Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Artificial Intelligence and Systems Engineering

Software Witness Validation in Theta

PROJECT LAB REPORT

Author
Klevis Imeri

Advisor
Zsófia Ádám

October 14, 2025

Contents

Chapter 1

Introduction

Software has become indispensable in modern life, permeating everything from smartphones to satellites. Software inevitably contains bugs that require fixing. Software verification is the process of identifying these bugs in computer programs or demonstrating their absence. [1] Various software verification methods exist, ranging from human-driven approaches like code reviews and software testing (including unit and integration testing) to automated techniques such as static and dynamic analysis. Formal methods, including model checking, offer mathematical approaches to software verification.

Numerous verification tools have been created by different organizations and research teams. Some prominent examples are `CPAchecker`, `UAutomizer`, `Theta`. Despite being developed independently, these tools can still share their results with other tools through witnesses.

These witnesses encapsulate verification information, enabling tools to validate results quicker, either by themselves or by other verification tools. The witness format undergoes continuous updates, with each iteration aiming to improve upon its predecessor. However, as these witness formats evolve, tools must also integrate these new versions into their frameworks. `Theta`, despite being a quite feature-rich verification framework, previously lacked a validator for the Witness 2.0 version. This paper endeavors to lay the groundwork by outlining the algorithms and implementation strategy for creating a validator for this witness type within `Theta`. The proposed approach for implementing violation witness validation for this new format involves converting the witness into an XCFA and subsequently creating a product automaton with the program XCFA. This enables `Theta`'s supported `multianalysis` to be run on this product automaton. Although the implemented validator currently lacks several features, initial results have demonstrated promising outcomes. The impact of this work lies in establishing the foundation for a feature-rich, state-of-the-art validator.

Chapter 2

Background

2.1 Model Checking

Model checking is a formal verification technique that systematically explores all possible states and transitions (the state space) of a formal model (representation) of the system to determine if it satisfies a formally specified property [3] [7] . This process typically involves:

Algorithm 1 Model Checking Reachability

```
Let $q$ := desired state to reach
Let $Q$ := set of initial states
while $q \notin Q$ and not all states visited do
    $Q \leftarrow Q \cup \mathrm{Tran}(Q)$
end while
```

2.2 Witnesses

Verifier tools analyze a program along with a safety property and other specifications, ultimately generating a result and supplementary details that support this result. This extra information, known as a witness, is structured according to a defined and widely recognized format, enabling compatibility across numerous tools. The initial proposed format for witnesses was Witness 1.0, which utilized GraphML [2]. Presently, the most recent proposed witness format is Witness 2.0, which employs YAML [1].

The structure of 2.0 witnesses is syntactically defined by three key components:

- `entry_type` – the type of the witness (”violation_sequence” or ”invariant_set”)
- `metadata` – metadata about the witness
- `content` – content that varies depending on the type of witness

There are two categories of witnesses:

- **Violation Witnesses (Counterexamples)** [1] [6]
  These are generated when the verifier finds that a safety property has been violated. They provide a vaguely define program path illustrating how the violation is

achieved. The content comprises a sequence of segments, with a final segment at the end.

Each segment contains *waypoints*, which are specific program locations the execution passes through. A segment consists of any number of avoid waypoints, and it concludes with a single follow waypoint signalling the segment's end. A waypoint includes:

- `type` – such as `assumption`, `branching`, `target`, `function_enter`, or `function_return`
- `constraint` – describes the program state or path condition
- `location` – maps to a line in the source code
- `action` – specifies whether to avoid or pass through this waypoint

- **Correctness Witnesses** [1] [5]
  These are produced when the verifier proves the program satisfies its specification. The content consists of *invariant elements*, each of which contains:
  - `type` – either a loop or location invariant
  - `location` – same as above
  - `value` – the invariant expressed in the format specified
  - `format` – the format of the invariant expression

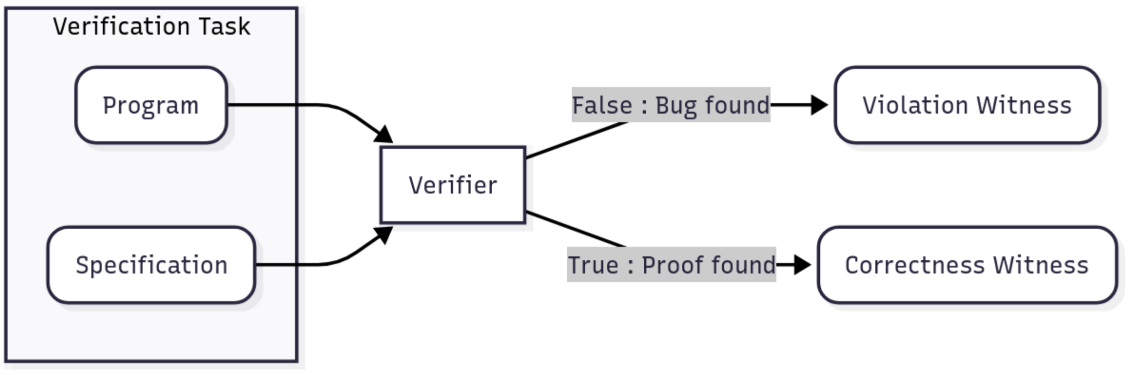


Figure 2.1: Verification

2.3 Witness-based Result Validation

Witnesses generated by different tools are typically cross-checked to ensure accuracy. A witness, being structured data in a standard format, facilitates faster result validation across various tools compared to a complete reverification. While verification and validation are often confused, validation specifically relies on a witness, which offers additional information about the program's verification process [1] [2]. Witnesses accelerate this process by providing precomputed verification data. Furthermore, they enable the validation of results obtained from other tools.

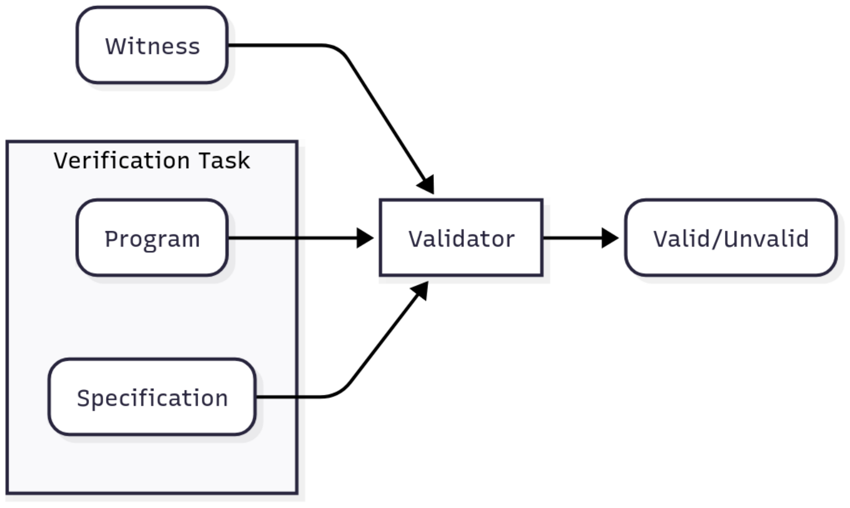


Figure 2.2: Validation

2.4 CFA

To perform model checking, a program written in source code must first be transformed into a formal model. The formal model used in the Theta validator built is an Control-Flow Automaton (CFA).

Definition 1 (Control-flow automata). Let:

- $v :=$ variable
- $D_v :=$ domain of variable v
- φ is an expression $\Rightarrow \text{var}(\varphi) = \{\forall v \mid v \in \varphi\}$
- predicate $:= p : D_{v_i} \times \cdots \times D_{v_n} \rightarrow \{\text{true}, \text{false}\}$

A control-flow automaton is a tuple CFA $= (V, L, l_0, E)$ where:

- $V := \{v_1, \ldots, v_n \mid \text{domain}(v_i) = D_{v_i}\}$
- $L := \{\text{program locations modeling the program counter}\}$
- $l_0 \in L :=$ initial program location
- $E \subseteq L \times \text{Ops} \times L := \{\text{directed edges representing operations}\}$
- $\text{Ops} = \{\text{op} \mid \text{op} = \text{assignment} \vee \text{op} = \text{assumption}\}$
  - assignment $:= (v := \varphi)$ s.t. $v \in V \wedge \varphi \in D_v \wedge \text{var}(\varphi) \subseteq V$
  - assumption $:= [\varphi]$ s.t. $\varphi :=$ predicate $\wedge\, \text{var}(\varphi) \subseteq V$ ∎

Evidently, a Control Flow Automaton (CFA) is a formalism based on graphs [7]. Furthermore, assignment and assumption statements can be represented as transition functions, making them applicable in model checking.

$$\operatorname{tran}\big(v := \varphi\big) \;=\; \big(v' = \varphi\big) \;\wedge \bigwedge_{u \in V \setminus \{v\}} (u' = u), \tag{2.1}$$

$$\operatorname{tran}\big([\psi]\big) \;=\; \psi \;\wedge\; \bigwedge_{v \in V} (v' = v). \tag{2.2}$$

Although the validator we developed in Theta uses the concept of a Control Flow Automaton (CFA) as defined here, it actually employs an Extended Control Flow Automaton (XCFA). The XCFA is simply a more comprehensive version of a CFA designed to manage more intricate processes. However, in our current implementation, we don't utilize any of the additional capabilities offered by the XCFA.

2.5 Product Automaton and Multianalysis

Another formalism we'll need to understand is the Product Automaton. The intuition behind its importance lies in its ability to represent the combination of different formalism. We often model individual systems separately, but we need a way to understand their combined behavior, or what happens when these models interact in a defined manner. Informally, a Product Automaton is precisely this: the result of combining two formalisms. `Multianalysis`, then, is the analysis performed on this resulting Product Automaton. While you can theoretically combine any two formalisms, in our specific case, we're interested in the product automaton of two (X)CFAs. The states of the combined system can be conceptualized as pairs, meaning the state space is the Cartesian product of the states from the two individual automata. Subsequently, it is necessary to define the transition rules between these product states. Typically, these transitions depend on the properties of both automata and the specified method of their combination. Section 3.3 details the construction of the product automaton from the witness XCFA and the program XCFA.

Chapter 3

Validation in Theta

3.1 Overview

As mentioned in `Theta`'s specification [4], `Theta` is a very generic and highly configurable framework for performing model checking. The way the Validator for witness 2.0 is implented in Theta is using XCFAs and Product Automaton. Even though XCFA's was chosen as formalism for further extensibility in the current version of the validator only the CFA capabilities where used. For te moment the validator only supports Violation Witnesss. The way the violation witness verifier works is described in the Figure 3.1.

There is no specification (and thus no specific safety property), because for the violation witness we are checking the reachability property—whether the error location in the program and the target location in the witness are reached. The target location of the witness is modeled as an error state to make the algorithm easy compatible with `multianalyses`.

The C program is parsed and transformed into the program XCFA. The witness is parsed as YAML and then transformed into the witness XCFA.

While transforming the C program from AST to XCFA, much of the AST node data is added as metadata to the XCFA locations and edges. This metadata is used to ensure that the waypoints of the witness point to the correct places in the program XCFA. The pointing and correlation between the XCFAs are done using a global variable waypoint, which is assigned in the program XCFA and assumed in the witness XCFA.

From each XCFA, a Multianalysis side is created. Then the sides are combined to form the `multianalysis`. Because of how the waypoint variable is handled, the `multianalysis` is straightforward: each side is moved in sequence, starting from the program XCFA, until error states are reached in both XCFAs.

If both error states are reached, the witness is considered valid; otherwise, it is considered invalid.

3.2 Generating XCFA from Violation Witnesses

Within the validator in Theta, a crucial step involves transforming Violation Witnesses from their YAML into a more formal structure, specifically XCFA.

To facilitate this conversion, we need a clear methodology for translating each waypoint into its corresponding XCFA elements. Below, for every waypoint type, we'll outline how

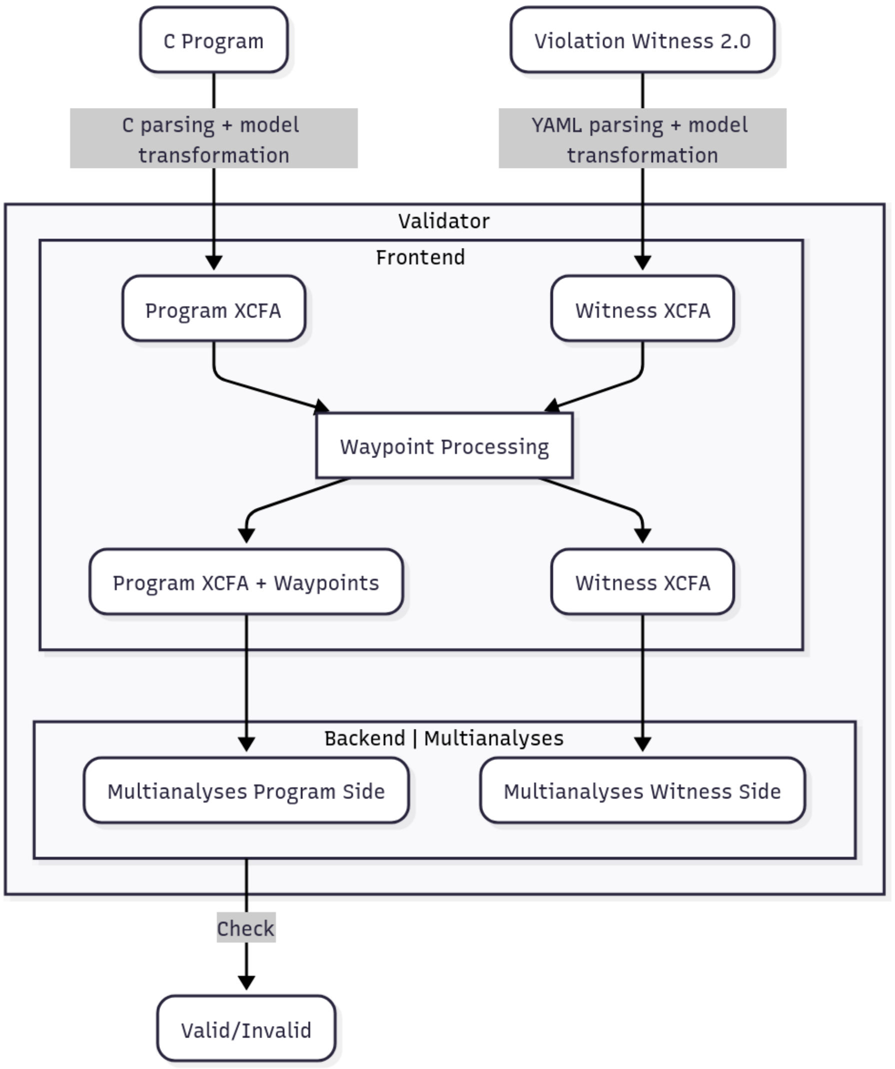


Figure 3.1: Validation overview in Theta

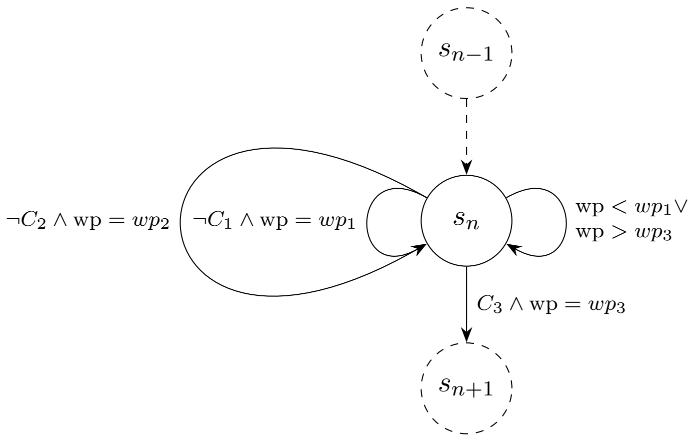
it's translated into the corresponding elements of the XCFA. For our current purposes, it is sufficient to consider a waypoint (`wp`) variable as directly corresponding to a specific location in the C source file. The XCFA program itself also identifies these locations. Accordingly, each time the XCFA program arrives at one of these points, the XCFA witness is called to process the same location. In our specific case, since the waypoints for the witness are essentially edges, the `wp` variable role is simply to determine which particular edge needs to be checked.

We consider two types of assumption waypoints:

- An **avoid assumption** specifies a condition that the program must *not* satisfy [1]. In other words, this assumption must not be met. An avoid assumption can be thought of as a negated follow assumption that does not cause an advancement to the next segment. This implies that each time the program logic encounters an avoid assumption, it must check that the specified condition does not hold. If the condition does not hold (i.e., the avoidance is successful), the program can be modeled as having an edge transitioning back to, or remaining within, the current segment node. Conversely, if the condition *does* hold (i.e., the program is unsuccessful in avoiding it), then the execution will have no further states to transition to.

- A **follow assumption** dictates that the program *must* traverse a specific waypoint within the current segment. After successfully passing through this waypoint, the program is required to transition to the next segment [1]. This means the waypoint associated with a **follow** assumption must be visited only once. Therefore, this type of assumption is represented by an edge leading from the current state (after satisfying the waypoint condition) to the node representing the subsequent segment.

```
- segment:
  - waypoint:
    type: "assumption"
    action: "avoid"
    constraint: <C_1>
    location: <wp_1>
  - waypoint:
    type: "assumption"
    action: "avoid"
    constraint: <C_2>
    location: <wp_2>
  - waypoint:
    type: "assumption"
    action: "follow"
    constraint: <C_3>
    location: <wp_3>
```



s_{n-1}
$\neg C_2 \wedge \text{wp} = wp_2$
$\neg C_1 \wedge \text{wp} = wp_1$
s_n
$\text{wp} < wp_1 \vee$
$\text{wp} > wp_3$
$C_3 \wedge \text{wp} = wp_3$
s_{n+1}

Figure 3.2: XCFA mapping of the assumption type waypoints and their constrains. The `C` variables are c expressions and the `wp` variables are just a tuple of row:col.

One of the edges depicted in the figure, which has not been previously explained, is the loop at state S_n. This loop is labeled with the condition `wp` $< wp_1 \vee$ `wp` $> wp_3$. Such an edge, which we will call the **segment guard** edge, is necessary because, within the current segment, we are only concerned with processing waypoints that specifically correspond to this segment. Since the waypoints are ordered according to their `wp` value (these are values mapped from their source code locations, rather than the locations themselves, which means that even if the source locations are not ordered, the `wp` values are), a range check, as exemplified by the aforementioned label, is sufficient.

A while loop provides a good example for understanding the difference between *avoid* and *follow* assumptions. To clarify the usual confusion surrounding these concepts, and to explain why they are necessary and defined in this manner, consider the following illustration. Suppose, looking at the simple `while loop` in Figure 3.3, the objective is to ensure that the variable v always remains greater than 200; effectively, this means we aim for the s variable to overflow. One could achieve this by employing a *follow* assumption, such as $v > 200$, applied across n distinct program segments. Alternatively, a more concise approach is to define a single *avoid* assumption for the relevant segment, for instance, to avoid paths where $v \leq 200$.

```
uchar v=0, s=0, i=0, n=2;
while (i < n) {
  v = nondet();
  // AVOID (assume v <= 200)
  s += v; ++i;
}
```

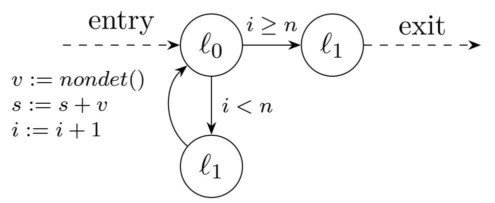


Figure 3.3: A simple `while loop` and its corresponding branching in the XCFA of the program.

Branching waypoints can be located at control flow statements such as `if`, `while`, a `ternary operator (?:)`, or a `switch` statement. It is important to first note that in an XCFA, these C constructs are all represented as branching structures. The XCFA representation for `if`, `while` statements and the `ternary operator (?:)` is generally similar, whereas it tends to be more complex for `switch` statements. As an illustration, Figure 3.3 depicts the XCFA of a `while` loop, where the XCFA branches into two directions. A `switch` statement, in contrast, typically involves a greater number of branches. This branching mechanism enables the differentiation of the path taken within the XCFA program solely by using the `wp` variable. For example, if a branching waypoint indicates that the `while` loop condition evaluates to true, the `wp` variable for that waypoint would be mapped to the edge between ℓ_0 and ℓ_1. Consequently, if the XCFA program follows the path corresponding to the condition $i > n$ (representing the 'true' branch in this scenario), we can determine that this specific branch was taken by inspecting the `wp` variable.

In the current version of Theta's validator, we only support `if` and `while` types of branching waypoints:

- An **avoid branching** waypoint dictates that the program execution should never traverse the branch specified by this waypoint [1]. In other words, the `wp` variable should never be equal to (i.e., point to) the location (in the program XCFA) associated with this type of waypoint. To represent this in the witness XCFA, no specific action is taken; essentially, no edge is defined for this type of waypoint. This is because encountering this type of waypoint signifies that the program execution should halt (i.e., "get stuck"). This approach differs from the previously discussed `avoid assumption`. In the case of an `avoid assumption`, even if the `wp` variable aligns with the waypoint's location, the execution can still proceed if the associated constraint does not hold. However, for the type of waypoint being discussed here, encountering it mandates an immediate stop of the execution. This is achieved in the witness XCFA by not adding an edge, effectively meaning there is no subsequent state to transition to.

- A **follow branching** waypoint indicates that the program execution *must* follow this particular branch [1]. We can easily verify if the program indeed followed this branch by checking if the `wp` variable is equal to the location of this waypoint. In the witness XCFA, this can be represented by an edge leading to the subsequent segment, which is traversed if the `wp` variable matches this waypoint's location.

It should be noted that branching waypoints possess a constraint indicating how the condition of an `if`, `while`, `ternary operator (?:)`, or `switch` statement evaluated. For `if`, `while`, and the `ternary operator (?:)`, this evaluation can result in "true" or "false". For a `switch` statement, it can be an integer constant (representing the value of the variable being switched upon) or the "default" case. Consequently, this constraint informs us which path the program should take. Since the `wp` variable is associated with the path we intend the program to follow, this implies that the `wp` variable also implicitly ”contains” this constraint (i.e., the constraint can be inferred from it). Therefore, in a potentially simplified YAML witness representation (such as one discussed below), the explicit constraint might be omitted, with only the location being kept. This, however, is not the case in the complete or actual YAML witness specification.

```
- segment:
  - waypoint:
    type: "branching"
    action: "avoid"
    location: <wp_1>
  - waypoint:
    type: "branching"
    action: "avoid"
    location: <wp_2>
  - waypoint:
    type: "branching"
    action: "follow"
    location: <wp_3>
```

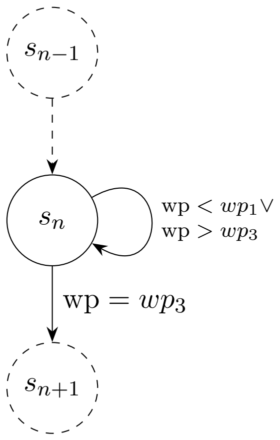


Figure 3.4: XCFA mapping of a branching-type waypoint, where the `wp` variable is a tuple (row:col, constraint) indicating which XCFA branch was taken.

There is only one type of target waypoint:

- A **follow target** waypoint indicates that the program execution should reach the designated error location [1]. It is positioned immediately before the error, ensuring that no other program evaluations occur before the error state is encountered. In the witness XCFA, this can be readily represented by an edge pointing directly to the error location.

This type of waypoint does not have an associated constraint.

```
- segment:
  - waypoint:
    type: "target"
    action: "follow"
    location: <wp_0>
```

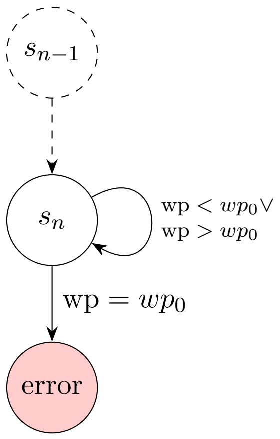


Figure 3.5: XCFA mapping of target type waypoint. The `wp` variables are a tuple of row:col.

A segment is designated as final if its follow waypoint is a target. The following is an example segment that includes all the aforementioned types of waypoints.

```
- segment:
  - waypoint:
    type: "branching"
    action: "avoid"
    location: <wp_1>
  - waypoint:
    type: "assumption"
    action: "avoid"
    constraint: <C_2>
    location: <wp_2>
  - waypoint:
    type: "target"
    action: "follow"
    location: <wp_3>
```

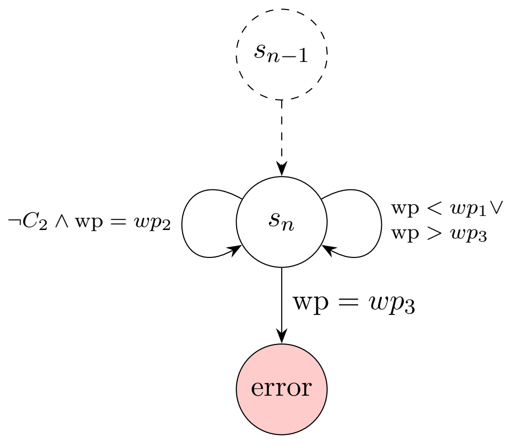


Figure 3.6: XCFA mapping of a final segment containing branching, assumption, and target waypoints, along with their respective constraints.

3.3 Combining and Executing the XCFA's

As seen in Figure 3.1, after the program and witness XCFAs are created and their corresponding waypoints (locations) are matched, we proceed to run the `multianalysis`. Before explaining how the XCFAs are executed together, we should first examine what a generic witness may look like. Figure 3.7 (left) shows such a witness.

A witness XCFA can be composed of a start location, a sequence of one or more segments, and an error location. Segments are represented by nodes labeled S_i. To represent the possibility of many segments, nodes such as S_{n-1} and S_{n+1} are visualized as dashed (meaning optional). From the discussion above, it becomes clear that segment S_n is a depiction of a general segment. Dashed edges signify that they may or may not be present. For instance, several dashed edges on the right side indicate that there can be multiple `avoid` conditions within a segment. In contrast, the bold edges, which represent the `follow` waypoint and the segment guard, are necessary components. Here, the range $1 < i \leq n-1$ for an index i delineates the waypoints within the current segment S_n, excluding the final one; these waypoints are labeled $wp_{s_n i}$. The `C` variables in this context represent several expressions combined. They can aslo be empty.

On the right, a representation of a generic program XCFA is shown. Here, `k` within the program XCFA represents the branching number. For an `if` statement, this number is 2, whereas for a `switch` statement, it can be as large as the number of cases in the switch. It is important to note that not all edges in this XCFA necessarily have associated waypoints. The bold edge depicted between ℓ_n and ℓ_{n+1} represents an arbitrary edge within the program XCFA that does have an associated waypoint.

Now, let's discuss how the `multianalysis` is run. We begin from the `start` node in both XCFAs. The program XCFA's execution proceeds until a waypoint is encountered. Suppose the current waypoint reached is $\texttt{wp}_{s_n i}$. The `wp` variable is set to this current waypoint, $\texttt{wp}_{s_n i}$, and the `run` variable is set to `true`, indicating that a waypoint has been processed and the witness XCFA should now execute. The `run` := `true` assignment is associated with every waypoint location in the program XCFA, while a `run` := `false` assignment is present on every edge of the witness XCFA. This detail was abstracted away previously, but here it can be clearly seen that every edge of the witness XCFA has the

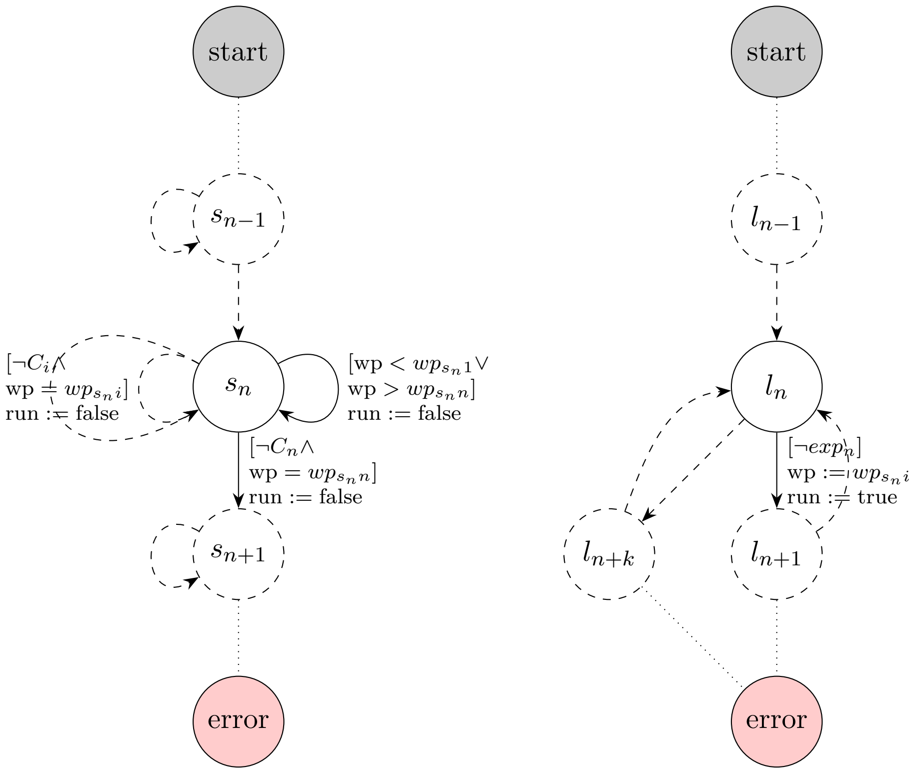


Figure 3.7: Comparison of a generic witness XCFA (left) and a generic program XCFA (right). Dashed nodes or edges are optional. Dotted lines represent an abstracted list of segments. Each XCFA has a start and an error location.

`run` := `false` assignment. Thus, after the `run` variable is set to `true` by the program XCFA, control transfers to the witness XCFA.

The variables and the state are shared between the XCFAs, so the witness XCFA utilizes the set `wp` variable and the current program state to take one step. The `wp` variable, in conjunction with edge guards, is used to determine which edge (and thus which waypoint) is currently being evaluated. The conditions of this chosen edge are then checked for satisfiability. If the guard conditions on this edge are not met, the execution halts, and the witness is deemed invalid. Conversely, if the guard conditions are met, depending on the `wp` variable, the witness XCFA may either remain in the current segment or transition to the next one:

- If `wp` does not correspond to a waypoint within the current segment, the segment guard edge is taken, and the witness XCFA remains in the current segment.
- If `wp` corresponds to one of the `avoid` edges, the witness XCFA remains in the current segment.
- If `wp` corresponds to the `follow` waypoint, the witness XCFA transitions to the next segment.

After an edge in the witness XCFA is taken, the `run` variable is set to `false`, signaling that the program XCFA now resumes control of the execution. This process continues until the error states of both XCFAs are reached simultaneously, at which point the witness is considered valid.

Even though one might intuitively consider the states of the program XCFA and the witness XCFA separately, they are, in this context, components of product states within a unified product automaton. The analysis method employed, termed multianalysis, operates directly on this product automaton. The product states are pairs, denoted as (s, l), where s is a state from the witness XCFA and l is a state from the program XCFA. A set, which we will call `states`, is maintained to hold the current product states being processed. For traversing these product states, various strategies such as Breadth-First Search (BFS) or Depth-First Search (DFS) can be employed. For this explanation, a BFS order will be adopted due to its illustrative simplicity. However, it is worth noting that for the specific example discussed below, the choice between BFS and DFS will not make a discernible difference in the trace. This is because the set of current states, `states`, will consistently contain only a single state. This is a consequence of the witness (in the example below, as not every witness would necessarily provide such a narrow path) explicitly dictating which branch the program must take at any decision point, thereby preventing the simultaneous exploration of multiple paths.

The construction of edges in this product automaton is governed by the edges of the constituent XCFAs:

- If an edge exists in the witness XCFA from state s_i to s_k, then for any program XCFA state l, an edge is formed in the product automaton from the product state (s_i, l) to (s_k, l).
- Symmetrically, if an edge exists in the program XCFA from state l_i to l_k, then for any witness XCFA state s, an edge is formed in the product automaton from the product state (s, l_i) to (s, l_k).

We also require a set to maintain the shared state, which we will call `varsExpl`. For this example, we will employ an explicit analysis, meaning we track explicit variable values rather than symbolic conditions, for simplicity. The execution begins with `states` $= \{(\text{start}_p, \text{start}_w)\}$ and `varsExpl` $= \{\}$. Variables are initially undefined and will be set upon their first assignment.

Both XCFAs are positioned at their respective start nodes. Since `run` is initially `undefined`, by default the program XCFA proceeds first.

The core algorithmic process is relatively straightforward: the active XCFA is identified, its permissible outgoing edges are determined, and these edges are then traversed to explore the states reached. This transition can be represented by the set of edges_taken and the resulting state. A single step of the algorithm can thus be thought of as:

```
||||| Step n |||||
edges_taken = {...}; // edges where the enabling condition evaluated to true
states = {...};        // resulting states of taking those edges
varsExpl = {...};    // the explicit shared state of variables belonging to the XCFAs.
```

These steps are run until the state `states` $= \{(\text{error}_p, \text{error}_w)\}$ is reached, where error_p is the error state of the program XCFA and error_w is the error state of the witness XCFA. If we manage to reach this state, it means we have successfully passed all the segments and the waypoints in them.

3.4 Example

To clarify how the validator works, Figures 3.8 and 3.9 display an example of an unsafe C program and a corresponding violation witness for this program, along with their respective XCFAs. In this context, the `nondet()` function returns a nondeterministic value within the range of 0 to 255, inclusive (i.e., the range of an `unsigned char`).

```c
int main() {
  unsigned char n = nondet();
  if (n == 0) return 0;

  unsigned char v = 0;
  unsigned char s = 0;
  unsigned int i = 0;
  while (i < n) {
    v = nondet();
    s += v;
    ++i;
  }

  if (s < v) {
    reach_error();
    return 1;
  }

  return 0;
}
```

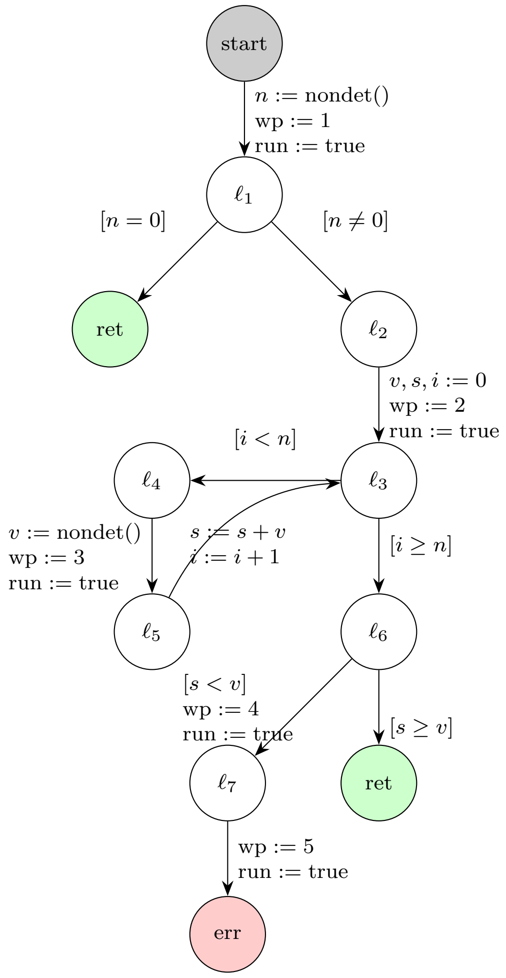


Figure 3.8: The unsafe C program on the left and its corresponding XCFA on the right.

Let's trace the execution steps of the algorithm. The initial state is defined as follows:

```
||||| Step 0 |||||
edges_taken = {};
states = {(start_w, start_p)};
varsExpl = {};
```

We consider the edge from start_w to l_1 in the program XCFA. The assignment $n :=$ nondet() is equivalent to the condition $0 \leq n \leq 255$. However, since predicate analysis is not utilized in this context, this broad range is typically not explicitly tracked. This initial range is, nonetheless, factored in if another condition emerges that further constrains it. Consequently, n will only be included in the set of explicit variables if its range is reduced to a singular, precisely determined value. Since the start state of the program XCFA does not have a loop edge to itself, and assuming the transition $\text{start}_\text{w} \to l_1$ is taken, the new program state is l_1. The witness XCFA remains in start_p. Thus, the new combined state

```
- entry_type: "violation_sequence"
metadata: <...>
- segment:
  - waypoint:
    type: "assumption"
    action: "follow"
    constraint:"n == 2"
    location: 3:3
- segment:
  - waypoint:
    type: "assumption"
    action: "follow"
    constraint: "v==0 && s==0 && i==0"
    location: 8:3
- segment:
  - waypoint:
    type: "assumption"
    action: "avoid"
    constraint: "v<=254"
    location: 10:5
  - waypoint:
    type: "branching"
    action: "follow"
    constraint: "true"
    location: 14:3
- segment:
  - waypoint:
    type: "target"
    action: "follow"
    location: 15:5
```

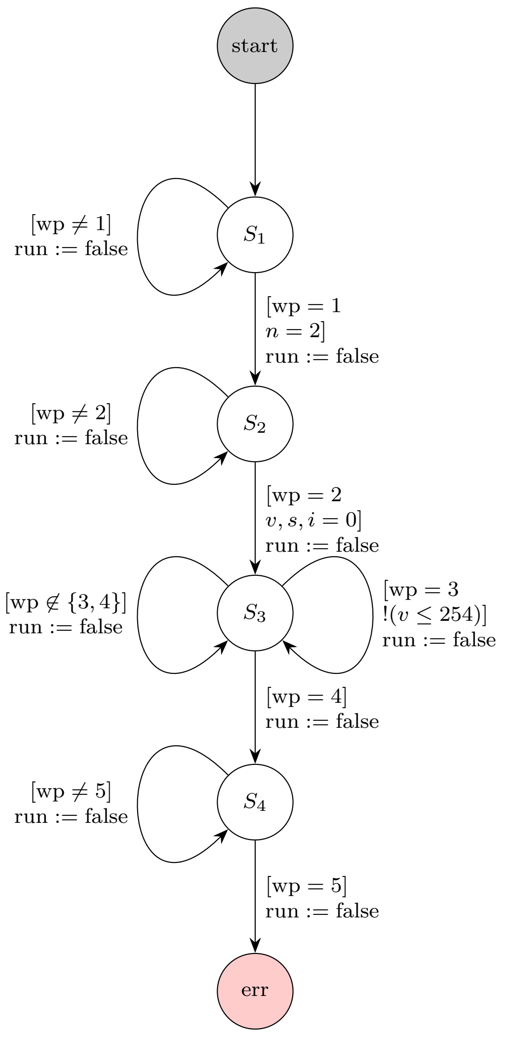


Figure 3.9: The violation witness on the left and its corresponding XCFA on the right.

discovered is $(l_1, \text{start}_\text{p})$. In this transition, `wp` is set to 1 and `run` becomes true. Thus, the state after this step is:

```
||||| Step 1 |||||
edges_taken = {(start_w, l_1)};
states = {(l_1, start_p)};
varsExpl = {wp=1, run=true};
```

Since `run` is set to true, we proceed with the witness XCFA. In the witness XCFA, currently in state start_p, we first take the edge $\text{start}_\text{p} \rightarrow s_1$. This transition does not set `run` to false, so we would typically continue within the witness XCFA if further local transitions were possible and highest priority. Assuming this is the chosen transition for this step:

```
||||| Step 2 |||||
edges_taken = {(start_p, s_1)};
states = {(l_1, s_1)};
varsExpl = {wp=1, run=true};
```

The next possible edges from s_1 in the witness XCFA are $s_1 \rightarrow s_1$ and $s_1 \rightarrow s_2$. For the edge $s_1 \rightarrow s_1$, the condition does not hold (as `wp` is 1), so this edge is not taken. However, for the edge $s_1 \rightarrow s_2$, the condition $\text{wp} = 1$ holds, so we take this edge. This indicates that we have passed the current segment and now move to the next segment in the witness. Here, n is assigned a value (in this instance, $n = 2$), which considerably narrows down the potential paths to an error state. The resulting step is:

```
||||| Step 3 |||||
edges_taken = {(s_1, s_2)};
states = {(l_1, s_2)};
varsExpl = {n=2, wp=1, run=false};
```

The `run` variable has now been set to false, so control returns to the program XCFA. From state l_1 in the program XCFA, the possible edges are $l_1 \rightarrow \text{ret}$ and $l_1 \rightarrow l_2$. We do not take the edge $l_1 \rightarrow \text{ret}$ because the witness set n to 2, and $2 \neq 0$. The other edge, $l_1 \rightarrow l_2$, is taken, so the step is:

```
||||| Step 4 |||||
edges_taken = {(l_1, l_2)};
states = {(l_2, s_2)};
varsExpl = {n=2, wp=1, run=false};
```

Let's fast-forward the algorithm. The subsequent steps, until both XCFAs reach their error locations, are summarized below. Each block represents one step.

```
||||| Step 5 |||||
edges_taken = {(l_2, l_3)};
states = {(l_3, s_2)};
varsExpl = {v=0, s=0, i=0, n=2, wp=2, run=true};
```

```
||||| Step 6 |||||
edges_taken = {(s_2, s_3)};
states = {(l_3, s_3)};
varsExpl = {v=0, s=0, i=0, n=2, wp=2, run=false};
```

Here, if $v, s,$ and i were not zero, the assumption would have failed, and the witness would get stuck. Consequently, our analysis would conclude with an invalid witness.

```
||||| Step 7 |||||
edges_taken = {(l_3, l_4)};
states = {(l_4, s_3)};
varsExpl = {v=0, s=0, i=0, n=2, wp=2, run=false};
```

```
||||| Step 8 |||||
edges_taken = {(l_4, l_5)};
states = {(l_5, s_3)};
varsExpl = {s=0, i=0, n=2, wp=3, run=true};
```

The variable v gets removed from the explicit state because, as mentioned before, `nondet()` assigns a range (e.g., $[0, 255]$), not a single value required for an explicit state representation here.

```
||||| Step 9 |||||
edges_taken = {(s_3, s_3)}; // The one with the condition !(v<=254)
states = {(l_5, s_3)};
varsExpl = {v=255, s=0, i=0, n=2, wp=3, run=false};
```

As the `nondet` assignment put v in the range $[0, 255]$, the assumption $!(v \leq 254)$ implied that v can only be 255.

```
||||| Step 10 |||||
edges_taken = {(l_5, l_3)};
states = {(l_3, s_3)};
varsExpl = {v=255, s=255, i=1, n=2, wp=3, run=false};
```

```
||||| Step 11 |||||
edges_taken = {(l_3, l_4)}; // We enter the loop again
states = {(l_4, s_3)};
varsExpl = {v=255, s=255, i=1, n=2, wp=3, run=false};
```

```
||||| Step 12 |||||
edges_taken = {(l_4, l_5)};
states = {(l_5, s_3)};
varsExpl = {s=255, i=1, n=2, wp=3, run=true}; // v removed from explicit state
```

Even though `wp` stays the same, `run` is set to true, so the witness XCFA takes the turn. v is again removed from the explicit state for the same reason as before.

```
||||| Step 13 |||||
edges_taken = {(s_3, s_3)};
states = {(l_5, s_3)};
varsExpl = {v=255, s=255, i=1, n=2, wp=3, run=false}; // v becomes 255
```

```
||||| Step 14 |||||
edges_taken = {(l_5, l_3)};
states = {(l_3, s_3)};
varsExpl = {v=255, s=254, i=2, n=2, wp=3, run=false}; // s becomes 254
```

Notice here that s experienced an overflow and became 254.

```
||||| Step 15 |||||
edges_taken = {(l_3, l_6)}; // Exit the loop
states = {(l_6, s_3)};
varsExpl = {v=255, s=254, i=2, n=2, wp=3, run=false};
```

```
||||| Step 16 |||||
edges_taken = {(l_6, l_7)};
states = {(l_7, s_3)};
varsExpl = {v=255, s=254, i=2, n=2, wp=4, run=true};
```

```
||||| Step 17 |||||
edges_taken = {(s_3, s_4)};
states = {(l_7, s_4)};
varsExpl = {v=255, s=254, i=2, n=2, wp=4, run=false};
```

```
||||| Step 18 |||||
edges_taken = {(l_7, err_p)};
```

```
states = {(err_p, s_4)};
varsExpl = {v=255, s=254, i=2, n=2, wp=5, run=true};
```

```
||||| Step 19 |||||
edges_taken = {(s_4, err_w)};
states = {(err_p, err_w)};
varsExpl = {v=255, s=254, i=2, n=2, wp=5, run=false};
```

As you can see here, we reached the error states in both XCFAs (err_p for program, err_w for witness), which means our violation witness is a valid witness.

3.5 Limitations

As mentioned in Section 2.2, violation witnesses also include `function_enter` and `function_return` waypoints. Although their implementation is expected to be straightforward, these two waypoint types are not currently supported. Additionally, `switch` statements are not yet supported, though work is in progress to enable their functionality.

The previously mentioned limitations are relatively minor issues that could be resolved quickly within the existing framework. However, a more significant challenge, which the Theta team is addressing, was the limited metadata available for the program XCFA concerning the program structure and specific locations. This has now been partially addressed with the introduction of ARG node metadata. Nevertheless, this type of metadata can still be prone to bugs, potentially leading to failures in waypoint matching between the XCFAs.

Although the implemented validator has undergone considerable testing, its complexity necessitates a more comprehensive testing suite.

Chapter 4

Related Work

There are numerous verification and validation tools available, with `CPAchecker` and `UAutomizer` being among the most prominent, including `Theta`. Both are classified as static validators, meaning they analyze the program without actual execution, unlike dynamic validators [2]. We will now compare our solution to these two established validators.

It is well known that `CPAchecker` employs its Configurable Program Analysis (CPA) framework for both verification of programs and the validation of witnesses. For violation witness validation, `CPAchecker` executes a composite CPA. This typically includes a LocationCPA, two AutomatonCPA instances (one for the observer automaton representing the specification, and another for the witness automaton), and potentially further CPAs like an IntervalCPA to provide specific analysis capabilities [2]. The program is initially parsed into a Control-Flow Automaton (CFA). Subsequently, the CPA algorithm is run on this CFA, guided by the witness and specification automata, and the validation results are then outputted.

Compared to our `Theta` validator, the `CPAchecker` validator possesses a broader range of features and handles both violation and correctness witnesses. Furthermore, `CPAchecker` supports witness refinement, a capability our current validator does not offer.

On the other hand, `UAutomizer`, when validating violation witnesses, utilizes a technique quite similar to the one implemented in our `Theta` validator. It effectively creates a product of the program's automaton (CFA) and the witness automaton [2]. When compared to our validator, `UAutomizer` is a more complete tool, providing more features, including the significant capability of supporting correctness witnesses.

Chapter 5

Conclusion

With the exploding volume of software being produced, the necessity for robust testing and verification has reached an unprecedented scale. However, formal verification tools often lag in possessing the necessary features to enhance their practical applicability in industry. To address issues such as compatibility among these verification tools, witnesses were introduced as a form of certificate. The underlying concept was to establish a standard method for expressing verification results, while also incorporating additional information to expedite the validation process. Initially, Witness 1.0 was introduced, subsequently followed by the development of Witness 2.0. The Theta modeling framework, despite its rich feature set, previously lacked a validator for Witness 2.0; consequently, our objective was to develop one. This paper lays the foundational groundwork for such a validator. Although the validator currently supports only a limited range of features, we anticipate its future expansion into a comprehensive, feature-rich, state-of-the-art tool.

5.1 Future work

As discussed in the limitations section, numerous features are currently absent and will require eventual implementation. This paper primarily details the algorithms for violation Witness 2.0. In contrast, a substantial amount of work remains for the development and explanation of correctness witness validation. Furthermore, an important feature that our Theta validator currently does not support, but which may be incorporated in future development, is witness refinement.

Acknowledgements

I am deeply grateful to Balázs Rippl. His help in understanding how to effectively implement a `multianalysis` in `Theta` was invaluable to this work.

Bibliography

[1] Paulína Ayaziová, Dirk Beyer, Marian Lingsch-Rosenfeld, Martin Spiessl, and Jan Strejček. *Software Verification Witnesses 2.0*, page 184–203. Springer Nature Switzerland, October 2024. ISBN 9783031661495. DOI: `10.1007/978-3-031-66149-5_11`. URL `http://dx.doi.org/10.1007/978-3-031-66149-5_11`.

[2] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. Verification witnesses. *ACM Trans. Softw. Eng. Methodol.*, 31(4), September 2022. ISSN 1049-331X. DOI: `10.1145/3477579`. URL `https://doi.org/10.1145/3477579`.

[3] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer International Publishing, 2018. ISBN 9783319105758. DOI: `10.1007/978-3-319-10575-8`. URL `http://dx.doi.org/10.1007/978-3-319-10575-8`.

[4] Fault Tolerant Systems Research Group (FTSRG). Theta Project Repository: README.md File (Version from Commit 991b764). URL `https://github.com/ftsrg/theta/blob/991b764492bd3e65c9a379d9b141cde2525ed0d5/README.md`.

[5] SoSy-Lab. Online schema for correctness witness 2.0, 2025. URL `https://sosy-lab.gitlab.io/benchmarking/sv-witnesses/yaml/correctness-witnesses.html`.

[6] SoSy-Lab. Online schema for violation witness 2.0, 2025. URL `https://sosy-lab.gitlab.io/benchmarking/sv-witnesses/yaml/violation-witnesses.html`.

[7] Ákos Hajdu. *Effective Domain-Specific Formal Verification Techniques*. PhD thesis, Budapest University of Technology and Economics, Budapest, June 2020. URL `http://hdl.handle.net/10890/13523`. Ph.D. Dissertation, supervised by Zoltán Micskei.