From Informal Specifications to an ABV Framework for Industrial Firmware Verification

Samuele Germiniani, Moreno Bragaglio, Graziano Pravadelli

From informal specifications to an ABV framework for industrial firmware verification

Samuele Germiniani, Moreno Bragaglio, and Graziano Pravadelli

University of Verona, Department of computer science, Italy
name@surname@univr.it

Abstract. Firmware verification for small and medium industries is a challenging task; as a matter of fact, they generally do not have personnel dedicated to such activity. In this context, verification is executed very late in the design flow, and it is usually carried on by the same engineers involved in coding and testing. The specifications initially discussed with the customers are generally not formalised, leading to ambiguity in the expected functionalities. The adoption of a more formal design flow would require the recruitment of people with expertise in formal and semi-formal verification, which is not often compatible with the budget resources of small and medium industries. The alternative is helping the existing engineers with tools and methodologies they can easily adopt without being experts in formal methods.

The paper follows this direction by presenting MIST, a framework for the automatic generation of an assertion-based verification environment and its integrated execution inside an off-the-shelf industrial design tool. In particular, MIST allows generating a complete environment to verify C/C++ firmware starting from informal specifications.

Given a set of specifications written in natural language, the tool guides the user in translating each specification into an XML formal description, capturing a temporal behaviour that must hold in the design. Our XML format guarantees the same expressiveness of linear temporal logic, but it is designed to be used by designers that are not familiar with formal methods. Once each behaviour is formalised, MIST automatically generates the corresponding testbench and checker to stimulate and verify the design. To guide the verification process, MIST employs a clustering procedure that classifies the internal states of the firmware. Such classification aims at finding an effective ordering to check the expected behaviours and to advise for possible specification holes.

MIST has been fully integrated into the IAR System EmbeddedWorkbench. Its effectiveness and efficiency have been evaluated to formalise and check a complex test plan for industrial firmware.

Keywords: verification, testing, simulation, checker, PSL, LTL, specification

1 Introduction

In the last few decades, verification has become one of the most crucial aspects of developing embedded systems. Thoroughly verifying the correctness of a design often leads to identifying bugs and specification holes far earlier in the deployment process, exempting the developing company from wasting resources in costly maintenance.

Software bugs can become exceptionally expensive when they are intentionally used to exploit vulnerabilities [1] or when they cause accidental software failures [2]. The cost worsens depending on how late the bug is discovered in the developing process. The Systems Sciences Institute at IBM [3] reports that fixing a bug discovered during the implementation phase is roughly six times more costly than fixing a bug identified during requirements analysis; fixing an error discovered after release is up to 100 times more expensive than one identified during maintenance. To sum things up, the cost of bugs escalates exponentially after each step of the developing cycle. The National Institute of Standards and Technology (NIST) estimates that the US economy loses 60 billion annually in costs associated with developing and distributing patches that fix software faults and vulnerabilities [4].

However, our experience suggests that many companies have to cut down the verification process due to the lack of time, tools and specialized engineers. To make things worse, developing time is often hard to assess correctly [5], while managers usually tend to underestimate it. As a result, engineers and programmers are subject to very firm deadlines; hence they are mostly concerned about conjuring functionalities instead of carefully verifying the design [6].

That is even more critical in the case of firmware verification, which requires exceptional consideration to deal also with the underlying hardware. Complex industrial designs usually include various firmware instances executed on different target architectures, which need to be co-simulated. Furthermore, virtual platforms and simulators are not available for each target architecture or they are not equipped with the proper verification tools. Therefore, several companies postpone firmware verification at the end of the design process, when the real hardware is available, finally asking the verification engineers to manually check if the firmware meets the specifications.

Indeed, one of the main problems that prevent an effective and efficient firmware verification process is the incapability of formalizing the initial design specification, which is generally written in extremely long and ambiguous natural-language descriptions. Such descriptions risk being differently interpreted by designers and verification engineers, as well as by the project's customers themselves, thus leading to the misalignment between the initial specification and the final implementation [7]. Besides, the lack of formalisation prevents the engineer from exploiting automatic tools for verification, with the consequent adoption of ineffective and inefficient (semi-)manual approaches. In particular, without a well-defined specification, it becomes impractical to define any formal or semi-formal verification strategy. Generally, those strategies require describing the expected behaviours in terms of logic assertions unambiguously. In the

case of semi-formal approaches, the verification engineer has to define a set of testbenches to stimulate the design under verification. To accomplish that, the verification engineer must identify and learn additional tools, further increasing the verification overhead.

To fill in the gap, we present MIST: an all-in-one tool capable of generating a complete environment to verify C/C++ firmware starting from informal specifications. The tool provides a user-friendly interface to allow designers and their customers, which are not familiar with temporal logic, to formalise the initial specifications into a set of non-ambiguous temporal behaviours. From those, MIST generates a verification environment composed of monitors (checkers) and testbenches to verify the correctness of the firmware implementation automatically. Then, in order to guide the verification process, MIST employs a clustering procedure that classifies the internal states of the firmware. Such classification aims at finding an effective ordering to check the expected behaviours and to advise for possible specification holes. The verification environment has been fully integrated with the popular IAR Embedded Workbench toolchain [8]. We evaluated the tool by verifying the correctness of an already released industrial firmware, allowing the discovery of bugs that were never detected previously.

The rest of the paper is organized as follows. Section 2 summarizes the state of the art. Section 3 overviews the methodology. Sections 4, 5, 6, 7 explain in detail the methodology implemented in MIST. Section 8 reports the experimental results. Finally, in 9 we draw our conclusions.

2 Background

Formalisation of specifications is the process of translating requirements of a design into logic properties that can be used to verify its correctness automatically. Usually, the procedure consists of two main steps. Firstly, the verification engineer has to disambiguate the informal specifications written in natural language. Secondly, a formal specification language must be adopted to formalise the specifications into logical formulas that will be used to verify the design.

During the past decades, numerous approaches have been developed to perform verification with the above paradigm.

Moketar et al. [9] introduce an automated collaborative requirements engineering tool, called TestMEReq, to promote effective communication and collaboration between client stakeholders and requirements engineers for better requirements validation. The proposed tool is augmented with real time communication and collaboration support to allow multiple stakeholders to collaboratively validate the same set of requirements.

In [10] the authors describe a method to formalise specifications in a domain specific language based on regular expressions. The approach mainly consists in using a set of parallel non-deterministic Finite state machines to map formal specifications into behavioural models.

Subramanyan et al. [11] propose an approach to verify firmware security properties using symbolic execution. The paper introduces a property specifi-

cation language for information flow properties, which intuitively captures the requirements of confidentiality and integrity.

In [12], Buzhinsky presents a survey of the most popular existing approaches to formalise discrete-time temporal behaviours.

All the above works either use a standardised (such as PSL [13], SVA [14]) or a domain-specific formalisation language relying on temporal logic formalisms such as LTL (linear temporal logic) and CTL (computation tree logic). The LTL logic allows the formalisation of temporal behaviours unfolding on a single computational path; CTL is an extension of LTL which additionally allows branching time and quantifiers.

Once the informal specifications are thoroughly translated into logic formulas, automatic verification can be applied to the target design. The process of verifying a design using a set of formalised behaviours is called assertion-based verification (ABV); this technique aims at checking if the formalised behaviours hold in the design. ABV can be performed using model checking tools; although these procedures are capable either of proving that a property holds or generating a counterexample, they are not scalable, as they must explore the whole state-space of the design. To address the scalability problem, simulation-based approaches have been introduced to perform ABV. This techniques consist of simulating a design with a limited set of stimuli and memory configurations; therefore, they do not prove that properties hold for every possible computational path. To apply this verification model to a design, the verification engineer needs two additional elements aside from the assertions: a set of meaningful testbenches to stimulate and a virtual platform to simulate.

A set of significant testbenches is essential to thoroughly verify all functionalities of a design, to maximize its statement/branch coverage, and if possible, to discover hidden bugs.

Frattini et al. [15] address the topic of test-case generation by deepening into the possibility of generating a much more complete minimum set of stimuli for simulation-based verification.

In [16], the authors propose a self-tuning approach to guide the generation of constrained random testbenches using a sat solver. They employ a greedy search strategy to obtain a high-uniform distribution of stimuli.

Cadar et al. [17] present KLEE, a symbolic simulation tool capable of automatically generating tests that achieve high coverage for C/C++ programs.

In [18], the authors introduce a purely SAT-based semi-formal approach for generating multiple heterogeneous test-cases for a propositional formula.

"A Virtual Platform is a software based system that can fully mirror the functionality of a target System-on-Chip or board. These virtual platforms combine high-speed processor simulators and high-level, fully functional models of the hardware building blocks, to provide an abstract, executable representation of the hardware to software developers and to system architects" [19]. With a virtual platform, the DUV can be verified by injecting testbenches and by checking if the assertions hold during simulation. In this work, we generated a verification environment for the virtual platforms provided by IARSystem.

3 Methodology

As shown in Fig. 1, the proposed methodology is composed of four main steps executed sequentially. The input of MIST is a set of temporal behaviours generated in the first step of the methodology starting from informal specifications written in natural language. The output is a collection of files that need to be added to a target simulator to perform the verification of the design.

(1) Formalisation of specifications: The first step consists of translating the informal requirements into logic formulas. Initially, the user has to reinterpret the specifications into a set of cause/effect propositions, which naturally translate to logic implications $a \rightarrow c$. The user must fill in an XML scheme containing the implications, where each antecedent/consequent pair (a, c) is still written in natural language. After that, (a, c) pairs are formalised into formulas predicating on inputs/outputs and internal variables of the design under verification (DUV). To do so, the user uses an intuitive language of our craft to easily model complex temporal behaviours.

(2) Checker synthesis: In the second step, the tool parses the formalised specifications from the XML schema and generates a checker for each formula. Firstly, each formula is translated into a Büchi automaton. Secondly, a C/C++ representation of a corresponding checker is obtained from the automaton.

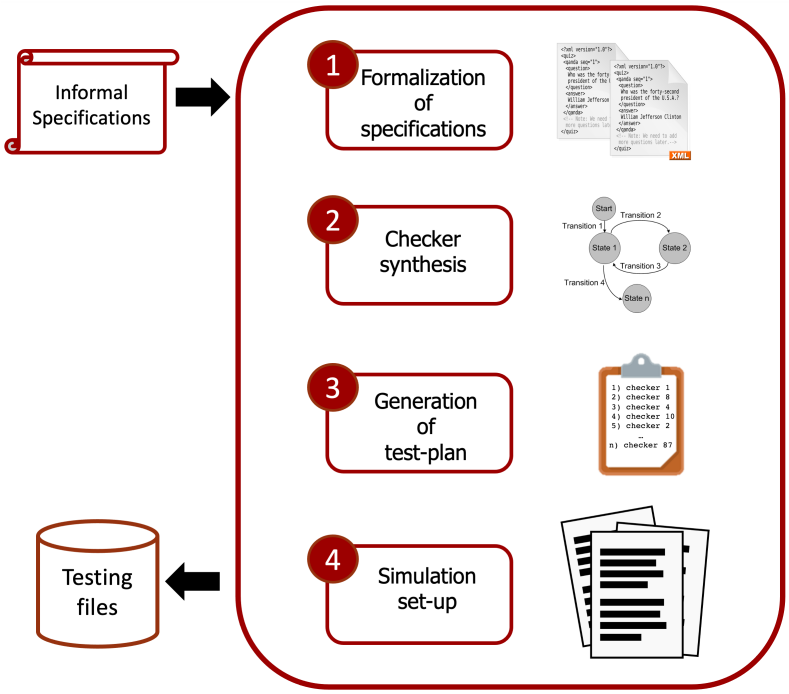

Fig. 1. Execution flow of MIST.

(3) Generation of test plan: The third step of the methodology aims at finding an effective verification order for the given specifications. Each behaviour must be verified when the firmware reaches a specific memory state that we call "precondition state", otherwise the verification would be vacuous. In this state, the behaviour can be verified by providing the proper stimuli. During the verification of a behaviour, the firmware changes to a new memory state that we call "postcondition state". Considering these assumptions, we identify a sorted list of behaviours that would connect each "postcondition state" to the "precondition state" of the following behaviour in the list to guarantee an effective verification process.

(4) Simulation set-up: In the last step, the tool generates all the files necessary to set-up the verification environment. This phase handles the architecture-dependent features of the employed simulator, such as time flow, interrupts and breakpoints. The output files can be described as follows:

- A set C/C++ source files implementing the checkers;
- A set of testbenches to stimulate the design;
- An orchestration file to verify each behaviour in the optimal "pre/postcondition" order;
- A set-up file to initialize the verification environment;
- A set of utility functions to handle the time flow and to manage the interrupts (if present).

Details related to the four steps implemented by MIST are reported hereafter.

4 Formalisation of specifications

In this section, we describe in detail how to employ our approach to formalise the specifications and to generate the testbenches. The process of formalisation consists of two subsequent steps. Firstly, the specifications are partially disambiguated using a high-level formalism. After that, they are completely formalised using our newly created language. If necessary, testbenches can be defined during formalisation.

4.1 High-level Formalisation

To clarify the whole procedure, we refer to the formalisation of the following example of specification:

"Firmware is in standard mode, boiler temperature is equal to 18°. Switches A and B are pressed or auto mode is active for at least 2000 ms, after that the boiler's temperature starts rising, then the firmware enters in comfort mode and sends an acknowledgment as output"

The user has to interpret the specification and translate it into a cause/effect behavior, which is represented by a high-level XML file as follows.

```
<assertion id=66>
  <precondition>
  Firmware is in standard mode, boiler temperature is
    equal to 18
  </precondition>
  <postcondition>
  Firmware is in comfort mode
  </postcondition>
  <antecedent>
  Switches A and B are pressed or auto mode is active
    for at least 2000 ms, after that the boiler's
   temperature starts rising
  </antecedent>
  <consequent>
  The firmware enters in comfort mode and sends an
   acknowledgement as output
  </consequent>
</assertion>
```

Listing 1.1. High-level specification

As depicted in the example, the high-level XML file consists of 5 tags:

- <assertion> contains the *id* attribute to uniquely identify the behavior;
- <antecedent> contains the antecedent part of the informal specification;
- <consequent> contains the consequent part of the informal specification;
- <precondition> contains the memory state the firmware must reach before checking the antecedent;
- <postcondition> contains the memory state reached by the firmware after the consequent has been successfully verified.

By performing this preliminary step, the user prepares the ground for the complete formalisation. Furthermore, the semi-formal specifications allow a better understanding of the quality of the informal specifications. Indeed, a specification that can not be formalised with the above pattern is either a non-functional specification or a poorly defined functional specification that must be clarified with the customer. This formalisation model could be even used directly during the initial interaction with the customer to guide the creation of a set of well-formed specifications from the beginning.

4.2 Low-level Formalisation

When the high-level XML file is completed, the user fills in the low-level XML file by adding unambiguous details to formalise the behaviors. To help non-expert in formal logic and temporal methods during the formalisation process, we defined a new language whose grammar is showed below.

```
assertion : antecedent -> consequent | precondition
   | postcondition
precondition : proposition
postcondition : proposition
antecedent : next_fragment
consequent : next_fragment
next_fragment : fragment | fragment; next_fragment
fragment : proposition [min, max, times, delay,
   forced, man_forced, until]
proposition : c_boolean_expression
```

Through this language, the user can formalise the specifications in forms of implications, where each antecedent/consequent is an ordered list of *fragments*. Each fragment contains a proposition p and a set of attributes specifying the temporal behavior of p. A proposition is a C/C++ boolean expression. From a temporal perspective, the verification of a consequent starts in the same instant in which the antecedent becomes true, and each fragment is evaluated one instant after the evaluation of the previous fragment completes. For example, in the implication $a \rightarrow c$, where a contains the sequence of fragments $[f_1; f_2; f_3]$ and c contains $[f_4; f_5]$: if f_1 holds in the interval $[t_0, t_n]$, f_2 evaluation starts at time t_{n+1}; on the contrary, if f_3 holds in the interval $[t_k, t_l]$, f_4 evaluation starts at t_l, since t_3 belongs to the antecedent while t_4 to the consequent. A fragment represents then a sequence of boolean events, similar to a PSL SERE [13]. Given a fragment f with a set of attributes
[*min*, *max*, *times*, *delay*, *until*] containing a proposition p, the semantics of the evaluation of f at time t_0 can be described as follows:

- **min** $= n$ with $n > 0$: f is true if p holds from t_0 to t_{n-1}. In other words, *min* attribute means that the proposition must remain true for a minimum of n instants.
- **max** $= n$ with $n > 0$: f is true if p becomes false before t_n. In other words, *max* attribute means that the proposition must remain true for a maximum of n instants.
- **times** $= m$ with $m > 0$ and **max** $= n$ with $n > 0$: f is true at time $t_k <= t_n$ if p holds for m (not necessarily consecutive) instants. If attribute *times* is set, then *max* must be set, while *min* and *until* are ignored.
- **delay** $= n$ with $n > 0$: f is true at time t_{n-1}.
- **until** $= q$ where q is a proposition, and **max** $= n$ with $n > 0$: f is true if q holds at time t_f with $t_0 \leq t_f \leq t_{n-1}$ and p holds from time t_0 to t_{f-1}. If attribute *until* is set then *max* must be set, while *min* and *times* are ignored.

To exemplify the use of the proposed language, we report hereafter the low-level XML resulting from the formalisation of the behavior previously used as a running example.

```
<assertion id=66>
  <precondition>
      mode == 0 && bTemp == 18.0
  </precondition>
  <postcondition>
      mode == 1
  </postcondition>
  <antecedent>
    <fragment min=2000 >
      (P0 == 0  && P4 == 16 && P12 == 4) || autoMode
    </fragment>
    <fragment until=bTmpRising max=9000>
        true
    </fragment>
  </antecedent>
  <consequent>
    <fragment min=1>
      mode == 1 && P16 == 1
    </fragment>
    <fragment min=1>
      (P16 >> 1) == 1
    </fragment>
  </consequent>
</assertion>
```

Listing 1.2. Low-level specification

The precondition (postcondition) is represented as a proposition identifying a concrete memory state that must be reached before (after) the verification of the behavior. In this example, the memory configuration identified by $mode == 0$ && $bTemp == 18.0$ is forced before checking the rest of the behaviour. The antecedent contains two fragments that, according to the described semantics, identify the following behavior: the first fragment is true if $P0 == 0$ && $P4 == 16$ && $P12 == 4$ || $autoMode$ holds true for 2000 consecutive instants; after that, the second fragment is true if $bTmpRising$ becomes true within 9000 instants. The consequent also contains two fragments. In the first fragment the proposition $mode == 1$ && $P16 == 1$ must be true for one instant. In the following instant, the second fragment is evaluated, and the proposition $(P16 >> 1) == 1$ must be true. From a temporal perspective, the antecedent is evaluated from time t_0 to t_k with $2000 < k < 11000$ while the consequent is evaluated from t_k to t_{k+1}.

4.3 Type system

In addition to the features described above, the propositions used in each fragment completely supports a C-compliant type system. In particular, variables can be defined using the usual C-styled syntax to declare their type. Moreover, the propositions support the explicit and implicit C type casting. Since the DUV

already contains the required declarations in the source code, the user needs only to spend few seconds to copy and paste them to the low-level XML file.

Furthermore, the user can declare debug variables to simplify the formalisation of complex behaviours. Debug variables are used during simulation but are held in memory outside the firmware under verification. This feature can be exceptionally useful to store intermediate values during the simulation of a behaviour. Listing 1.3 shows a possible declaration for the variables used in listing 1.2.

```
<declaration>
    unsigned char P0;
    unsigned char P4;
    unsigned char P12;
    unsigned char P16;
</declaration>
<assertion id=66>
  <declaration>
      int mode;
      float bTemp;
      bool bTmpRising;
      bool autoMode;
  </declaration>
  ...
</assertion>
```

Listing 1.3. Variables declaration

Note that we provide support for both global and local declarations. Local declarations are valid only inside the assertion in which they are defined; global declarations extend to all defined assertions.

4.4 Testbench generation

The formalisation language used in MIST provides three additional attributes: "**nTB**", "**forced**" and "**manual_forced**" to allow the generation of testbenches. The attribute **forced** can be specified for a fragment f to guide the testbench generator during the DUV simulation. If $forced = n$ with $n > 0$, MIST calls a SAT solver to generate a model for the proposition p that returns an assignment $var_i = val_i$ for each variable var_i included in p. If f is evaluated at time t_0, then each var_i is forced to value val_i in the interval $[t_0, t_{n-1}]$. The attribute nTB specifies how many testbenches must be generated for the current behaviour. If nTB is equal to p with $p > 1$, MIST generates p distinct test-vectors for the current fragment. If the number of available distinct test-vectors is less than p, MIST replicates the last generated test-vector to fill the empty spots.

```
    <FRAGMENT forced="200" delay="200">
            x || y
    </FRAGMENT>
```

Consider the example above, if $nTB = 4$ and $x||y$ is the proposition defined in the fragment, then there can exist only 3 distinct test-vector : (x = true,y = false), (x = false, y = true), (x = true, y = true). In this scenario, MIST replicates (x = true, y = true) to fill the fourth test-vector. Note that the attributes *forced* is completely independent of the evaluation of the fragment. If *forced* is the only attribute defined in the fragment, then the fragment is considered "empty"; nonetheless, a test-vector is generated anyway, but the evaluation of the empty fragment is skipped and the evaluation of the next fragment begins in the same instant (and not one instant later).

The attribute **manual_forced** follows the same semantic described for **forced**, except that the generated test-vector is manually provided by the user instead of being generated automatically. This is exceptionally useful in cases where the stimuli must vary in time or must follow a certain pattern. Moreover, the user could exploit this feature to integrate testbenches generated with specialised external tools, remarkably increasing the flexibility of MIST.

The syntax of manual_forced is slightly different: $manual\_forced = n$, where n is the id of a test-vector declared in the current assertion. Note that forced and manual_forced are mutually exclusive, only one of them can be used in a fragment at any time. A test-vector is defined with the following syntax:

```
<test_vector id="uInt">
    [var_1, var_2, ... , var_n] = {
            tv_tb_1;
            tv_tb_2;
               ...
            tv_tb_m;
    }
</test_vector>
```

$[var_1, var_2, \ ... \ , var_n]$ is the list of variables on which to apply the stimulus. tv_tb_i is the ith test-vector to be injected in the fragment when the simulator is stimulating the design with the ith testbench. Each tv_tb_i follows the syntax showed below.

$$
\begin{aligned}
&tv\_tb_i \ := \\
&(var_1\_val_1, \ var_2\_val_1, \ ..., \ var_n\_val_1, \ duration_1), \\
&(var_1\_val_2, \ var_2\_val_2, \ ..., \ var_n\_val_2, \ duration_2), \\
&\qquad\qquad ... \\
&(var_1\_val_k, \ var_2\_val_k, \ ..., \ var_n\_val_k, \ duration_n)
\end{aligned}
$$

Each tuple $(var_1\_val_j, \ var_2\_val_j, \ ... \ , \ var_n\_val_j, \ duration_j)$ identifies a piece of test-vector where the variables $var_1, var_2, \ ... \ , var_n$ are forced with the values $var_1\_val_j, \ var_2\_val_j, \ ... \ , \ var_n\_val_j$ for $duration_j$ instants. Once the values are injected for $duration_j$ instants, the following tuple $(j + 1)$ is used to inject the values.

In the example depicted in listing 1.2, we assumed that pressing the bottom and rising the temperature were internal events of the firmware that did not require any external stimulus. However, in many cases this is not true; usually, the user has to provide as input a sequence of stimuli to test the correct behaviour.

In the example below, we propose again the same formalised behaviour where the fragments of the antecedent are used to inject testbenches. Note that the consequent is the same of listing 1.2.

```
<assertion id=66 nTB=2>
  <precondition>
      mode == 0 && bTemp == 18.0
  </precondition>
  <postcondition>
      mode == 1
  </postcondition>
  <antecedent>
    <fragment forced=2000 delay=2000>
      (P0 == 0  && P4 == 16 && P12 == 4) || autoMode
    </fragment>
    <fragment man_forced=7 delay=1200/>
  </antecedent>
 <test_vector id=7>
   [bTemp] ={
   (18.0,200),(18.2,200),(18.4,200),(18.6,200)
   ,(18.8,200),(19.0,200); }
 </test_vector>
</assertion>
```

Listing 1.4. Low-level specification with testbenches

In this example, there are both automatic and manual test-vectors.

Since nTB=2, MIST generates two testbenches.

In the first fragment of the antecedent, the generated test-vector is (P0 = 0, P4 = 16 , P12 = 4, autoMode = false) for the first testbench and (P0 = 0, P4 = 0 , P12 = 0, autoMode = true) for the second testbench. The second fragment contains a manual test-vector with ID equal to 7. We also use the attribute "delay" to postpone the evaluation of the second fragment after injecting the test-vector of the first fragment. Likewise, we put off the evaluation of the consequent by delaying the second fragment. If we combine the automatic test-vector of the first fragment with the manual test-vector of the second fragment, MIST generates the following testbenches:

1. (P0 = 0, P4 = 16 , P12 = 4, autoMode = false) for 2000 instants, (bTemp = 18.0) for 200 instants, (bTemp = 18.0) for 200 instants, (bTemp = 18.2) for 200 instants, (bTemp = 18.4) for 200 instants, (bTemp = 18.6) for 200 instants, (bTemp = 18.8) for 200 instants, (bTemp = 19.0) for 200 instants
2. (P0 = 0, P4 = 0 , P12 = 0, autoMode = true) for 2000 instants, ... the rest is the same of the previous testbench.

Note that in the second testbench, the test-vector for the second fragment is the same used for the first. This happens because only one test-vector was defined for the second fragment while 2 were needed to generate the required testbenches.

From a temporal point of view, the two tesbenches can be represented as in figure 2. Both testbenches are injected from time t_0 to time t_{3199}.

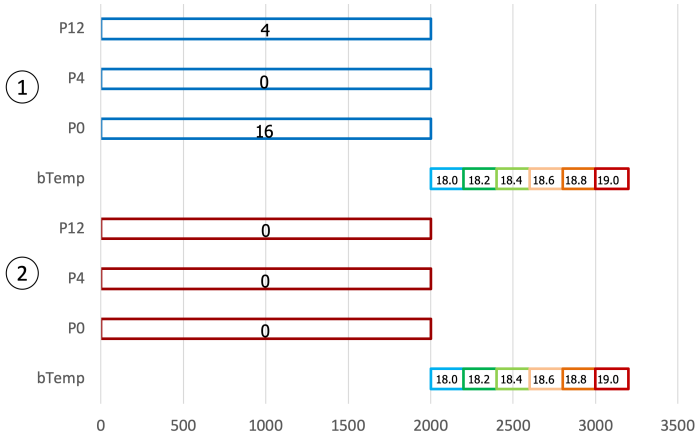

Fig. 2. Testbenches timeline

5 Checker synthesis

In the second step of the methodology, MIST parses the formalised specifications in the low-level XML files and generates a C/C++ checker for each implication. The process works in three main sub-steps. Firstly, the tool translates each XML assertions to a PSL formula. Secondly, each PSL formulas is used to generate its equivalent Büchi automaton. Finally, the Büchi automaton is translated to C/C++.

We treat each implication as two independent formulas, one for the antecedent and one for the consequent. This separation is necessary to pinpoint scenarios where the implication is vacuously true. If we considered the implication as a whole, a true evaluation could either mean that the consequent was true or the antecedent was false, we want to distinguish both cases to better warn the user. To convert an XML assertion to PSL, each sequence of *fragments* is treated as a PSL SERE. For example, the consequent of the specification used in Section 4 translates to the following PSL formula $\{mode == 1 \;\&\&\; P16 == 1;\; (P16 >> 1) == 1\}$.

Since the PSL syntax does not allow the use of many C operators such as the bit shift operator (<<), we execute an intermediate step to provide support to all C operators that can be used to form a boolean expression. In this step, the tool substitutes each fragment's proposition with a placeholder boolean variable representing the proposition. For example, the above formula would be translated to $\{ph1;\; ph2\}$ where $ph1$ is the placeholder for $mode == 1 \;\&\&\; P16 == 1$ and $ph2$ is the placeholder for $(P16 >> 1) == 1$; Once the translations above are completed, we generate a Büchi automaton for each formula. To do so, we use spotLTL [20], an external library capable of generating automata from LTL/PSL formulas. Finally, the resulting automaton is visited to generate a C/C++ implementation of the corresponding checker.

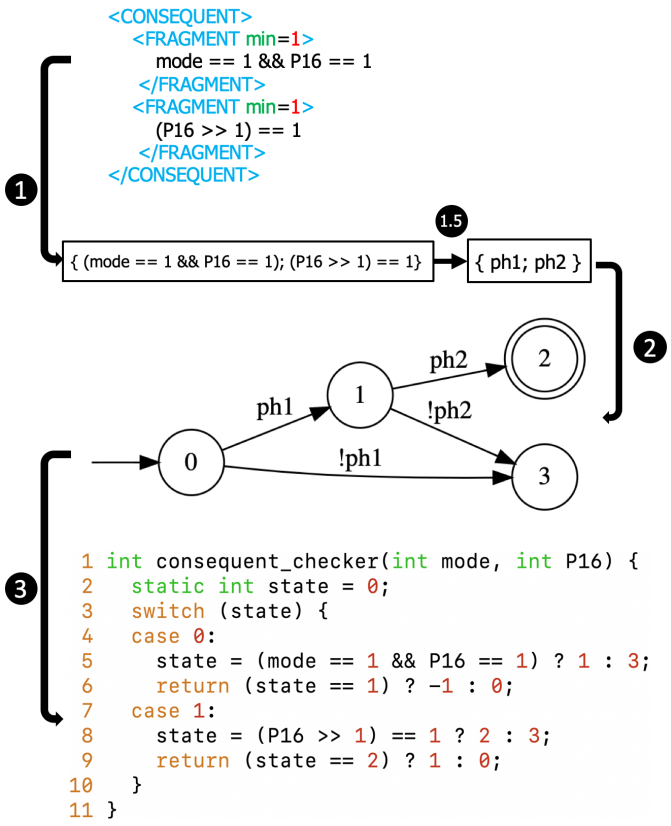

Fig. 3. Example of checker synthesis.

Fig. 3 shows an example to clarify the process. In steps (1) and (1.5), the fragment is converted to PSL, and its proposition is substituted with placeholders according to the aforementioned procedure. In step (2), the LTL formula is given as input to spotLTL to generate the depicted Büchi automaton. Before synthesizing the C/C++ checker, each placeholder is substituted back to its original proposition. In Fig. 3, placeholder $ph1$ and $ph2$ are substituted back to $mode == 1$ && $P16 == 1$ and $(P16 >> 1) == 1$. In step (3), the automaton is visited starting from the first state. For each state, the tool generates a *case* of a C *switch*, for each edge the tool generates the next-state function in each case. Note that the accepting (rejecting) state is optimized away. For example, the generated checker contains a *case* in which *state* is equal to 0. In this case, if the condition $mode == 1$ && $P16 == 1$ is satisfied then *state* is changed to 1, otherwise it is changed to 3. In this scenario, states 2 and 3 are respectively the accepting and rejecting states where the checker returns 1 (true) and 0 (false). In all other states, the checker returns -1 (unknown).

6 Test plan generation

In the third step of the methodology, the low-level XML file is used to generate an effective testing order. Such an order is intended for generating testbenches that make the firmware evolve in the right memory state before the verification of a behaviour is performed. Otherwise, the checker may pass vacuously or fail due to a wrong precondition state reached by the firmware when the checker is executed.

MIST can generate a test plan following two different strategies: a guided and an unguided strategy. The unguided strategy does not leverage the information provided by the posconditions to generate an effective testing order; therefore it is more prone to errors. On the other hand, since it does not require the definition of postconditions, it is easier to use. Inexperienced users should become confident with this first strategy before exploring the more sophisticated second one. The guided strategy makes full use of the postconditions to reduce unexpected failures of checkers due to formalisation mistakes. Furthermore, it provides feedback on the quality of the formalised specifications.

6.1 Unguided test plan generation

This procedure can be used to quickly generate a test plan without exploiting the relation between preconditions and postconditions. Although it is less secure, it might be more preferable for developers who do not want to put in the extra effort of applying the guided approach.

First, the user has to define a safe condition and a set of behaviours. After that, MIST automatically generates a test plan operating as follows. During the simulation, the verification process waits until the safe condition is satisfied. Then, the verification process stores the current firmware memory; this memory state is called "safe state". From there on, the following algorithm is executed:

1. Pick an untested behaviour (b_i); if all behaviours are tested, this process ends.
2. Load the safe state in the firmware's memory.
3. Force the precondition pr_i of b_i to be true in the current simulation, if pr_i does not hold after being forced, prompt an error and return to 1.
4. Test b_i using testbench tb^i_j and dump the result of the test in the verification report.
5. If j is the index of the last testbench of b_i, then go to 1, else, increment j and return to 2.

The safe condition is a non-temporal boolean expression following the same semantics of a fragment proposition. If it becomes true during simulation, it prompts the beginning of the verification process. Delaying the verification process until the safe condition is satisfied allows the simulation to perform a proper initialisation of the firmware; this step is mandatory for most implementations before testing any functional behaviour. A precondition is forced following a

similar procedure to the one used to force a proposition inside a fragment. Once again, we use a sat solver to identify an assignment of variables that satisfies the proposition, this assignment is then forced during the simulation.

Dumping and loading safe states are inexpensive procedures both computationally and memory wise. This is true because only a small writable part of the firmware's memory is dumped, as it is the only portion of memory that could change during execution, the rest remains unchanged for all simulations. Furthermore, only one safe state needs to be stored to make this approach work.

6.2 Guided test plan generation

The unguided test plan generation already provides a quick and simple approach to enable verification using MIST. However, to apply that procedure correctly without mistakes, the user would have to annotate each formalised behavior with the *exact* memory state to be forced before starting the test. This process can be extremely time-consuming and error-prone; as a matter of fact, to be sure of reaching the correct memory configuration, the user might have to address in the precondition the value of all variables used in the firmware, which could be thousands of variables in most industrial firmware. In many cases, errors in this procedure lead to a vacuous verification; the test is unable to fire the antecedent of the target assertion, as the testbench is injected in the wrong memory configuration. In this situation, the verification engineer would have to go through an excruciating process of trial and error to find the correct precondition.

To address this issue, we developed a guided test plan generation, to produce an effective testing order. This procedure relies on the assumption that the DUV was developed by following a coherent logic flow. The generated testing order tries to mimic the behavior of a human that manually tests the DUV. To check the correctness of a design, the human starts from the initial state and provides a sequence of stimuli to the DUV. Each sequence of stimuli moves the DUV from one configuration to the next in a coherent flow, such that the ending configuration represents the starting precondition for effectively checking the next behavior in a cause-effect cascade fashion. Through this approach, the specifications are verified in the order intended by the designer, thus reducing the necessity of forcing the memory state that represents the precondition of the target behavior, since the DUV gets naturally brought to the proper state. In other words, the verification engineer no longer has to regard the whole memory of the firmware in the precondition; the correct memory configuration is partially reached as a "side-effect" of the previously tested behaviours.

The guided test plan generation consists of two main procedures. Firstly, all assertions formalised in the low-level XML file are divided into subsets through a clustering procedure. Secondly, each subset is treated as a node of a multilevel graph, and a verification order is defined by generating a path that connects all nodes. Such a path is then traversed to generate an effective testing order.

In this procedure, we consider the precondition and postcondition tags of each assertion. Each precondition/postcondition consists of a propositional formula following the template $variable_1 == constant_1$ & $variable_2 == constant_2$ & ... &

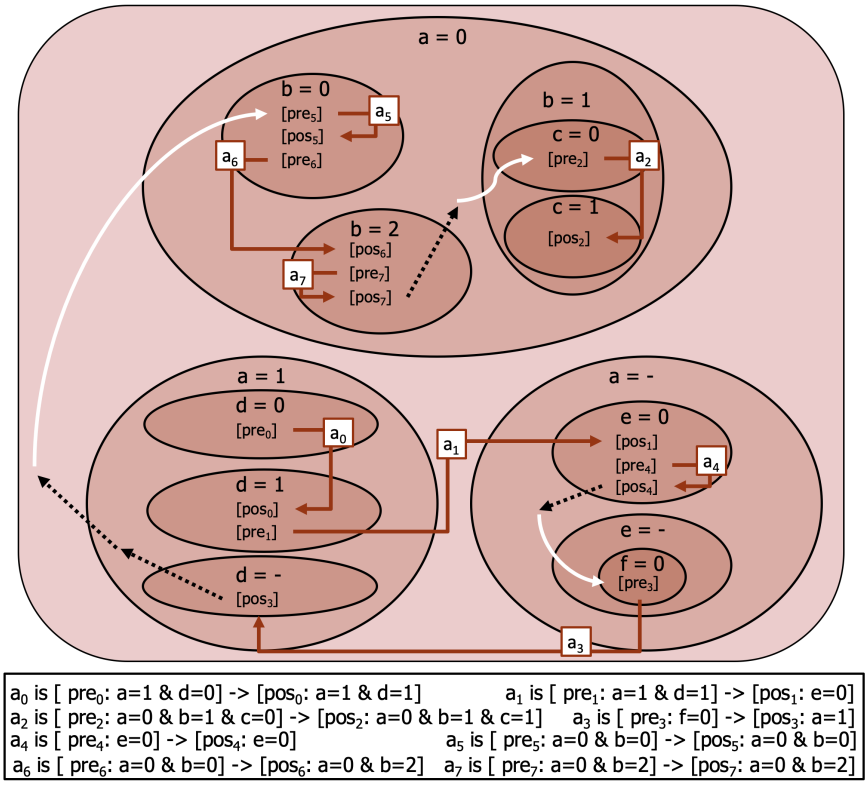

Fig. 4. Example of test plan generation

$variable_n == constant_n$ that represents a concrete memory configuration. To simplify the exposition, we will use the term "memory state" while referring to a precondition/postcondition.

In the clustering phase, the goal is to divide the set of all memory states into subsets. We will refer to the example depicted in Fig. 4 to clarify the procedure. At the bottom of Fig. 4 we report the list of assertions used in the example. For instance, the assertion described in Section 4 is represented in the example as "a_{66} is $[pre_{66} : mode = 0] -> [pos_{66} : mode = 1]$", where *pre (pos)* is the precondition (postcondition) of the assertion with *id* equal to 66. The clustering process starts by considering the whole set of memory states, and then it is recursively repeated for each generated sub-set until no set can be further divided. The process counts the occurrences of each variable in all memory states in the current set; the variable with the highest count is used to perform the split. In the example, the most frequent variable in the whole set is a. The current set is split into as many sub-sets as the number of different assignments of the most frequent variable. Also, we add an optional sub-set containing all memory states that do not include the most frequent variable (do not care sub-set). In

the example, the whole set is divided into three clusters, two clusters for $a = 0$ and $a = 1$ and one don't care cluster $a = -$. The same process is repeated until all sub-sets contain only memory states with equivalent assignments. In the example, the cluster identified by $a = 0$ and $b = 0$ contains three equivalent memory states $[pre_5]$, $[pos_5]$, $[pre_6]$ that have the same assignments $[a = 0 \ \& \ b = 0]$. This heuristic approach is intuitively justified by the assumption that the most frequent variables represent better the whole state; therefore, it is reasonable to make them represent wider clusters than those represented by less frequent variables. The clustering procedure aims at making all similar memory states "close" to each other.

In the second part of the approach, each sub-state is used to infer an effective testing order. Starting from the precondition of an assertion chosen randomly (or by the user), the tool finds a path that covers all the memory states. To move from one memory state to the next, the procedure applies the following rules:

R1: Checking an assertion i in memory state $[pre_i]$ moves the process to $[pos_i]$ (solid red arrow);

R2: If the process can not find any other unused precondition in the current state cluster, it must jump to its upper cluster and continue the search (dotted black arrow);

R3: After a jump, the process searches for the first unused precondition $[pre_j]$ in the current cluster. If it finds one, it continues the process from that state (rounded white arrow).

To clarify the procedure, we explain the process by considering the example of Fig. 4. In this example, the user chooses to start with assertion a_0; therefore, the starting state is $[pre_0]$. By applying rule R1, assertion a_0 is added to the test plan, and the execution moves to state $[pos_0]$. In the destination cluster, we find an unused precondition $[pre_1]$. We apply again rule R1, assertion a_1 is added to the test plan, and the execution is moved to pos_1. We repeat the process for assertion a_4, and we reach the state pos_4. In this case, no more preconditions are available in the current cluster; therefore, the execution must apply rule R2 and jump to the upper cluster identified by $a = -$. By applying rule R3, the process finds an unused precondition pre_3 and continues from there. Again, we add assertion a_3 to the test plan, and we move the execution to pos_3. We apply rule R2 as no other preconditions can be found in the current cluster, and we reach cluster $a = 1$. We must apply rule R2 again for the same reason and jump to the upper cluster. The procedure continues as described above until all assertions are added to the test plan. The resulting test plan is $[a_0, \ a_1, \ a_4, \ a_3, \ a_5, \ a_6, \ a_7, \ a_2]$.

Note that the ideal case, where all behaviors described by the initial specification perfectly connect to form a coherent path, requires the user to completely formalise the specifications such that all assertions belong to a unique cluster. This requirement could be extremely tedious to achieve manually and could be unfeasible for most large-scale designs. For this reason, each time we identify a hole in the specification, such that the postcondition of an assertion does not connect with the precondition of any other assertion, our heuristic approach jumps to a similar close state and warn the verification engineer. To be clear, in

Table 1. Completeness analysis for example in Fig. 4

max applications of rule R2	completeness
0 times	62.5%
1 times	87.5%
2 times	100 %

the case of fully connected specifications, our approach uses only rule R1. Each time rules R2 and R3 are used, we are approximating.

After generating the test plan, MIST informs the user of the *completeness* of the given set of behaviors by comparing the total number of assertions with the number of times rule R2 was applied to continue the clustering process. The completeness index is calculated with the following formula:
$(1 - exceeded\_maxR2\_applications \ / \ tot\_assertions)$.
Where $exceeded\_maxR2\_applications$ represents the number of times the process has to violate the maximum number of consecutive applications of rule R2. Intuitively, the resulting completeness is an index describing how much the set of behaviors is likely to cover all functionalities of the DUV without holes. Each time a missing link is found, the completeness is reduced.

Table 1 shows the completeness for the running example. The first row of the table shows the completeness when no approximation is allowed, or in other words, when the process should not use rule R2 to continue. In the example, rule R2 is used 3 times non-consecutively; therefore, the resulting completeness is $(1- 3/8) = 0.625$. In the example, the second (third) row shows the completeness reachable by allowing the consecutive application of rule R2 at most once (twice).

The user can exploit this information to improve the set of formalised behaviors such that rule R2 is applied as less as possible while achieving high completeness.

7 Simulation setup

7.1 Setup

In the last step of the methodology, the verification environment is set up. This phase handles the architecture-dependent features of the target simulator. For now, MIST is capable of generating a verification environment for the IARsystem workbench, which is an industrial compiler and debugger toolchain for ARM-based platforms. In particular, we exploit the provided breakpoint system to evaluate the checkers and to handle the time flow.

Since our checkers provide support for temporal behaviors, we need a way to sample the time flow. To accomplish that, we provide a debugging variable *sim_time* that can be used by the user to simulate the advancement of time in the DUV. To capture this event in the debugger, we place a breakpoint on that variable to recognize *write* operations. Each time *sim_time* is incremented, the

simulated time advances by one instant producing a re-evaluation of the active checker. Usually, the best way to use sim_time is to place it in a timed interrupt that keeps increasing it at a constant rate. Furthermore, we use breakpoints to inject stimuli in the ports and variables of the fragments using the $forced$ and $manual\_forced$ attributes. Following the above mechanisms, MIST generates the files to perform the verification of the DUV using IARsystem. The generated files consist of an entry point to set up the verification environment, utility functions to handle the time events, the orchestration file that executes each checker using either a guided or unguided strategy and a set of files containing the checkers. To integrate the generated verification environment with IARSystem, the user only has to provide the MIST's entry point file to the simulator; after that, the verification process proceeds automatically until its completion.

7.2 Report

```
[CHECKER #66_a]
FRAGMENT FALSE in Antecedent, Fragment 2
-> Testbench none
    - Proposition "true until bTmpRising, max = 9000" is false!
    - Reason: timer ran out!

[CHECKER #66_b]
FRAGMENT FALSE in Consequent, Fragment 1
-> Testbench 2
    - Proposition "mode == 1 && P16 == 1" is false!
    - Reason: mode = 1, P16 = 4 after 0 instants of <min,1>

################################
########### SUMMARY ############
################################
- Number of tests: 10
- Test plan order [checker id, nTB]: [1, 1] [1, 2] [66_a, none]
 [66_b, 1] [66_b, 2] [2, 1] [3,1] [3,2] [3,3] [3,4]
- Verified : 8 [80%]
- Vacouse : 1 [10%]
- Failed : 1 [10%]


```

Fig. 5. Example of report

Once all behaviours are tested, the verification process provides a verification report containing the results of the simulation. The report includes information related to the coverage and failure of checkers, together with the applied test-benches. Checkers whose antecedent was false are reported as vacuously satisfied; otherwise, they are either reported as "verified" if the consequent was true, or as "failed" if the consequent was false.

Since our formalisation language has a well-structured and simplified syntax, failed checkers are also capable of reporting additional information about the failure. Not only they can report exactly the location of the failure in the behaviour, but they can also infer its cause. We show an example of a verification report in fig. 5.

In this example, we show the result of two possible failures for the running examples depicted in listing 1.2 (66_a) and 1.4 (66_b). In particular, 66_a is vacuously verified, as the failure makes the antecedent false; 66_b fails on testbench 2, as the failure occurred in the consequent. All other behaviours are correctly verified for all testbenches. The verification report is composed of two main parts, the first part contains the details of the failures, while the second part contains the summary of the whole simulation. For each failed test, the verification environment is capable of reporting the exact location of the failure. For behaviour 66_b, it is reported that the failure occurred in the first fragment of the consequent while injecting the second testbench. Thanks to the limited number of temporal operators and a well-defined structure of the propositions, we can provide a custom message for each failure, greatly simplifying the understanding of its cause. These messages usually contain the assignment of variables that made the proposition fail together with additional remarks on the applied temporal operator. By reading the message for behaviour 66_b, we can quickly understand the cause of the failure: the assignment of variables $mode = 1$, $P16 = 4$ clearly does not satisfy proposition $mode == 1$ && $P16 == 1$. In particular, variable $P16$ is the cause of the failure. Furthermore, the message "after 0 instants of $\langle min, 1 \rangle$" warns the user when the proposition became false, that is, in the first instant of evaluating a fragment annotated with the *min* attribute.

8 Experimental results

The experimental results have been carried out on a 2.9 GHz Intel Core i7 processor equipped with 16 GB of RAM and running Windows 10.

8.1 Case study

We evaluated the effectiveness of our tool to verify an industrial firmware composed of over 10000 lines of C code. The analyzed case study is represented by firmware implementing the controller of a boiler implant. The user can interact with the firmware through an HMI (Human machine interface) composed of LCD display and 4 alphanumeric digits, 7 keys, an RS485 connection and 1 TTL connection (possibility of a second modbus with the addition of the ITRF14 interface). Moreover, the firmware is connected to several external devices providing inputs/outputs such as thermostats, boilers, clocks and an internet gateway. The firmware runs on an RL78 microcontroller, allowing communications with the external devices through Modbus and I2C protocols. Finally, the internal time flow is handled using timed interrupts. The case study configuration is depicted in fig. 6.

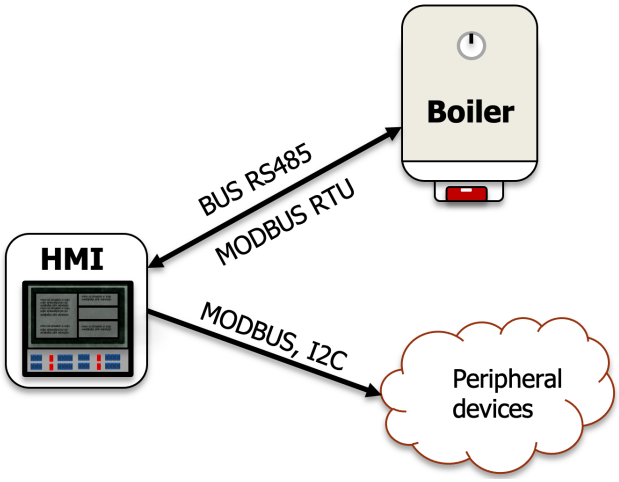

Fig. 6. Case study

8.2 Results

We put emphasis on the timing results of the complete verification process, from the formalisation of specifications to the simulation of the behaviours. Starting from the informal specification of the firmware, we formalised 100 behaviours. On average, each behaviour takes 30 seconds to be formalised into the high-level XML format. The formalisation of the low-level XML format depends significantly on the skill of the verification engineer and his/her knowledge of the underlying implementation details. After some practice, we were capable of formalizing a behaviour in less than three minutes. Overall, we formalised all 100 behaviours in less than 6 hours. After that, MIST generated the testing files and produced an effective test plan in less than 10 seconds. We don't report numerical results proving the scalability of the tool in terms of time/memory as the complexity of the approach is linear with respect to the number of formalised behaviours; therefore, the tool might take minutes at most to formalise thousands of behaviours. Finally, we set-up the verification environment in the simulator (IAR System Workbanch). The simulation took less than 40 minutes to verify non-vacuously each behaviour and to produce a report of the verification.

The employment of our methodology to an industrial legacy firmware discovered numerous bugs related to an inaccurate sampling of time. One notable example concerns the usage of switches in the HMI. Many specifications implied that some switches needed to be pressed for a certain amount of time to active a functionality. However, during simulation, the correct behaviour did not occur even when providing the correct stimuli. Using MIST for the verification of such a firmware was considerably helpful in identifying a temporal inconsistency of Modbus and I2C protocols that caused a delay in its execution.

Table 2. Completeness analysis for the considered case study.

max applications of rule R2	completeness
0	45.5%
1	72.73%
2	79.22%
3	81.82 %
4	97.73%
5	100%

Table 3. Completeness analysis of the case study after the improvements.

max applications of rule 2	completeness
0	48.5%
1	75.73%
2	88.2%
3	100 %

Furthermore, the generation of the test plan for 100 behaviours suggested a remarkable incompleteness in the firmware specifications. In table 2 we can observe the completeness estimations produced for the case study by considering the approach proposed in Section 6. We used those statistics to improve the completeness of the specifications by adjusting the behaviours underlining the highest incompleteness and by adding 10 behaviours to cover some specifications holes. After completing this procedure, we achieved new completeness estimations reported in Table 3. To achieve 100% completeness with the new specifications, we needed to apply rule R2 only 3 times, while with the initial specifications, it was used 5 times.

To test the effectiveness of the new language developed for MIST, we arranged a 2-day workshop with the company that provided the industrial case study. In this short time, the developers have been capable of quickly grasping the fundamentals of the language, and before long, they have begun formalising specifications and using the tool on their own.

9 Conclusions and future works

In this paper, we presented MIST, an all-in-one tool capable of generating a complete environment to verify C/C++ firmwares starting from informal specifications. MIST reduces the verification effort by providing a user-friendly interface to formalise specifications into assertions and to generate the verification environment automatically. Furthermore, MIST employs a clustering procedure to generate an effective test plan that reduces potential mistakes while formalizing the specifications.

Collaborating with the industry gave us the opportunity to make the tool go through a long tuning process. Moreover, the feedback received by experi-

enced developers allowed us to thoroughly assess the potentials and limitations of MIST. The majority of limitations were overcome during the tuning process; however, there are still few issues that need to be addressed in future works. Most drawbacks of the verification environment generated by MIST are related to unjustified constraints imposed by C-Spy, which is the debugger used in the IARSystem Workbench. Below, we report some of those constraints.

- No observability of non-static variables: we can not test the value or put breakpoints on automatic variables, therefore, we can not write assertions with those variables
- Macros declared with the "#define aliasName originalName" C statement are not visible during simulation: the user is forced to use the right side of the macro when writing propositions, as the debugger does not keep track of aliasing. This limitation deeply affect the readability of the formalised behaviours.
- Lack of strongly typed variables and complex C data structures in the C-Spy language: this major constraint strongly affected the development of MIST; furthermore, we believe that it will also heavily affect extensibility and maintainability.

To avoid being dependent on the constraints imposed by a specific simulator, we will modify the back-end of MIST to be easily extendable to other target simulators.

Hereafter, we report some limitations of MIST that we would like to overcome in future releases.

- No support to generate testbenches that affect only a portion of bits of a target variable: consider the variable *unsigned char P*0, for now, the user can not generate a testbench that, for instance, would modify the value of the first bit of P0 while keeping the other bits unchanged. We planned to introduce a custom operator to overcome the above limitation.
- All behaviours are linked to the same temporal event: we would like to have the user define what temporal event should produce the advancement of time inside each behaviour.

MIST is an open source project (GNU license) freely available at `https://gitlab.com/SamueleGerminiani/mist`.

References

1. P. K. Shamal, K. Rahamathulla, and A. Akbar, "A study on software vulnerability prediction model," in *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, 2017, pp. 703–706.
2. N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," 06 2005, pp. 580– 586.
3. M. Dawson, D. Burrell, E. Rahim, and S. Brewster, "Integrating software assurance into the software development life cycle (sdlc)," *Journal of Information Systems Technology and Planning*, vol. 3, pp. 49–53, 01 2010.

4. M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security Privacy*, vol. 7, no. 2, pp. 87–90, 2009.
5. M. Jørgensen, K. Teigen, and K. Moløkken-Østvold, "Better sure than safe? overconfidence in judgment based software development effort prediction intervals," *Journal of Systems and Software*, vol. 70, pp. 79–93, 02 2004.
6. T. D. Oyetoyan, B. Milosheska, M. Grini, and D. Cruzes, *Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital*, 05 2018, pp. 86–103.
7. M. H. Osman and M. F. Zaharin, "Ambiguous software requirement specification detection: An automated approach," in *2018 IEEE/ACM 5th International Workshop on Requirements Engineering and Testing (RET)*, 2018, pp. 33–40.
8. [Online]. Available: https://www.iar.com/iar-embedded-workbench
9. N. A. Moketar, M. Kamalrudin, S. Sidek, M. Robinson, and J. Grundy, "An automated collaborative requirements engineering tool for better validation of requirements," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 864–869.
10. Y. Kakiuchi, A. Kitajima, K. Hamaguchi, and T. Kashiwabara, "Automatic monitor generation from regular expression based specifications for module interface verification," in *2005 IEEE International Symposium on Circuits and Systems*, 2005, pp. 3555–3558 Vol. 4.
11. P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Architecture of a tool for automated testing the worst-case execution time of real-time embedded systems firmware," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 337–342.
12. I. Buzhinsky, "Formalization of natural language requirements into temporal logics: a survey," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, 2019, pp. 400–406.
13. "Ieee standard for property specification language (psl)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010.
14. "Ieee standard for systemverilog–unified hardware design, specification, and verification language - redline," *IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline*, pp. 1–1346, 2009.
15. S. Yang, R. Wille, and R. Drechsler, "Improving coverage of simulation-based verification by dedicated stimuli generation," in *Formal Methods in Computer Aided Design*, 2014, pp. 599–606.
16. Y. Zhao, J. Bian, S. Deng, and Z. Kong, "Random stimulus generation with self-tuning," in *13th International Conference on Computer Supported Cooperative Work in Design*, 2009, pp. 62–65.
17. C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.
18. S. Agbaria, D. Carmi, O. Cohen, D. Korchemny, M. Lifshits, and A. Nadel, "Sat-based semiformal verification of hardware," in *Formal Methods in Computer Aided Design*, 2010, pp. 25–32.
19. [Online]. Available: https://www.esa.int
20. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0 — a framework for ltl and ω - automata manipulation," 10 2016, pp. 122–129.