Contribution to the Verification of Complex Graphical Equations Using Z3

Ali BEN TAOUET

ali.ben-taouet@alumni.enac.fr
Université de Toulouse - ENAC, LII
7 av. Edouard Belin, 31055 Toulouse, France

Abstract

In this paper, we focus on solving nonlinear SMT (Satisfiability Modulo Theories) problems involving complex logical and arithmetic constraints using Z3 solver. These equations are the output of an algorithm developed in the framework of Smala/Djnn, which validates graphically oriented requirements based on performing deductive verification. By solving these equations that originate from a source code, it becomes possible to conclude whether the properties regarding its graphical requirements are fulfilled. When a program fails to function as expected, determining the inputs for which the program produces undesirable outcomes could aid developers in the debugging journey.

1 Introduction

This research is part of a four-month initiation-to-research project, which was conducted by a second year engineering student at the French Civil Aviation University (ENAC). The paper focuses on employing Z3 API in Python to solve 8 equations [1]. Each of them is modeling one graphical property of an elementary component of the TCAS integrated on IVSI and implemented in Smala (Figure 1).

Further details, including the equations, script and supplementary materials, can be accessed at: `https://github.com/benwhoiam/pir`.

While we were provided 8 equations to be treated, this paper presents examples of equations E1 (a satisfied linear equation), E3 (an unsatisfied linear equation), E5 (an unsatisfied non-linear equation) and E8 (an example of the potential failure of the adopted analytical approach).

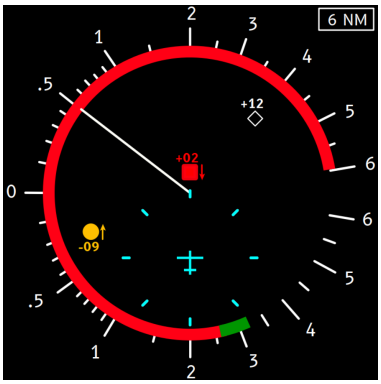

Figure 1: TCAS integrated on IVSI and implemented in Smala (from [2]).

The following sections summarize the purpose of each selected equation:

- E1 makes sure that the own aircraft symbol is depicted in the middle of the screen, displayed at 1/3 of the screen height and the color could be white or cyan to differentiate with other aircraft.
- E3 makes sure that the threat aircraft are depicted as red solid squares in the IVSI area so as to easily identify them.

- E5 checks the display of proximate aircraft as solid diamonds in white or cyan, different from their own aircraft and well positioned.
- E8 ensures the correctness of the red arc which depicts the vertical speeds to avoid (Figure 6).

Those provided equations are collectively ensuring the compliance with TCAS safety and usability standards.

2 Problematic

The problematic we address in this paper is assisting developers in identifying bugs in their code. Traditionally, verifying properties of code that does not involve interactive or graphical elements relied on techniques such as static analysis [3], symbolic execution [4], or formal verification [5]. These techniques are effective when it comes to analysing the variables, logical structure, and the relationships between inputs and outputs of linear, textual code to ensure that certain properties are satisfied. However, these approaches are still not widely used for systems with complex graphical or interactive components [1].

We take as input equations derived using the weakest preconditions approach to verify properties [6]. In short, the weakest precondition is a logical condition that must be true when applied on the inputs to guarantee that a property is satisfied for the outputs. For example, in this simple problem statement for calculating distance between 2 points:

- Input: X, Y
- Computation: $D = X - Y$
- Desired Property: $D \geq 0$

The weakest precondition would be $X \geq Y$, since this condition on X and Y ensures that the property $D \geq 0$ is true. This approach is based on rewriting the equations to determine the constraints required for correctness.

However, real-world systems often introduce high complexity. For example, some equations generated to verify graphical elements of Traffic Alert and Collision Avoidance System, such as E5 that verifies the display of proximate aircraft as solid diamonds in white or cyan, contain more that 3,000 occurrences of relational expressions involving equality and inequality. This demands an efficient method to solve the generated equations and identify input conditions.

Hence, this work focuses on leveraging Z3 API in Python in an attempt to efficiently handle simplifying and solving the equations [7]. By analysing the solver's output, we aim to determine the conditions on the inputs that ensure that the graphical elements satisfy the required properties. If such conditions are identified, the next step would be isolating and studying the relevant data to assist in debugging the system and addressing any underlying issues.

3 State of the art

The most direct methods to realize the solution of the equations and guarantee the correctness of the solution are Brute-force techniques. The results of a brute force are correct (barring numerical approximations) [8], but quite computationally prohibitive. For our TCAS analysis [2], considering a limited set of input parameters:

- current_range: $[6, 40]$
- bearing: $[-180, 180]$ degrees with a step of 0.2
- range: $[0, 128]$ with a step of $1/12$
- level: $\{0, 1, 2, 3, 4\}$

A brute-force approach would require approximately 470,016,000 simulations, which is over 100 days of computation on an average laptop. This argues for the impracticability of brute-force for complex systems.

Hence, the use of Constraint solvers such as Satisfiability Modulo Theories (SMT) solvers. These solvers present a much more viable alternative given their capability to determine the satisfiability of logical formulas involving constraints over different theories, including arithmetic, arrays, and bit-vectors.

Previous applications of SMT solvers include being used in software verification [9], Mapping and Scheduling on Multi-core Processors [10], and validating models for AI problems [11] and more.

Previous work has explored the use of SMT solvers in conjunction with weakest preconditions for verifying software properties [12].These techniques have shown considerable promise in the context of object-oriented programs and others.

Z3 stands out as due to its versatility in handling various theories, it is able to solve complex problems efficiently due to its high performance, and its ease of integration with languages like Python, which enhances usability for developers and researchers. Z3 is widely used for solving logical formulas and constraints in SMT, which extend Boolean logic with theories like arithmetic and arrays. It works by encoding problems as logical formulas and systematically exploring the solution space using advanced decision procedures. However, Z3 struggles with trigonometric functions, as it lacks native support for solving nonlinear equations involving these functions, requiring workarounds or alternative solvers for such cases.

4 Contribution

In this work, we use Z3 API in Python 3.12.8 address the verification of graphical properties associated with the TCAS system implemented in Smala. The subsequent subsections detail the methodology adopted. Firstly we introduce the equation conversion process, outlining how infix Boolean logic equations are transformed into prefix functional logic expressions. Secondly, we delve into two approaches for solving these equations: a numerical method leveraging discretisation and approximations, and an analytical approach utilising Z3's simplification capabilities.

4.1 Equation conversion

In order to solve the 8 equations provided using the Z3 API in Python, it is necessary to transform them from infix Boolean logic to prefix functional logic expressions, as this is the syntax required by Python and the Z3 solver for proper parsing and evaluation. To accomplish this, we developed a robust approach for performing the conversion.

The solution we propose employs Python's Abstract Syntax Trees (AST) to parse and systematically preform the conversion. The method involves analysing the structure of an expression, identifying its components (operators, operands, and logical relationships), and translating them into a hierarchical format that adheres to desired prefix logic. This approach handles a wide variety of operations, including arithmetic (e.g., $+, -, *$), Boolean logic (e.g., AND, OR), unary operations (e.g., NOT), and comparisons (e.g., $<, =, \leq$).
The implementation consists of the following steps:

1. Preprocessing the input expression and adapting it with python syntax, as well as replacing certain symbols (e.g., '&' with 'and', '|' with 'or'). This helps in preserving the right hierarchy when reconstructing the prefix expression. Without this replacement, the statement '$x < 90$ & $true$' will be falsely converted to '$x < And(90, true)$' instead of '$And((x < 90), true)$' due to the hierarchical positioning of the binary operator '&' ($ast.BitAnd$) compared to boolean 'And' ($ast.And$).

2. Parsing using the *ast.parse* function, which converts the expression into an abstract syntax tree.

3. Recursively calling a custom function that applies a transformation rule that matches the desired mapping from infix to prefix. Recursive calling allows the navigation of the entire tree.

For example, consider the simple input '$A\&(B \mid\sim true)$', the preprocessing will transform it into 'A and (B or $\sim$ True)' which will then be parsed to form the tree in Figure 2.

The recursive function takes the tree in input parameters and navigates it constructing the desired prefix functional logic expressions: $And(A, Or(B, Not(True)))$.

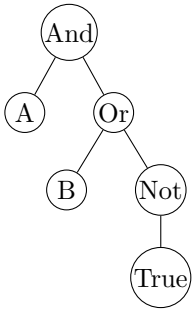

Figure 2: Example of the logical expression tree.

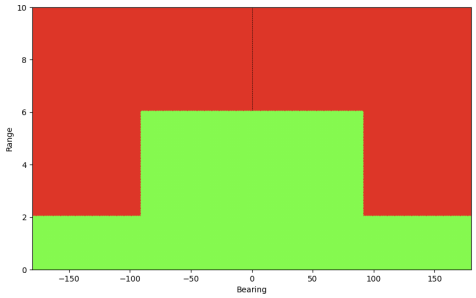

Figure 3: Result of checking E8 with level = 4 and current_range = 6.

4.2 Solving the equation using Z3

4.2.1 Numerical approach

In this section we will attempt to solve and 2d visualise the solutions.

A. Linear equations:

In the case of linear equations, this is a straightforward operation. Z3 can find possible solutions if they exist. If there are no solutions, meaning that the properties regarding its graphical requirements are fulfilled, Z3 responds with 'unsat' to the negation of the equation as it is the case with E1. However, if there is a bug as it is the case for E8 (Figure 3), Z3 may return many possible solutions given that both *range* and *bearing* of type 'Float', which can be computationally challenging even for a well optimised and advanced solver. To mitigate this, we discretised the bearing and range values by defining specific steps, limiting the search space and reducing execution time. While these steps differ from the required steps (1/12 for range and 1/5 for bearing), they help in reducing execution time while maintaining sufficient precision for the 2D graph.

B. Non-linear equations:

Z3 lacks a native support for solving non-linear equations. Our equation only contain sine, cosine and square root functions. Hence, our approximations only treats these function.

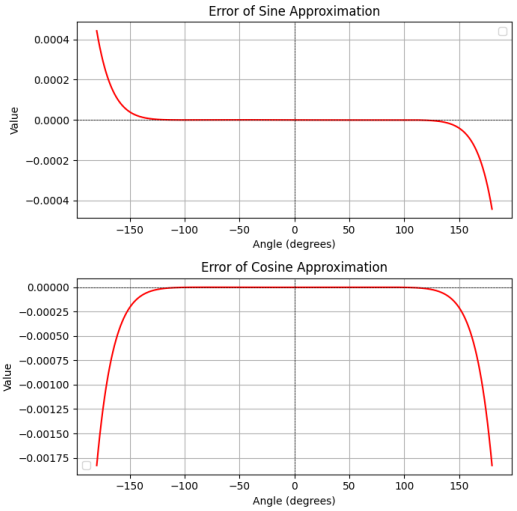

Figure 4: Error of the 11th-degree Taylor approximation of Sine and Cosine.

For sine and cosine, given that bearing is defined in the domain $[-180, 180]$ and that our equations calculate uniquely sin(bearing) and cos(bearing), the 11th-degree Taylor approximation is sufficient (Figure 4).

$$\sin(x) \approx x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} - \frac{x^{11}}{39916800}$$

$$\cos(x) \approx 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40320} - \frac{x^{10}}{362880}$$

For square root function, given the nature of our equations, the square root was always used in this form:

$$\sqrt{A} < B \text{ or } \sqrt{A} \leq B$$

Hence, performing a manual intervention by squaring the left term ($B \rightarrow B^2$) and redefining the square root (sqrt) in our script as the identity function was sufficient. After preforming the necessary modifications, the calculation of the approximated solutions is now possible(Figure 5).

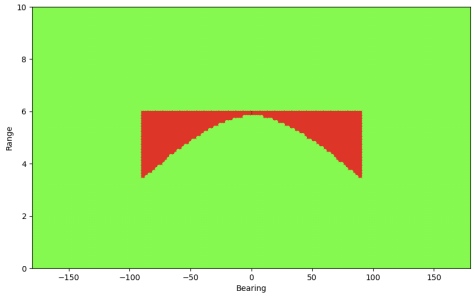

Figure 5: Result of checking E3 with level = 4 and current_range = 6.

4.2.2 Analytical approach

In this approach, we utilize Z3's *simplify* function to analyse the equations. A notable aspect of this methodology is its treatment of trigonometric functions such as sine and cosine. Within the Z3 framework, these functions do not possess inherent mathematical definitions. Instead, they serve as symbolic representations. This abstraction allows for a direct simplification of equations without delving into the specific values or approximations of these trigonometric expressions. When Taylor series approximations are employed to model sine and cosine functions, simplifications become more computationally intensive and result in less readable outputs. Leveraging this simplification approach, along with a similar trick here as in 4.2.1 to handle the square root function effectively, we can efficiently deduce the values of variables such as *current_range* and *level*. Unlike numerical methods that are computationally limited, the analytical approach directly outputs equations. This enables us to extend the analysis beyond two dimensions, offering greater flexibility and efficiency in exploring multi-dimensional parameter spaces.

```
(((((~(false) | ~(true) | (sqrt((400 - (60 + ((400 + ((sin((((bearing) *
(3.14159)) / (180)))) * (range)) * (((435) / (current_range)))) - (35))))
^2 + (400 - (47 + ((537 - (((cos((((bearing) * (3.14159)) / (180)))) *
(range)) * (((435) / (current_range))))) - (52))))^2) < 300 & sqrt((400 -
(54 + ((400 + ((sin((((bearing) * (3.14159)) / (180)))) * (range)) *
(((435) / (current_range)))) - (35))))^2 + (400 - (47 + ((537 - (((cos
((((bearing) * (3.14159)) / (180)))) * (range)) * (((435) /
(current_range))))) - (52))))^2) < 300 & sqrt((400 - (60 + ((400 + ((sin
((((bearing) * (3.14159)) / (180)))) * (range)) * (((435) /
(current_range)))) - (35))))^2 + (400 - (50 + ((537 - ((((cos((((bearin...
```

Figure 6: A snippet of E8 equation.

We provided the script with the equation E8 as input (Figure 6), fixing level to 4 and current_range to 6. It returned the following simplified logical expression:

$$\text{And}(\text{Range} \leq 6, \text{Or}(\text{Range} \leq 2, |\text{bearing}| \leq 90))$$

This output demonstrates the script's capability to reduce the original equation to a concise and interpretable form. We can also visually verify that this output is correct, given that it does not contradict the numerical solution (Figure 3). However, that is not the case for E8, as the resulting set of equations is difficult to be concluded from the graphical 2d representation (Figure 7).

```
Simplified readable     :
Or(Not(And(Range <= 6,
           Or(Range <= 2,|bearing|<=90))),
   And(Not(90000 <=(13 + -β*Range*sin(bearing))**2
+(-125 + β*Range*cos(bearing))**2),
       Not(90000 <=(-12 + -β*Range*sin(bearing))**2
 +(-125 + β*Range*cos(bearing))**2),
       Not(90000 <=(13 + -β*Range*sin(bearing))**2
+(-150 + β*Range*cos(bearing))**2),
       Not(90000 <=(-12 + -β*Range*sin(bearing))**2
 +(-150 + β*Range*cos(bearing))**2)))

Avec β = 145/2
```

Figure 7: Result of simplifying E3 with level = 4 and current_range = 6.

To understand the result, we manually simplified the equations and graphed them in a 2D plot (Figure 8). The highlighted terms represent the colored zones, while the colored terms are represented by the line graphs in Figure 8.

Or
(
Range >= 6,
And(Range >= 2, |bearing|>=90),
And(
((13 - α x sin(bearing))^2 +
(-125 + α x cos(bearing))^2 < 90000),
((-12 - α x sin(bearing))^2 +
(-125 + α x cos(bearing))^2 < 90000),
((13 - α x sin(bearing))^2 +
(-150 + α x cos(bearing)^2 < 90000),
((-12 - α x sin(bearing))^2 +
(-150 + α x cos(bearing))^2 < 90000)
)
)

Avec α = 145/2 * Range

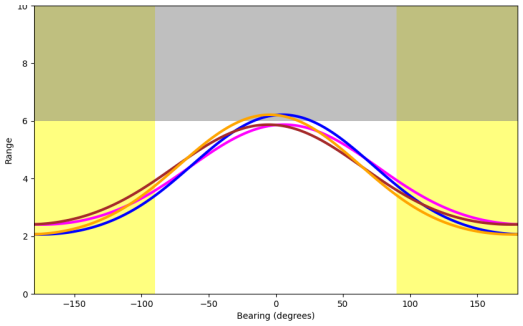

Figure 8: Visualisation of simplifying E3 with level = 4 and current_range = 6.

Figure 8 also serves as a visual way to verify and mutually compare both the first (Figure 5) and second approaches. This figure reveals that multiple conditions on the input led to the error, rather than a single factor. This insight was not apparent when the solutions were obtained numerically, highlighting the value of the analytical approach in identifying the underlying causes of the error.

5 Conclusion and Discussion

In this paper, we have presented a method through which infix boolean logic can be converted to prefix functional logic expressions. This conversion is important for using the Z3 API in Python to solve the equations in question. The conversion process includes preprocessing the input expression, converting it to Abstract Syntax Trees (AST) and the recursively transforming it to the required prefix logic format.

The numerical approach in section 4.2.1 allowed us to solve the equations by approximating the trigonometric functions (sine and cosine) and handling the square root functions. Nevertheless, such an approach is computationally complex because it requires approximation that used high degree polynomials. The stepwise approach, which consisted of including the conditions such as 'or(range = ... , bearing = ...)', helped to reduce the search space but at the same time increased the time needed for the computation. We were also able to solve the equations thanks to the fact that the trigonometric functions were in the form of sin(bearing) and cos(bearing), and the square root functions were in the form of 'sqrt(A) < B'.

The analytical approach discussed in section 4.2.2 proved to be more efficient. By using Z3's *simplify* function, we could directly simplify the equations without relying on numerical approximations. This method allowed us to handle more complex equations and provided a more straightforward way to deduce the conditions on the inputs. However, it sometimes made mistakes. For example, in the case of equation E5, the solution included a line that could have been eliminated given that the condition is always true (Figure 9). As the black line is unnecessary, it could be removed to simplify the equation further. This is due to the way sine and cosine were abstracted and used symbolically in the Z3 framework. While this approach is more efficient and flexible, it may require additional manual intervention to verify the correctness of the solutions.

In conclusion, while both approaches have their strengths and weaknesses, there is still work to be done to trace back the solutions of the equations to

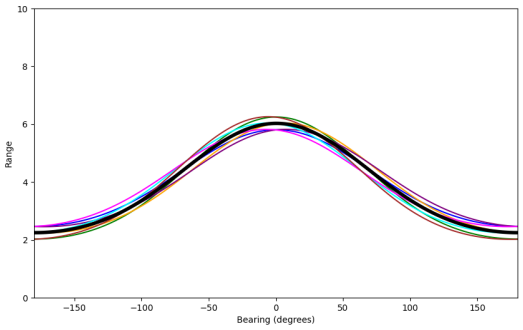

Figure 9: Result of simplifying E5 showing an unnecessary line.

the actual source code. Understanding which part of the source code generated the bugs is essential for effective debugging and ensuring the correctness of the graphical elements in the system.

References

[1] Béger, P. (2020). Vérification formelle des propriétés graphiques des systèmes informatiques interactifs. https://theses.hal.science/tel-02990362/

[2] Prun, D., & Béger, P. (2022). Formal Verification of Graphical Properties of Interactive Systems. Proceedings of the ACM on Human-Computer Interaction, 6(EICS), 1–30. https://doi.org/10.1145/3534521

[3] Cousot, P., & Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 238–252. https://doi.org/10.1145/512950.512973

[4] King, J. C. (1976). Symbolic execution and program testing. Commun. ACM, 19(7), 385–394. https://doi.org/10.1145/360248.360252

[5] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. Commun. ACM, 12(10), 576–580. https://doi.org/10.1145/363235.363259

[6] Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM, 18(8), 453–457. https://doi.org/10.1145/360933.360975

[7] Bjørner, N., de Moura, L., Nachmanson, L., & Wintersteiger, C. M. (2019). Programming Z3. In J. P. Bowen, Z. Liu, & Z. Zhang (Eds.), Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures (pp. 148–201). Springer International Publishing. https://doi.org/10.1007/978-3-030-17601-3_4

[8] Langer, S. H., & Dubois, P. F. (1998). A comparison of the floating-point performance of current computers. Computers in Physics, 12(4), 338–345. https://doi.org/10.1063/1.168693

[9] Bjørner, N., & de Moura, L. (n.d.). Applications of SMT solvers to Program Verification.

[10] Tendulkar, P. (2014). Mapping and Scheduling on Multi-core Processors using SMT Solvers [Phdthesis, Universite de Grenoble I - Joseph Fourier]. https://theses.hal.science/tel-01087271

[11] Arusoaie, A., & Pistol, I. (2019). Using SMT Solvers to Validate Models for AI Problems (No. arXiv:1903.09475). arXiv. https://doi.org/10.48550/arXiv.1903.09475

[12] Chandra, S., Fink, S. J., & Sridharan, M. (2009). Snugglebug: A powerful approach to weakest preconditions. Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, 363–374. https://doi.org/10.1145/1542476.1542517