c programs. In *6th Int. Conf. on Formal Engineering Methods (ICFEM'04)*, pages 15–29, Nov. 2004.
- [19] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proc. of PLDI'05*, pages 213–223, 2005.
- [20] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. of ISSTA'98*, pages 53–62, 1998.
- [21] A. Gotlieb, T. Denmat, and B. Botella. Goal-oriented test data generation for pointer programs. *Information and Software Technology*, 49(9-10):1030–1044, Sep. 2007.
- [22] P.V. Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37:139–164, 1998.
- [23] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proc. of 10th Workshop on Model Checking of Software (SPIN)*, pages 235–239, 2003.
- [24] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of ICSE '94*, pages 191–200, 1994.
- [25] Rustan Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
- [26] C. Livadas, J. Lygeros, and N.A. Lynch. High-level modeling and analysis of TCAS. In *IEEE Real-Time Systems Symposium*, pages 115–125, 1999.
- [27] Patricia Mouy, Bruno Marre, Nicky Williams, and Pascale Le Gall. Generation of all-paths unit test with function calls. In *First International Conference on Software Testing, Verification, and Validation, (ICST'08)*, pages 32–41, 2008.
- [28] U.S. Department of transportation Federal Aviation Administration. *Introduction to TCAS II - version 7*, Nov. 2000.
- [29] J.A. Offut, Z. Jin, and Pan J. The dynamic domain reduction procedure for test data generation. *Software-Practice and Experience*, 29(2):167–193, 1999.
- [30] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proc. of ESEC/FSE-13*, pages 263–272. ACM Press, 2005.
- [31] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proc. of ESEC/FSE-13*, pages 253–262. ACM Press, 2005.
- [32] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *In Proc. Dependable Computing - EDCC'05*, pages 281–292, 2005.
- [33] www.open-std.org/JTC1/SC22/WG14/www/standards. *ISO/IEC 9899 - Programming languages - C*, 1999.