UNIVERSITY OF VERONA

DOCTORAL THESIS

A complete assertion-based verification framework from the edge to the cloud

Author:
Samuele Germiniani

PhD Advisor:
Prof. Graziano PRAVADELLI

Department of Computer Science

May 26, 2023

A complete assertion-based verification framework from the edge to the cloud — SAMUELE GERMINIANI
PhD Thesis
Verona, May 26, 2023
ISBN <ISBN>

Abstract

Samuele Germiniani

A complete assertion-based verification framework from the edge to the cloud

Assertion-based verification (ABV) is a well-known approach for checking the functional correctness of a system. Since modern cyber-physical systems are increasingly complex and distributed, it is no longer appropriate applying ABV only to the single components; instead, it is necessary to embrace holistic approaches that look at the entire system.

Furthermore, due to the dynamic nature of the system under verification (SUV), ABV cannot be applied only in an offline fashion. Alternatively, it is necessary to extend the verification process to the post-deployment phase; however, this collides with the issues of dealing with a distributed system affected by unpredictable latency and providing limited computational resources. Therefore, it becomes essential to develop a dynamic orchestration approach where checkers perform runtime verification without negatively influencing the computation of the functional parts of the SUV.

To fill in the gap, I propose a complete framework to verify complex distributed systems, from the formalisation of specifications to runtime execution. The proposed framework aims at covering several holes in the verification process of systems executing in an edge-to-cloud computing environment.

Contents

List of Figures

List of Tables

List of Abbreviations

ABV	Assertion-Based Verification
AT	Approximation Tokens
COM	COMmutative operators
CPS	Cyber-Physical System
CTL	Computation Tree Logic
DLL	Double Linked List
DPDG	Dynamic Program Dependency Graph
DT	Decision Tree
DTA	Decision Tree Algorithm
DUE	Design Under Exploration
DUV	Design Under Verification
HK	Hierarchical Clustering
HT	Hardware Trojan
IG	Information Gain
KDE	Kernel Density Estimation
LLVM	Low-Level Virtual Machine
LTL	Linear Temporal Logic
MCB	Main Control Board
MILP	Mixed-Interger Linear Programming
NAT	Network Address Translation
NR	Non-Reflexive operator
PDG	Program Dependency Graph
PIT	Potentially Instantiated Template
PSL	Property specification Language
PTP	Precision Time Protocol
ROS	Robot Operating System
RV	Runtime Verification
RVE	Runtime Verification Environment
SERE	Sequential Extended Regular Expression
SSIM	Structural SIMilarity index
STL	Signal Temporal Logic
SUV	System Under Verification

Chapter 1

Introduction

Verification is the process of checking if an electronic system satisfies the designer's intent, namely if it complies with a set of predetermined requirements. The system under verification (SUV) is usually analysed through manual inspection or by using more sophisticated approaches involving automatic tools; each time an unwanted behaviour is found, the corresponding bug is fixed. In the last few decades, verification has become one of the most crucial aspects of developing cyber-physical systems (CPS) where physical and software components are deeply intertwined. Thoroughly verifying the correctness of a CPS often leads to identifying bugs and specification holes far earlier in the deployment process, exempting the developing company from wasting resources on costly maintenance. Unexpected bugs can become exceptionally expensive when they are intentionally used to exploit vulnerabilities or when they cause accidental failures.

Ideally, the verification process should be carried out purely by employing automatic tools. However, state-of-the-art techniques still require plenty of manual efforts to I) translate informal requirements written using natural language (such as English) to unambiguous formal specifications (such as logic formulas); II) detect bugs (when the system does not meet the requirements); III) understand and fix the detected bugs.

Since modern CPSs are increasingly complex and distributed, it is no longer appropriate to focus the verification process only on the single components; instead, It is necessary to embrace holistic approaches that look at the entire system. To this end, it is crucial to consider an ecosystem of integrated tools interconnected in a complete supply chain: from the formalisation of specifications to runtime verification. Even though several tools have been proposed, there is no single framework that can be considered an integrated ecosystem. This leads to a number of inefficiencies and holes in the verification process.

In this context, assertion-based verification (ABV) is a well-known approach for checking the functional correctness of a system. In ABV, the specifications of the system are formalised through assertions, which are logic properties that should hold during the system's execution. Due to the complexity and dynamic nature of the SUV, ABV cannot be applied only in an offline fashion before the deployment of the system: the typical SUV in an industry 5.0 scenario adds the complexity of non-deterministic interaction between people and machines at runtime. Therefore, it is necessary to extend the verification process to the post-deployment phase, that is, by running checkers (pieces of code verifying the functional behaviours of the system) during the execution of the system. However, this collides with the issues of dealing with a distributed system affected by unpredictable latency where the SUV is made of several components with limited available resources; to make things even more challenging, these resources are usually already completely saturated from executing the functional tasks. As a consequence, it becomes essential to develop

a dynamic orchestration approach where checkers are allowed to perform runtime detection of critical (in terms of safety and functionality) and suspicious (in terms of security) situations without negatively influencing the computation of the functional parts of the SUV.

To fill in the gap, I propose a complete framework to verify complex distributed systems, from the formalisation of specifications to runtime execution. The proposed framework aims at covering several holes in the verification process of systems executing in an edge-to-cloud computing environment. Furthermore, I show how to repurpose the verification effort to perform design exploration with the goal of approximating the SUV without negatively affecting its functional correctness. The rest of this thesis is organised as follows. In chapter 2, I report useful background information. In chapter 3, I show an overview of the proposed framework and the objective of this thesis. In chapters 4, 5, 6, 7 and 8, I describe in detail the methodologies and tools implemented to achieve the objectives; furthermore, to improve readability, each of these chapters will contain an introduction, related work, methodology and result section related only to the contributions described in the chapter. Finally, in chapter 9, I draw my conclusions.

Chapter 2

Background

In this chapter, I provide the required background information to understand this dissertation. Furthermore, I report useful definitions used in the rest of the paper. Each chapter may add further definitions (in the “preliminaries" section) relevant only to the specific subtopic.

2.1 Introduction to assertion-based verification

Definition 1. *An **assertion** is a logic property that must hold (safety property) or must become true (liveliness property) during the execution or simulation of the design.*

ABV is a methodology that utilizes assertions as a central target for a variety of verification techniques; in particular, it is used for the efficient verification of a collection of specifications by the synergistic application of simulation, formal verification, and semi-formal verification. ABV helps overcome three major challenges of the verification process: bug detection, observability and controllability.

1. Bug detection: when an assertion fails, it will provide enough information to the designer to start fixing the problem. In contrast, in the absence of assertions, it may take hours or days to even find out the reason for a failure. As a result, effective use of assertions can drastically reduce the verification and debug time.
2. Observability: when the DUV is available, assertions may have access to the internal states of the design, allowing the verification engineer to immediately observe bugs without needing to propagate them to the outputs.
3. Controllability: low controllability issues occur when the number of required test benches becomes prohibitive. In this context, formal techniques may employ assertions to mathematically prove the correctness of the design without stimulating it with all possible inputs.

Another benefit of ABV is reusability, as the same set of assertions may be used at different abstraction levels of the designing process and with different verification techniques. Furthermore, assertions are a useful tool to formalise the initial requirements for early identification of inconsistency or incompleteness of the design’s specifications. Finally, assertions can be used for documentation purposes.

2.2 Assertion-based verification techniques

Assertions can be used to effectively verify specifications using simulation, formal/semi-formal verification as well as hybrid methods. Verification techniques present different characteristics and challenges depending on whether they are applied offline

(before deployment) or online, during the execution of the system. The two most popular families of offline verification approaches are dynamic verification and formal verification.

- **Dynamic verification** is applied to both HW and SW designs. In Dynamic verification, the design is simulated using a limited set of test patterns; in this context, assertions are checked dynamically according to the evolution of the design during simulation. Dynamic approaches are scalable for large designs; however, they do not provide any mathematical guarantees about the correctness of the design.
- **Formal verification** is the process of mathematically checking that the behaviour of a system (described using a formal model) satisfies a given property (also described using a formal model). Formal verification is widely used to prove the functional correctness of HW designs. There are a wide variety of formal verification methods, including theorem proving, model checking, satisfiability solving, and equivalence checking. Model checking is most suitable in the context of assertion-based verification since assertions can be viewed as properties.

Even though offline verification approaches can provide high confidence in the correctness of a design, they can not guarantee the absence of unexpected behaviours in most large industrial designs. This becomes a major pitfall in safety-critical contexts where a failure can harm people or things. To overcome this limitation, research pushed toward a new kind of technique where verification is applied "online" during the execution of the system. This family of approaches is known in the literature as runtime verification (RV).

Runtime verification is based on extracting information from a running system and using it to detect (and possibly react to) observed behaviours satisfying or violating certain assertions. The typical aspects of an RV application are the generation of a checker from a specification and then the use of the checker to analyze the dynamics of the SUV. If a failure is detected, the system can either halt to avoid damage or initiate a recovery procedure.

2.3 Specification languages

Several languages have been proposed to formalise specifications for different application domains. Assertions can be usually classified into two different categories.

- Non-temporal assertion: usually implemented as an $assert(p)$ function defined inside the source code of the design; it checks if the propositional formula p is satisfied when *assert* is called during execution.

  Definition 2. *A **proposition** is a Boolean expression that can be constructed by using Boolean operators (&&, ||, !) between Boolean expressions, or relational operators ($<, >, >=, <=, ==, !=$) between numeric expressions. Numeric expressions are constructed by using arithmetic operators ($+, -, *, /$) or bitwise operators (&, |, $\sim$, $>>$, $<<$). Boolean constants and SUV variables are propositions. Numeric constants and SUV variables are numeric expressions.*

- Temporal assertions: the formula is formalised using temporal logic. The truth value of the formula is usually checked independently from the execution of the design.

Temporal logic is expressive enough to represent most properties that a finite-state system needs to satisfy. Therefore, it is usually the first choice for defining assertions. The most popular languages to formalise temporal assertions are linear temporal logic (LTL) and computation tree logic (CTL). LTL can describe properties of individual executions starting from a set of initial states, and the semantics is defined as a set of paths; on the other hand, CTL describes properties of a computation tree: formulas can reason about many executions at once, and the semantics is defined in terms of states and paths. CTL logic cannot be used to check assertions with semi-formal techniques involving simulation, instead, this logic is mostly used with formal techniques such as model checking. In this thesis, I will mainly use LTL and some of its extensions.

Definition 3. ***Linear temporal logic*** *is a modal temporal logic used to formalise behaviours spanning multiple instants of time. In LTL, one can encode formulae about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, and so on. Given a finite set of propositions P, the set of LTL formulas over P can be defined, in negation normal form, as follows:*

- *$a \in P$ and $\neg a$ are LTL formulas;*
- *if ϕ_1 and ϕ_2 are LTL formulas then $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $X\ \phi_1$, $\phi_1 U \phi_2$, $\phi_1 R \phi_2$, $G\ \phi_1$ and $F\ \phi_1$ are LTL formulas.*

Intuitively, the semantics of temporal operators X (next), U (until), R (release), G (always) and F (eventually) is:

- *$X\ \phi_1$ holds at time t if ϕ_1 holds at time $t+1$;*
- *$\phi_1 U \phi_2$ holds at time t if ϕ_1 holds for all instants $t' \geq t$ until ϕ_2 holds;*
- *$\phi_1 R \phi_2$ holds at time t if ϕ_2 holds for all instants $t' \geq t$ until and including the instant where ϕ_1 first becomes true; if ϕ_1 never becomes true, ϕ_2 holds forever.*
- *$G\ \phi_1$ holds at time t if ϕ_1 holds at all instants $t' \geq t$. In other words, ϕ_1 is true globally in the future.*
- *$F\ \phi_1$ holds at time t if ϕ_1 holds at some instant $t' \geq t$. In other words, ϕ_1 eventually becomes true in the future.*

Besides the classical operators, LTL usually allows the use of more advanced SERE (Sequential Extended Regular Expressions) operators. I recommend [1] for a full reference of the semantics of LTL+SERE.

2.4 Distributed systems and the Edge-to-Cloud computing paradigm

Distributed systems are generally defined as computational artefacts or components that run into execution units placed at different physical locations, and that exchange information to achieve a common goal. A localized unit of computation in such a setup is generally assigned its own process of control (possibly composed of multiple threads) but does not execute in isolation. Instead, the process interacts and exchanges information with other such remote units using the communication infrastructure imposed by the distributed architecture, such as a computer network. Distributed systems are notoriously difficult to design and verify.

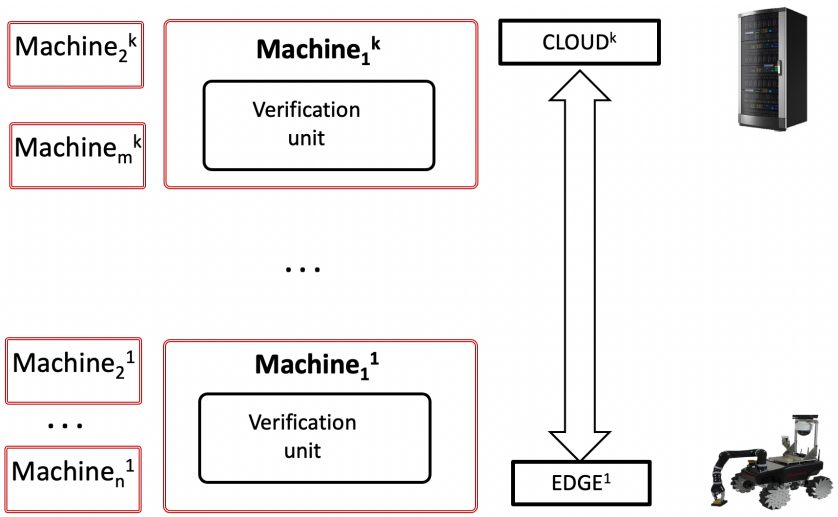

FIGURE 2.1: Example of edge-to-cloud paradigm

The edge-to-cloud computing paradigm is a way of classifying the nodes of a distributed system depending on how close each machine is to the source of the data, each node is logically placed at a certain computational layer. The bottom layer is called "the edge". Here we can find computing platforms such as microcontrollers and off-the-shelf devices. These devices guarantee low latency for verification because data is elaborated on the spot. However, they provide low computational resources. The upper layer is called "the Cloud". Here we have high-performance computing platforms, such as computer clusters and servers. The verification latency could become unpredictable and sometimes exceptionally high. This issue is due to the delay induced by moving data through the network from lower layers (where data is generated) to the Cloud (where verification occurs), and vice versa. Fig. 2.1 depicts an example of the edge-to-cloud computing paradigm, where machines are classified into either the edge or the cloud computing layer.

Chapter 3

Objectives

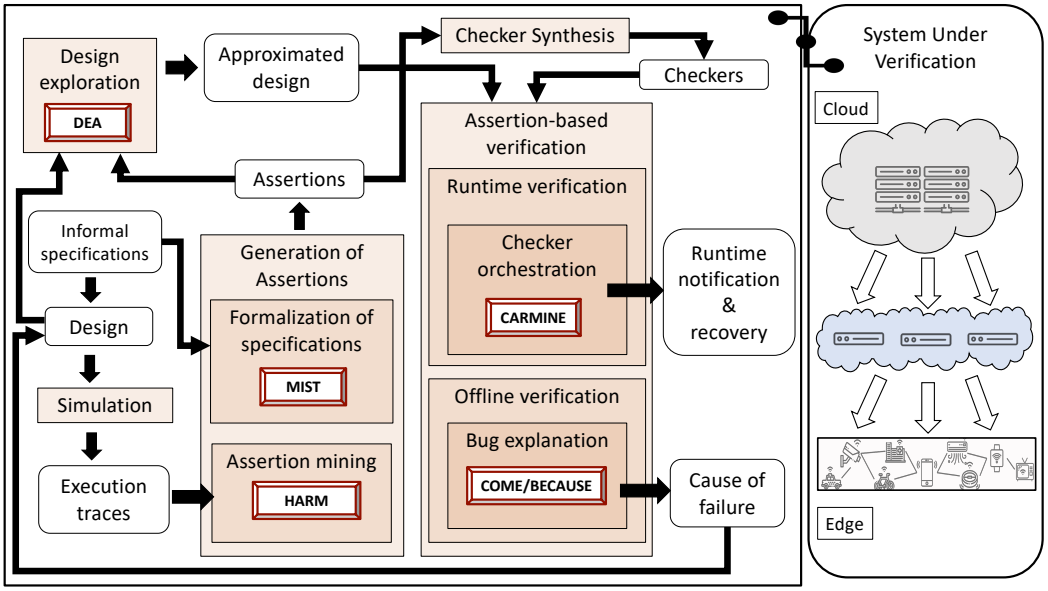

FIGURE 3.1: Overview of the verification flow

In fig.3.1, I report an overview of the proposed verification flow. The SUV architecture is organised following the edge-to-cloud paradigm.

The verification flow starts from the formalisation of specifications. In this process, the design intent is formalised into assertions. Assertions can be generated through manual effort or by using automatic tools called assertion miners. After that, the generated assertions are used to verify the SUV. This process is carried out by using semi-formal approaches involving assertions. Assertions are usually synthesised into checkers capable of verifying if the corresponding assertions hold during execution or simulation. Finally, the synthesised checkers can be either used following an offline approach to identify and correct bugs before deployment or to perform runtime verification after deployment.

Once the verification process is completed, the generated assertions can be used to guide the exploration of the SUV. In particular, I show a meaningful use case where assertions are employed to guide the approximation process by exhibiting the portion of the design that can be approximated without introducing major functional bugs or system vulnerabilities.

3.1 Verification tools

In the aforementioned verification flow, I propose several novel methodologies implemented into five tools called MIST [2], HARM [3], [4], CARMINE [5], COME [6], BECAUSE [7] and DEA [8].

In the context of the formalisation of specifications, I propose two tools, MIST and HARM. MIST is 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 (who 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 checkers and testbenches to verify the correctness of the firmware implementation automatically. MIST is described in chapter 4.

HARM is a tool to generate LTL assertions starting from a set of user-defined hints and the simulation traces of the SUV. The tool is agnostic with respect to the design from which the trace was generated, thus the SUV source code is not necessary. The user-defined hints involve LTL templates, propositions and ranking metrics that are exploited by the assertion miner to reduce the search space and improve the quality of the generated assertions. This way, the tool supports the work of the verification engineer by including his/her insights in the process of automatically generating assertions. HARM is described in chapter 5.

The assertions generated with HARM are used in chapter 8 in the context of approximate computing. That chapter proposes an automatic methodology and a corresponding tool called DEA to guide the approximate-computing design exploration at the RT (register transfer) level.

In the context of bug explanation, I propose COME and BECAUSE. Given a set of execution traces corresponding to sequences of instructions highlighting unexpected behaviours (where an assertion failed), COME and BECAUSE aim at reducing the traces, keeping only the relevant instructions to understand and correct the bugs.

BECAUSE works by combining dynamic program slicing with a clustering procedure. First, program slicing is applied to remove instructions not belonging to the cone of influence of the unexpected behaviour. Then, clusters of instructions based on “store operations" of the SUV are created to guide the heuristic in removing further irrelevant instructions.

COME exploits symbolic simulation to generate a set of execution traces. The tool employs alignment algorithms to compare the generated traces to extract the essential instructions actually characterising the unexpected behaviours. COME and BECAUSE are described in chapter 6.

Finally, in the context of runtime verification, I propose CARMINE, a tool to automatically generate a runtime verification environment (RVE) for distributed systems. The RVE is automatically synthesised starting from a set of LTL assertions. The RVE is capable of automatically balancing the verification load by orchestrating the checkers on the different machines of the distributed system, preventing the RVE from saturating the computational resources of the SUV. CARMINE is described in chapter 7.

Chapter 4

Formalisation of specifications and offline verification: MIST

4.1 Introduction

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[9], 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 [10].

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 specifications, which are 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 specifications and the final implementation [11]. 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 set of specifications, it becomes impractical to define any formal or semi-formal verification strategy. Generally, those strategies require describing the expected behaviours in terms of assertions unambiguously. In the case of semi-formal approaches, the verification engineer has to define a set of testbenches to stimulate the DUV. 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 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 [12]. 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 chapter is organized as follows. Section 4.2 summarizes the state of the art. Section 4.3 overviews the methodology. Sections 4.4, 4.5, 4.6 and 4.7 explain in detail the methodology implemented in MIST. Section 4.8 reports the experimental results.

4.2 Related work

The formalisation of specifications is the process of translating the 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.[13] introduce an automated collaborative requirements engineering tool, called TestMEReq, to promote effective communication and collaboration between client stakeholders and 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 [14] 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. [15] propose an approach to verify firmware security properties using symbolic execution. The paper introduces a property specification language for information flow properties, which intuitively captures the requirements of confidentiality and integrity.

In [16], Buzhinsky presents a survey of the most popular existing approaches for formalising discrete-time temporal behaviours.

All the above works either use a standardised (such as property specification language PSL [17], (SystemVerilog Assertion SVA [18]) or a domain-specific formalisation language relying on temporal logic formalisms such as LTL and CTL.

Once the informal specifications are thoroughly translated into logic formulas, automatic verification can be applied to the target design.

To address the scalability problem, simulation-based approaches have been introduced to perform ABV. These 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.[19] 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 [20], 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.[21] present KLEE, a symbolic simulation tool capable of automatically generating tests that achieve high coverage for C/C++ programs.

In [22], the authors introduce a purely SAT-based semi-formal approach for generating multiple heterogeneous testcases 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. Virtual platforms combine high-speed processor simulators and high-level functional models of the hardware building blocks, to provide an abstract and executable representation of the hardware to software developers and to system architects"[23]. 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.

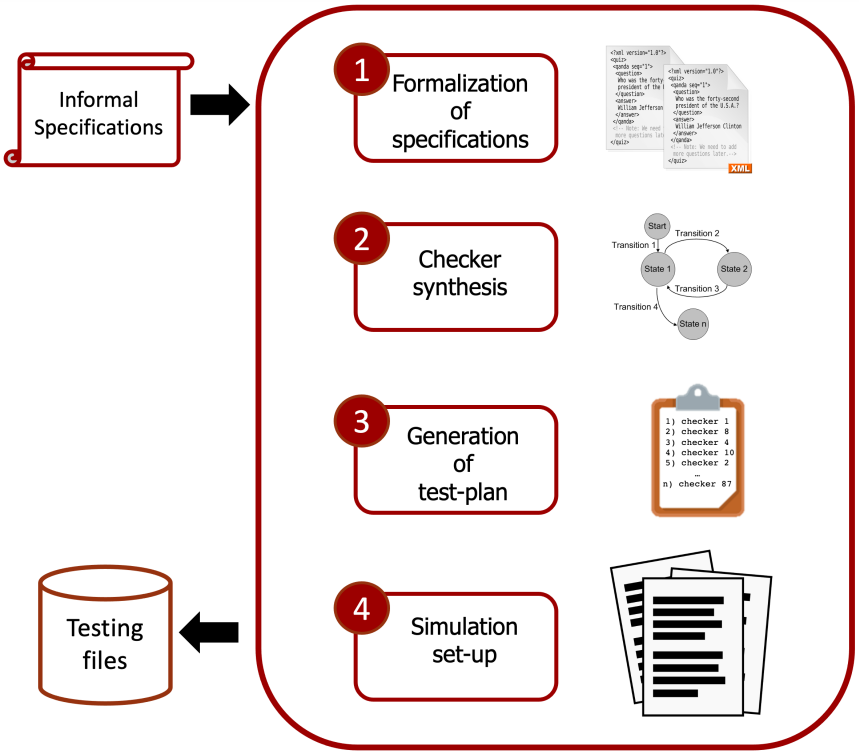

FIGURE 4.1: Execution flow of MIST.

4.3 Methodology

As shown in Fig. 4.1, the proposed methodology is composed of four main steps executed sequentially. The input of MIST is a set of temporal behaviours. These behaviours are generated in the first step of the methodology, starting from a set of 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.

(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, promoting 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.

4.4 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.

4.4.1 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 the comfort mode and sends an acknowledgement 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>
  The firmware is in standard mode, boiler temperature is
   equal to 18
  </precondition>
  <postcondition>
  The 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 the comfort mode and sends an
   acknowledgement as output
  </consequent>
</assertion>
```

LISTING 4.1: 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.

4.4.2 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 shown 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 the form 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 the 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 [17]. 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 4.2: 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}.

4.4.3 Type system

In addition to the features described above, the propositions used in each fragment completely support 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 a 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 4.3 shows a possible declaration for the variables used in listing 4.2.

```
<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 4.3: 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.

4.4.4 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 an 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>
```

$[var_1, var_2, \ldots, var_n]$ 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 shown below.

$$
\begin{array}{l}
tv\_tb_i \ := \\
(var_1\_val_1,\ var_2\_val_1,\ \ldots,\ var_n\_val_1,\ duration_1), \\
(var_1\_val_2,\ var_2\_val_2,\ \ldots,\ var_n\_val_2,\ duration_2), \\
\qquad\qquad\qquad \ldots \\
(var_1\_val_k,\ var_2\_val_k,\ \ldots,\ var_n\_val_k,\ duration_n)
\end{array}
$$

Each tuple $(var_1\_val_j,\ var_2\_val_j,\ \ldots,\ var_n\_val_j,\ duration_j)$ identifies a piece of test-vector where the variables $var_1, var_2, \ldots, var_n$ are forced with the values $var_1\_val_j,\ var_2\_val_j,\ \ldots,\ var_n\_val_j$ 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 4.2, 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 4.2.

```
<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 4.4: 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 one 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 4.2. Both testbenches are injected from time t_0 to time t_{3199}.

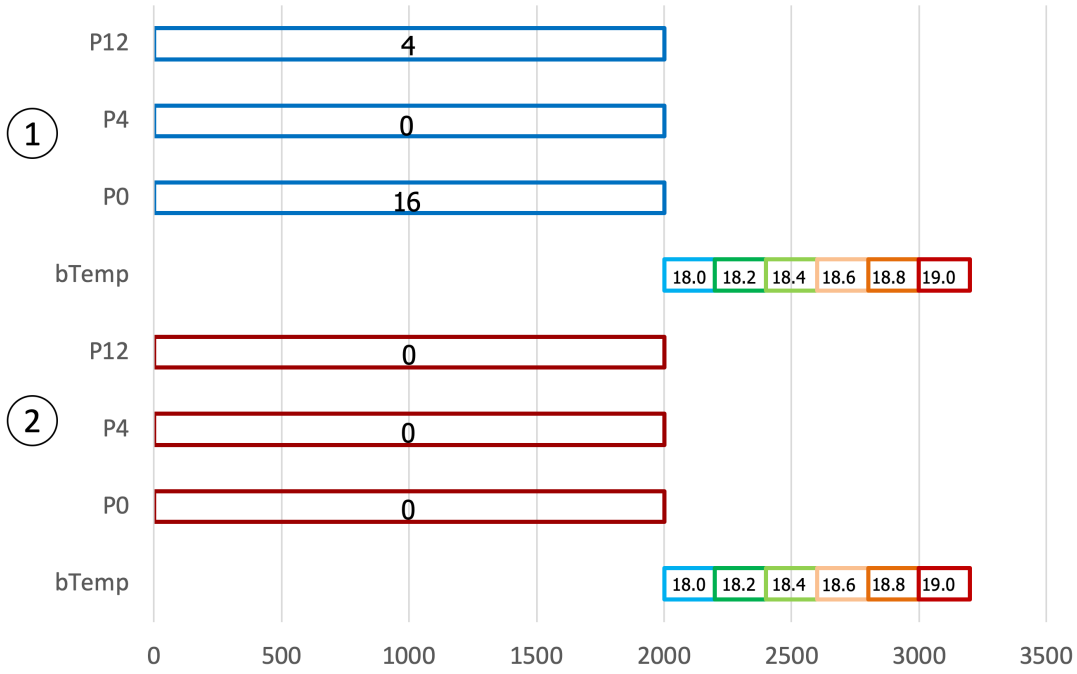

FIGURE 4.2: Testbenches timeline

4.5 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 assertion to a PSL formula. Secondly, each PSL formula 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

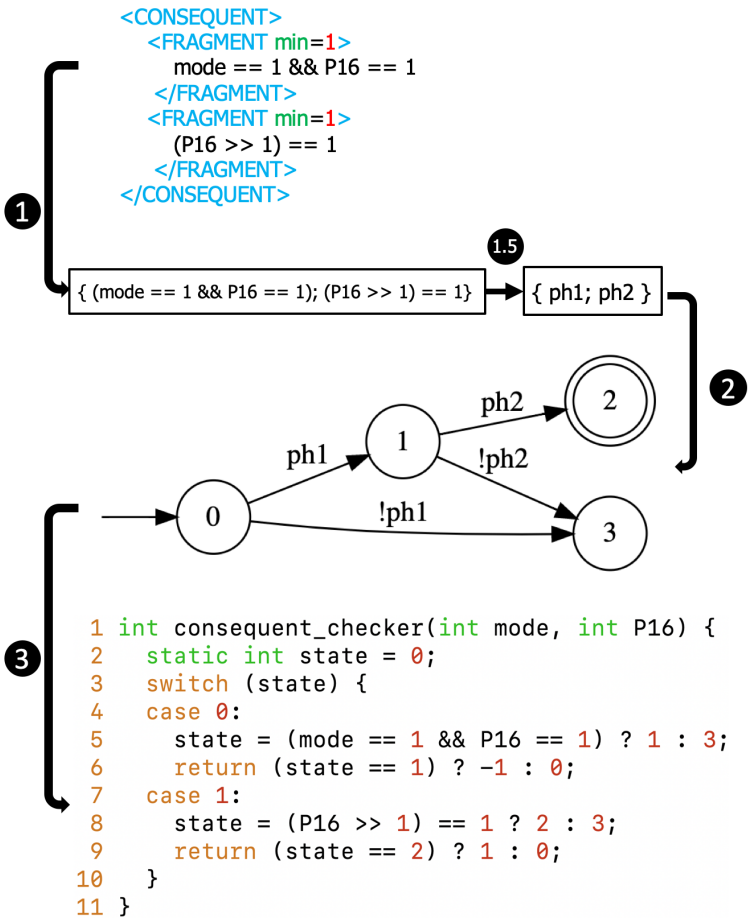

FIGURE 4.3: Example of checker synthesis.

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 4.4 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[24], 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. 4.3 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. 4.3, 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).

4.6 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 postconditions 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.

4.6.1 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 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_j^i 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.

4.6.2 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 $variable_1 == constant_1$ & $variable_2 == constant_2$ & ... & $variable_n == constant_n$ that represents a concrete memory configuration. To simplify

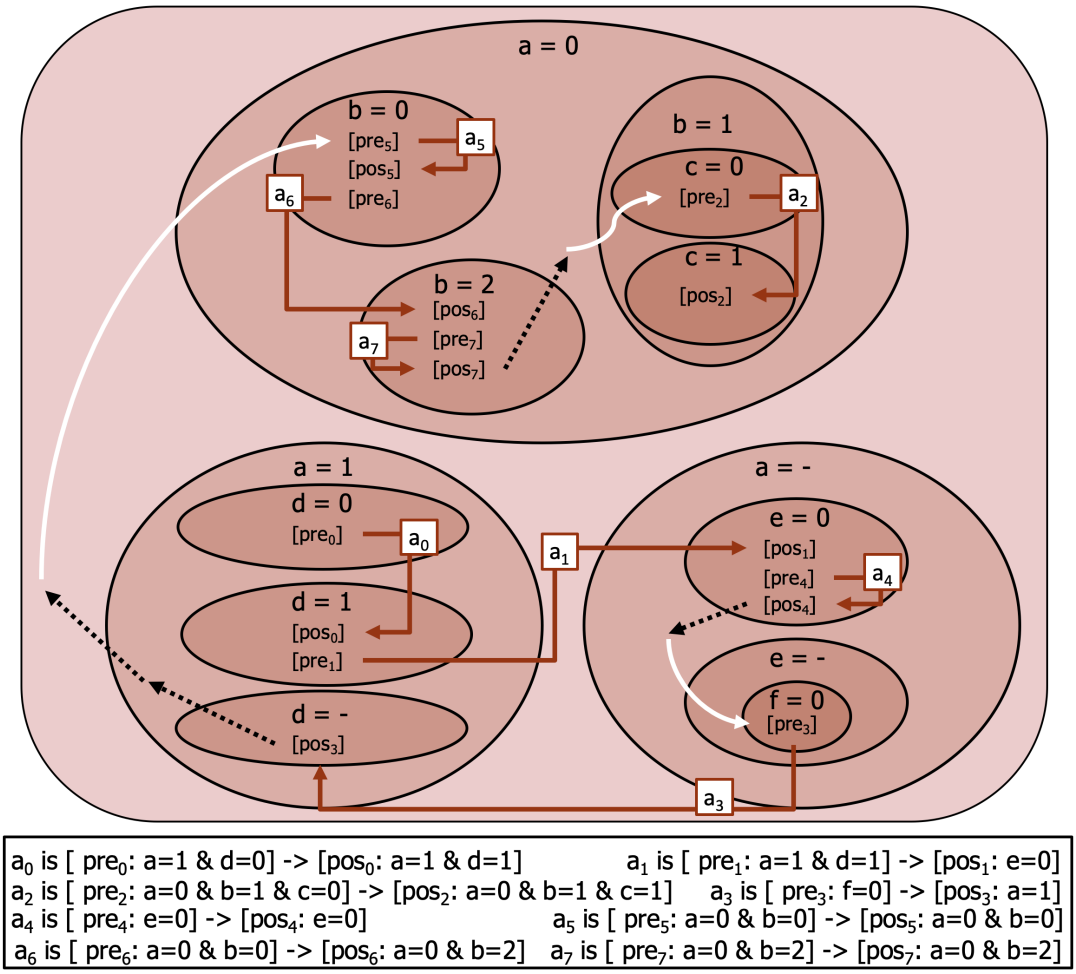

FIGURE 4.4: Example of test plan generation

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. 4.4 to clarify the procedure. At the bottom of Fig. 4.4 we report the list of assertions used in the example. For instance, the assertion described in Section 4.4 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. 4.4. 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 warns the verification engineer. To be clear, in 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 level of *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 must violate the maximum number of consecutive applications of rule R2 tolerated

TABLE 4.1: Completeness analysis for example in Fig. 4.4

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

by the user. Intuitively, the resulting completeness is an index describing how likely it is for the set of behaviours to allow the generation of a sequential test capable of verifying all the specified behaviours without holes; a “hole" is a system state $s_{\perp}$ reached after testing a behaviour such that no other behaviour can be tested starting from $s_{\perp}$, hence, requiring the user to specify how to reach (force) the initial state of the following testable behaviour in the test suit. Each time a missing link is found, the completeness is reduced.

Table 4.1 shows the completeness of 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.

4.7 Simulation setup

4.7.1 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 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.

4.7.2 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%]


```

FIGURE 4.5: 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 testbenches. 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. 4.5.

In this example, we show the result of two possible failures for the running examples depicted in listing 4.2 (66_a) and 4.4 (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.

4.8 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.

4.8.1 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 an LCD display, 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 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. 4.6.

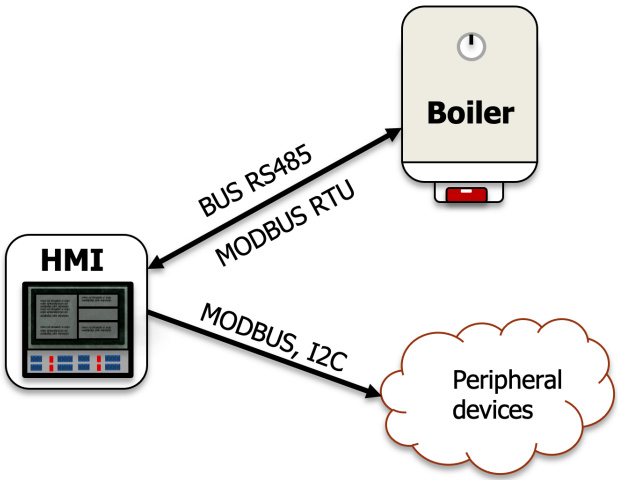

FIGURE 4.6: Case study

4.8.2 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

TABLE 4.2: 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 4.3: 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 %

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 Workbench). 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 activate a functionality. However, during the simulation, the correct behaviour did not occur even when providing the correct stimuli. Using MIST for the verification of such firmware was considerably helpful in identifying a temporal inconsistency of Modbus and I2C protocols that caused a delay in its execution.

Furthermore, the generation of the test plan for 100 behaviours suggested a remarkable incompleteness in the firmware specifications. In table 4.2 we can observe the completeness estimations produced for the case study by considering the approach proposed in Section 4.6. 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 4.3. 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.

Chapter 5

Assertion mining: HARM

5.1 Introduction

Unfortunately, assertion definition is a time-consuming and error-prone task, which requires high expertise to reason in terms of logic formulas [25].

To overcome the severe limitations of manually defining assertions, starting with the pioneering work on specification mining proposed in [26], verification engineers have developed several assertion mining approaches to automatically extract assertions from the actual implementation of the DUV (see Section 5.2) Then, the mined assertions can be compared against the initial specifications to verify if all expected behaviours have been implemented in the DUV. Furthermore, by analysing the mined assertions, the verification engineer can discover the presence of unexpected behaviours caused by design errors or malicious code deliberately inserted in the DUV [27]. Finally, mined assertions can be used also for documentation purposes.

There exist mainly two ways for automatically extracting assertions from the DUV implementation. A first possibility is to statically analyse the DUV source code searching for (and exploiting) cones of influence and control-flow graphs among variables. The alternative method consists of dynamically applying data mining techniques searching for association rules on a set of traces obtained by simulating the DUV. Static and dynamic analyses present complementary advantages and disadvantages in terms of accuracy and scalability [28], [29]. While static mining pursues generalisations and abstractions, dynamic approaches can only extract likely assertions, i.e. formulas that represent solely the behaviours that are exposed in the considered traces. On the other hand, static methods have scalability issues while dynamic assertion mining is much more computationally efficient.

Even if the quality of likely assertions extracted by dynamic approaches strictly depends on the level of exhaustiveness exhibited by the considered traces (indeed, a common characteristic of any simulation-based method), dynamic mining has gained more and more consensus in the last decade, thanks to its higher scalability. In addition, it has been shown that dynamic approaches can be applied also when the source code of the DUV is not available and only a black-box implementation is available (e.g., for checking the presence of malicious code on a third-party IP).

A popular way of extracting assertions from traces consists in using greedy-based heuristics with decision trees and association rules to generate invariants that follow the template $always(antecedent \rightarrow consequent)$, where *antecedent* and *consequent* represent arbitrarily temporal behaviours. The heuristic way is fundamental to guarantee the scalability of the approach as checking long traces for every possible temporal behaviour is not feasible for large designs. However, heuristic approaches can be unable of mining a satisfying set of assertions, as some interesting behaviours

are not generated, while irrelevant assertions are extracted. As a result, the set of mined assertions can superficially reflect the real behaviours exhibited by the simulation traces; therefore, the verification engineer needs to manually write further assertions anyway. Thus, the majority of existing dynamic assertion miners exploit heuristic approaches to guarantee scalability at the cost of a poor set of generated assertions, while other tools sacrifice scalability to provide a more complete set of assertions.

An additional drawback of existing approaches, both static and dynamic, is related to the fact that they blindly extract hundreds of assertions without considering the designer's intent and the application domain. This negatively affects the post-mining analysis from the verification engineers, which lacks an automatic way of ranking the mined assertions in terms of interestingness. Consequently, they need to manually analyse the mined assertions searching for those that better meet their verification purposes. Indeed, the interestingness of an assertion is an abstract concept strictly correlated with the user's perspectives and necessities. Overall, existing miners miss a method that allows the user to decide the desired trade-off between scalability, completeness and manual effort to be applied for each context of interest.

To handle the aforementioned trade-off, in this chapter, we propose a new assertion miner, called HARM (Hint-Based AsseRtion Miner). Given a set of traces and a set of *hints*, HARM generates LTL assertions in the form $always\ (antecedent \rightarrow consequent)$. The hints are represented by (i) *template* formulas characterising temporal behaviours of interest for the verification engineer, (ii) *propositions* defining relations between variables that he/she wants to investigate, and (iii) *metrics* to assess the interestingness of the generated assertions. In particular, templates are employed to model the temporal relations between the operands involved in the implication inside the "always" statement. They can be defined by using all LTL and SERE operators. A proposition can be any kind of Boolean expression that can be constructed in C/C++ by connecting variables through Boolean, relational and arithmetic operators. Metrics are indexes used to measure the quality of an assertion. The tool allows the definition of *contexts* to cluster and rank the generated assertions according to the given hints. In general, the quality and the completeness of the mined assertions, with respect to the considered traces, vary with the number of hints and computational resources provided by the user. If the user is unsatisfied with the mined assertions, he/she can provide more resources and/or change the hints to improve their completeness and quality.

With respect to the existing approaches, the main contributions of this work are listed below:

- A very fast miner engine (with respect to the state of the art). HARM features a parallelised linear-time algorithm to evaluate an assertion and generate its contingency table, efficiently managing input traces that are millions of time units long.

- An agnostic and general-purpose miner. HARM does not require the sources of the DUV as input. As a matter of fact, the tool does not even require the existence of the design implementation, but only its traces.

- A customizable template-based miner. Most tools provide only one mining template, and a few others allow the user to choose between a limited subset of templates. HARM provides $always(antecedent \rightarrow consequent)$ as a base template, the user is then allowed to customise the antecedent and consequent by using the full set of LTL temporal operators included in the PSL. While

writing the right set of templates to mine high-quality assertions is not a trivial task, the users can start from a set of well-known generic templates, like those proposed in [30], and gradually refine their hints according to the effectiveness of mined assertions.

- Support to non-Boolean types. HARM allows the mining of assertions predicating on both Boolean and non-Boolean variables, it supports all Verilog/System Verilog and C/C++ primitive data types.
- A generalised procedure to generate assertions through a template-based and entropy-based decision tree algorithm.
- A context-based approach to single out interesting assertions according to the user requirements.

HARM is an open-source tool freely available at [31].

The rest of this chapter is organised as follows. Section 5.2 reports the related work; section 5.3 provides a set of preliminary definitions useful to understand the technical parts of the chapter; section 5.4 describes the architecture of HARM; sections 5.5, 5.6, 5.7 and 5.8 report a detailed description of each step of the methodology implemented in the tool; section 5.10 reports the experimental results.

5.2 Related work

Static and dynamic approaches have been proposed for assertion mining. The first focus on analysing the source code of the DUV, while the second considers only the traces obtained by simulating the DUV by means of input stimuli. As static and dynamic analyses present complementary advantages and disadvantages concerning accuracy and scalability, some works employ mixed static and dynamic techniques.

Among the first works in the software domain, [32], [33] propose scenario-based specification mining approaches where the source code is instrumented to mine linear sequence charts. However, these approaches are not aimed at discovering the complete behaviour of the DUV, but only the collaboration among its components. Other works mine the specifications of the DUV in the form of algebraic equation [34] or Hoare-style equations of pre and post-conditions [35], [36], but the temporal behaviours are not considered. The work in [37] is one of the first attempts at statically generating assertions by using a template-based technique. The employment of user-defined templates greatly increases the applicability of the approach as templates can be chosen according to the design's characteristics.

In the hardware domain, the authors of [38] describe IODINE, a tool to automatically extract likely design properties of hardware descriptions. It generates invariants by making hypotheses on one or more variables in the design and by analysing its behaviour over a set of inputs. In [39], the authors developed a new procedure called "Semantic Inference" with the specific goal of automatically translating the behaviour of a cyber-physical system into a formal specification. Seshia et al. proposed a gate-level approach that extracts assertions compliant with a predefined set of temporal templates [40]. In [29], [41]–[43], Vasudevan et al. proposed Goldmine, a tool for extracting LTL assertions following the template $G\ (boolVar_1\ \&\ next[1](boolVar_2)\ \&...\&\ next[N](boolVar_m) \rightarrow boolVar_k)$. It generates assertions by using static analysis and data mining. In particular, GoldMine is composed of five main parts: first, a data generator stimulates the DUV with random input patterns to generate behavioural traces; second, a static analyser extracts design constraints like

cones of influence; then an assertion miner engine generates the assertions through a decision tree-based supervised learning procedure; then an assertion qualifier evaluates and ranks the assertions by using support and confidence metrics; and finally, the mined assertions are formally verified by using SMV [44] to check if they actually hold in the DUV. The authors have recently improved their ranking method by introducing complexity and importance metrics in [45]. In [46], Ghasempouri et al. proposed a paper on the same topic where the interestingness of assertions is determined by using data mining metrics.

More recently, system-level approaches that work also on non-Boolean data types have been presented by Danese et al. in [47], [48]. In [47], a time-window-based approach, focused only on DUV control signals, is proposed to extract assertions related to the I/O protocol. A more generic tool is instead described in [48], but mined assertions are restricted to a subset of pre-defined temporal patterns. The only two approaches that generate temporal assertions considering arithmetic/logic expressions among the variables of the DUV are ODEN [49] and the work described in [50]. In [51] a tool called A-TEAM is introduced for template-based assertion mining. The tool allows the definition of templates by using propositional logic in addition to the *next*, *until* and *release* LTL operators, in order to generate temporal assertions. A-TEAM employs the Apriori algorithm [52] to extract high-frequency atomic propositions. Then, the propositions are used to instantiate the templates through a set of justification rules. Finally, the generated assertions are qualified in terms of fault coverage.

Commercial tools are also available for automatic assertion generation at RTL, e.g., Atrenta BugScope [53] and Jasper ActiveProp [54]. The first generates SVA or PSL assertions where only the *next* temporal operator is considered. The second generates both structural and behavioural SVA *next*-based assertions.

The above tools are effective in automatically generating LTL assertions; however, they present several limitations which are solved by HARM:

- All approaches employing static analysis techniques require the source code of the DUV, making them unsuitable in all verification flows in which the DUV is not available. Furthermore, they are dependent from the implementation language of the DUV, greatly reducing their applicability.

- The basic template for HARM is $G(antecedent \rightarrow consequent)$, however, the antecedent and the consequent can be customised by the user through the grammar shown in Fig. 5.1, thus allowing the specification of a wide set of different LTL formulas. On the contrary, existing tools work on a narrower set of predefined templates. For example, IODINE [38] and DAIKON [36] can extract only invariant formulas, then no temporal operators are allowed. Goldmine [29], [41]–[43] extracts assertions of the form $G(antecedent \rightarrow consequent$ but it supports only the use of the temporal operator next (i.e., X). ODEN [49] implements only $G(a \rightarrow next(b))$, $G(a\ alternating\ b)$ and $G(a \rightarrow a\ U\ b)$, where a and b are simple propositions. A-TEAM [51] is the only other tool supporting a set of templates similar to HARM, but it only partially supports the decision tree operator defined in Def. 6 of Section 5.3, as it does not allow conjunction of *next* operators with holes in the temporal extension.

- Goldmine performs assertion ranking by analysing the source code of the DUV; [46] proposes ranking metrics employing the contingency table of the miner; A-TEAM proposes a minimal fault-coverage ranking technique. Instead, HARM is more general and does not require the source code of the DUV. Furthermore,

these tools do not provide a completely configurable context-based approach to allow the generation of an interesting set of assertions for every domain of application.

- None of the above tools provides a configurable and generalised entropy-based decision tree procedure allowing the use of a variety of decision tree operators (def. 6) that can be instantiated in a templated formula according to the user's requirements.

5.3 Preliminaries

In this chapter, we make use of well-known LTL operators, such as *Next (X)*, *Until (U)*, *Release (R)* and *Always (G)*. We kindly refer the reader to [1] for a complete description of the corresponding semantics.

Definition 4. *Given a finite sequence of time units* $\langle t_1, ..., t_n \rangle$ *and a set of variables* $\{v^1, ..., v^m\}$, *a* ***data trace*** *is a sequence of tuples* $(t_i, v_i^1, ..., v_i^m)$ *such that* v_i^j *is the value assumed by variable* v^j *at time* t_i. *We consider time as a discrete sequence of samples* $t_0, t_1, ...t_N$ *where* t_{i+1} *occurs after* t_i.

For the sake of compactness, the term *trace* will be used instead of "data trace" in the rest of the chapter.

Furthermore, in this chapter, we will make extensive use of propositions (def 2). Here a proposition is labelled with one of the following tokens "a", "c", "ac" to specify that it appears in the assertion, respectively, into the antecedent, the consequent or both. For example, in Fig. 5.3, P contains four "a" propositions ($v1, v2, v3, v4$) and two "c" propositions ($v5, v6$).

Definition 5. *A* ***placeholder*** *is a Boolean variable that can be substituted by a proposition.*

Similarly to propositions, we will refer to three kinds of placeholders: those who appear only in the antecedent (aP), only in the consequent (cP), or in both the antecedent and consequent (acP). Placeholders of kind aP, cP and acP can only be substituted by propositions labelled with "a", "c", and "ac", respectively. In the rest of the paper, placeholders are always indicated in the form PN, where N is a positive integer number. For example, template t_1 of Fig. 5.3 contains two placeholder of kind cP (P1, P2) and one of kind aP (P0).

Definition 6. *A* ***decision tree (DT) operator*** *is a special type of temporal (or propositional) operator that can be instantiated by using a decision tree algorithm.*

Currently, HARM implements three DT operators as shown in Fig. 5.1:

- ..&&.. to generate an "and expression", e.g. $v1$ && $v2$ && ...;
- ..##N.. to generate a "chain of nexts", e.g. $v1$ ##1 $v2$ ##1 ...;
- ..#N&.. to generate a "chain of nexts of and expressions", e.g. $v1$ && $v2$ ##1 $v3$ && ...##1 ....

A DT operator can only appear once in the antecedent of each template. This is necessary to preserve the scalability of the approach. An uninstantiated DT operator is equal to the *true* Boolean constant.

```
template : G(implication)

implication : tformula -> tformula
        | tformula => tformula
        | {sere} |-> tformula
        | {sere} |=> tformula

tformula: proposition | placeholder | ..&&..
    | (tformula) | !tformula | tformula && tformula
    | tformula || tformula | tformula xor tformula
    | tformula U tformula | tformula W tformula
    | tformula R tformula | tformula M tformula
    | X [N..(N)?] tformula | X tformula
    | F tformula | {sere}

sere : proposition | (!)? placeholder | ..&&..
     | ..##N.. | ..#N&..  | (sere) | {sere}
     | sere | sere | sere & sere | sere && sere
     | sere;sere | sere:sere | sere[*N(..N)?]
     | sere[*] | sere[+] | sere[=N(..N)?]
     | sere[->N(..N)?] | ##N sere | ##[N(..N)?] sere
     | sere ##N sere | sere ##[N(..N)?] sere

placeholder: 'P' N
N: numeric
```

FIGURE 5.1: Template grammar adopted in HARM.

Definition 7. *A **template** is a temporal expression constructed by connecting propositions, placeholders and DT operators through PSL temporal operators in the form G (antecedent $\rightarrow$ consequent). A template is **potentially instantiated** (PIT) if all placeholders are substituted with propositions or if it does not contain any placeholders. A PIT may contain uninstantiated DT operators. In this chapter, a PIT where all DT operators are instantiated with propositions (or where there is no DT operator) is considered an **assertion**.*

Templates define the initial temporal behaviour (the temporal behaviour can change while instantiating a DT operator) of the mined assertions. The grammar for the templates supported in HARM is reported in Fig.5.1. It is a subset of the grammar of the popular framework spotLTL [55]. The grammar also supports sequential extended regular expressions (SERE).

Definition 8. *Given a trace tr of length n and an assertion as, an **evaluation** of as with respect to tr is the sequence of evaluation units $\langle e_1, ..., e_n \rangle$, where the **evaluation unit** e_i is the truth value of the assertion at instant t_i.*

The concept of evaluation for propositions and potentially instantiated templates is defined similarly, by substituting "assertion" with either "proposition" or "potentially instantiated template" in Def. 8. The evaluation unit e_i may be *true*, *false* or *unknown*. It assumes the *unknown* value if it depends on at least one instant t_j where j is greater than the length of the trace.

Definition 9. *An assertion, a proposition or a potentially instantiated template **holds** in a trace if and only if its evaluation does not contain any evaluation unit whose value is false.*

Definition 10. *Given a trace tr, and an assertion as, a **contingency table** is a 3 × 3 matrix displaying the frequency distribution of true, false and unknown evaluations units of the antecedent with respect to the consequent of as in tr.*

For example, in Table 5.1, ATCT represents the number of time instants in a trace where both the antecedent and the consequent evaluate to *true*.

	Cons. true	*Cons. false*	*Cons. unknown*
Ant. true	**ATCT**	**ATCF**	**ATCU**
Ant. false	**AFCT**	**AFCF**	**AFCU**
Ant. unknown	**AUCT**	**AUCF**	**AUCU**

TABLE 5.1: Contingency table

Definition 11. *A **metric** is a numeric formula measuring the impact of an assertion's feature in the assertion ranking (i.e. the process of sorting the assertions in increasing order of interestingness).*

The more prominent the feature, the higher its impact on the final ranking of the assertion. The elements of the contingency table are examples of features of an assertion. For example, in Fig. 5.3, M contains three metrics. Metric m_2 (frequency) results in a value closer to 1 as the assertion gets a higher value of ATCT (as the feature gets more prominent).

Definition 12. *A **context** is a set of propositions, templates and metrics.*

The user can provide HARM with his/her own context to guide the mining.

5.4 HARM architecture

HARM takes as inputs a trace (def.4), obtained by concatenating the set of traces of the DUV[1], and a set of contexts $C = \{c_1, ..., c_n\}$ (def.12). Each context c_i is a tuple (P, T, M), where P is a set of propositions predicating over the variables belonging to the trace, T is a set of templates, and M is a set of metrics. The output of the tool is a set of ranked assertions according to M. The architecture of HARM is shown in Fig. 5.2 and it is composed of the following 3 main steps, which are executed sequentially:

1. **Instantiation of placeholders:** in the first step of the methodology the placeholders in the templates T are substituted by using propositions belonging to P to generate PITs (def. 7).

2. **Assertion mining:** in the second step, all PITs are used to generate assertions holding on the input trace. There are two scenarios. (1) If a PIT does not contain a DT operator (def. 6), then the tool directly generates an evaluation (def. 8) for the PIT, and if the PIT holds (def. 9) then it corresponds to a mined assertion. (2) If a PIT contains a DT operator, then the tool invokes an entropy-based DT algorithm. The algorithm generates an assertion for each instantiation of the DT operator that makes the PIT hold on the trace.

[1]The generation of the traces is outside the scope of the paper. They can be generated by means of a user-defined testbench or an automatic test pattern generator. Its quality definitely affects the quality of the mined assertions, as it happens in any other simulation-based verification approach. It is reasonable to assume that in a simulation-based verification flow, a high-quality test set is available at the time the assertion mining is executed.

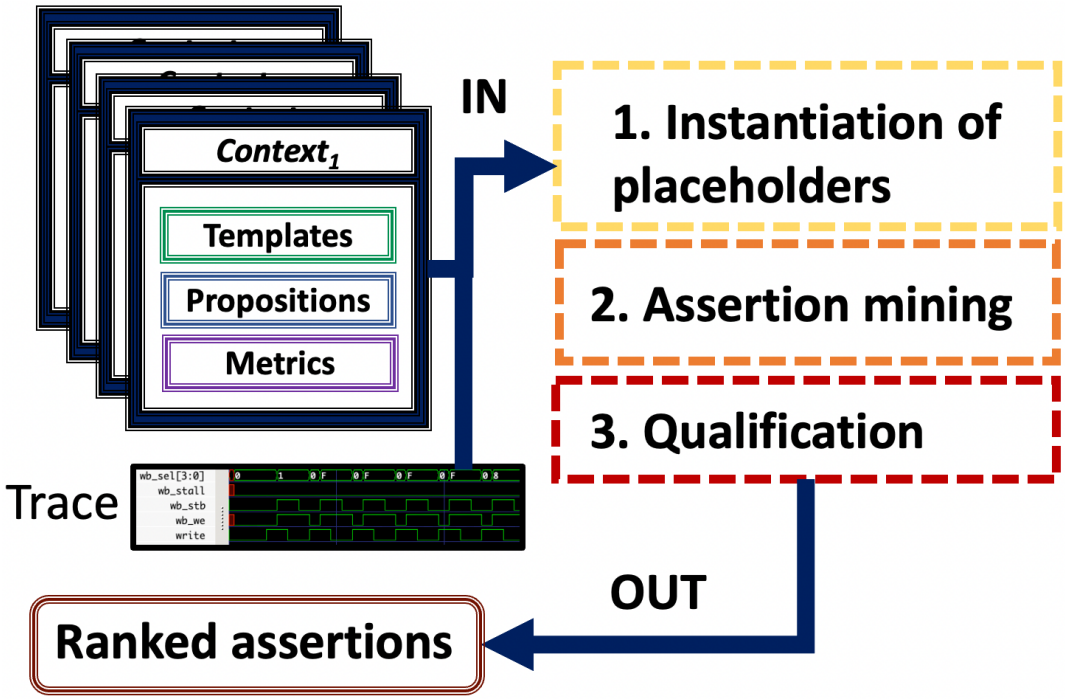

FIGURE 5.2: Methodology overview

3. **Qualification:** in the last step of the methodology, a context-based approach is applied to filter and rank the generated assertions according to their characteristics. This technique allows the user to single out the assertions that best fit all the features measured by the metrics of the considered context.

A detailed description of the methodology implemented in HARM is reported in the following sections. In addition, we give a practical demonstration of the various features of the tool by applying the methodology to a running example throughout the manuscript, whose inputs are shown in Fig. 5.3. In particular, the left side of Fig. 5.3 reports how the trace of the running example appears, while the right part describes the target context. According to Def. 4 the input trace is a sequence of values for the variables of the DUV at the varying of time. The time granularity depends on the DUV abstraction level. For example, at RTL the time granularity is based on the clock cycle, while at TLM it refers to the events associated with DUV transactions. Independently from this, the input trace can be informally seen as a matrix, as shown in Fig. 5.3, where each row corresponds to a different time instant, and each column refers to a single variable of the DUV. From the operational point of view, HARM accepts the standard .vcd and .csv file formats to read the input trace. In addition, please note that, for the sake of simplicity, without lacking generality, in the running example, we consider only the propositions $vi = true$ (with $i \in [1, 6]$), and to simplify the writing we will refer to $vi = true$ with vi. Finally, $v1$, $v2$, $v3$, and $v4$ will be considered antecedent propositions, while $v5$ and $v6$ consequent propositions).

5.5 Instantiation of placeholders

In the first step of the methodology, we generate a set of PITs by instantiating the templates in T with the propositions in P. For each template, HARM generates a set

Trace

t	v1	v2	v3	v4	v5	v6	v7
0	1	0	0	0	0	0	8
1	0	1	0	0	0	0	8
2	1	0	1	1	0	0	7
3	0	0	1	0	0	0	8
4	0	0	0	1	0	0	0
5	0	1	1	0	0	0	0
6	0	0	0	1	0	0	9
7	1	0	0	0	0	0	8
8	0	1	1	0	0	0	7
9	0	0	0	1	0	0	0
10	0	0	0	0	1	0	0
11	0	0	0	0	1	0	0
12	0	0	0	0	1	1	9
13	1	1	0	0	0	0	0
14	0	0	1	1	0	0	6
15	0	0	0	0	0	0	7

Context

$P = \{(v_{1-4}, a), (v_{5-6}, c)\}$

$T = \{ G(\{ ..\#1\&.. ; P0\} \mid\rightarrow X(P1\,R\,P2)),\ t_1 \;\; G((P0\ \&\&\ P1)\ ||\ (P2\ \&\&\ P3) \rightarrow X(v7 > 5))\ t_2 \}$

$M = \left\{ \left(causality, f, 1 - \frac{aftc}{traceLength}, 0.45\right), m_1 \;\; (frequency, s, \frac{atct}{traceLength}), m_2 \;\; \left(reps, s, \frac{1}{pRepetitions*2+1}\right) m_3 \right\}$

FIGURE 5.3: Running example

of proposition permutations with respect to the placeholders belonging to the template. Then, it substitutes each placeholder in the template with a proposition. Each permutation corresponds to a PIT. According to def. 5, a consequent (antecedent) placeholder can be substituted only with a consequent (antecedent) proposition. For example, in Fig. 5.3, template t_1 has 2 placeholders in the consequent (P1, P2) that can be instantiated only with the consequent propositions $v5$ and $v6$.

To generate all the PITs of a template using the given set of propositions P and the set of placeholders PH, we would have to generate $|P|^{|PH|}$ template instantiations, one for each permutation of propositions inside the placeholders. However, such a naive approach would not consider how templates are structured, forcing the miner to analyse several redundant permutations. For example, permutations generating the assertions $G((v1\ \&\&\ v3)\ ||\ (v2\ \&\&\ v4) \rightarrow X(v7 > 5))$ and $G((v2\ \&\&\ v4)\ ||\ (v1\ \&\&\ v3) \rightarrow X(v7 > 5))$ are equivalent because the $||$ operator is commutative.

To solve the above issue, we have developed an algorithm that generates a reduced set of non-redundant permutations.

The algorithm exploits the structural characteristics of the templates to avoid redundancy. In particular, we remove redundancy in two classes of operators: commutative operators (COM), such as && and $||$, and non-reflexive operators (NR), which are binary operators of the form $left\ \Re\ right$, where if $left == right$ then $left\ \Re\ right$ is equivalent to $left$ (or $right$). Examples of NR operators are the U (until) and R (release). Then, for COM operators the number of permutations generated by HARM is $\binom{\#Propositions}{\#Operands}$, while for NR operators they are $\#Propositions * (\#Propositions - 1)$. For example, by considering the set of propositions $\{v0, v1, v2\}$ and the template $P0$ && $P1$, the resulting permutations are $\binom{3}{2} = 3$, i.e., $v0$ && $v1$, $v0$ && $v2$, $v1$ && $v2$, as && is a COM operator, while for the template $P0\ U\ P1$ they would be $3 * (3 - 1) = 6$, as U is an NR operator.

Algorithm 1 describes the GEN_PERMS function implemented in HARM to generate the reduced set of permutations according to the requirements mentioned above. The input of the function is the parameter s of type FormulaStruct. This is a tree-like

Algorithm 1 Generate the reduced set of permutations

Input: the structure of the formula
Output: matrix of permutations

```
1:  function GEN_PERMS(FormulaStruct s)
2:      switch s.type do
3:          case PH
4:              return makeDomainMatrix(s.dim.nRows)
5:          case NR
6:              return makeNRMatrix(GEN_PERMS(s.ch[0]), GEN_PERMS(s.ch[1]))
7:          case COM
8:              return makeComMatrix(GEN_PERMS(s.ch[0]))
9:          case *
10:             ret ← GEN_PERMS(s.ch[0])
11:             for i ← 1 to s.ch.size do
12:                 ret *= GEN_PERMS(s.ch[i])
13:             return ret
```

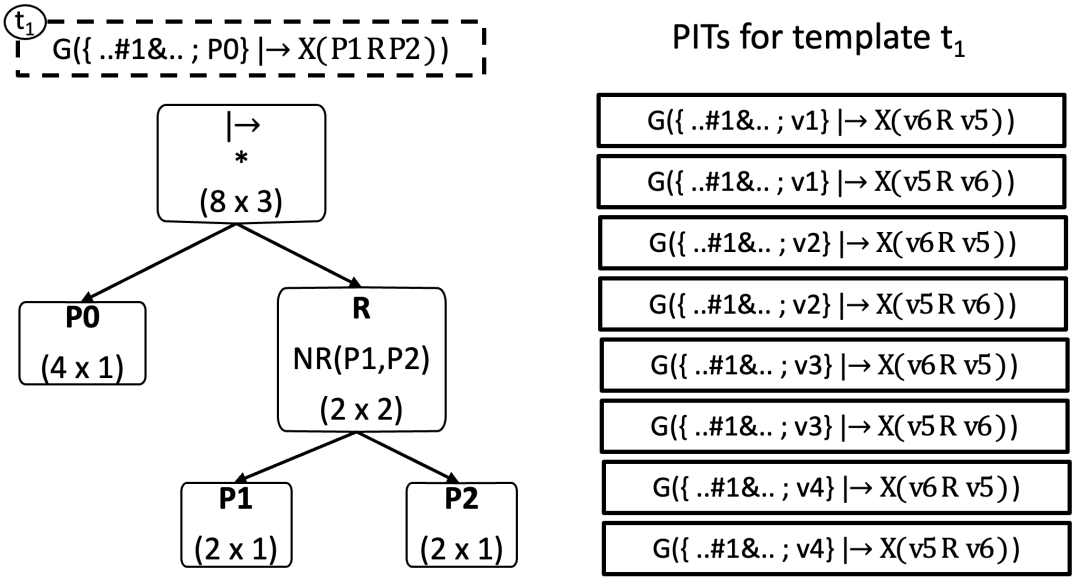

FIGURE 5.4: Permutations in the running example.

data structure generated from the abstract syntax tree of a template. Each node of s is either an operator or a placeholder. The function considers three kinds of operators: COM for commutative operators, NR for non-reflexive operators, and $*$ for all the other types of PSL operators. Each node in s is labelled with the dimension ($row \times col$) of the corresponding permutation matrix, where row is the number of available permutations and col is the number of placeholders to be instantiated. For example, in the left side of Fig. 5.4, the FormulaStruct of template t_1 contains 5 nodes, i.e., $\mapsto$, $P0$, $P1$ R $P2$, $P1$, and $P2$. The node corresponding to the operator $\mapsto$ is labelled with (8×3) to indicate that the 3 placeholders $P0$, $P1$ and $P2$, included in the template, can be replaced by 8 permutations, obtained by combining the 4 permutations related to node $P0$ (which can be replaced by any of the 4 antecedent propositions $v1, v2, v3, v4$) and the 2 permutations related to node $P1$ R $P2$ (which can be replaced by either $v5$ R $v6$ or $v6$ R $v5$). The output of the function is a matrix representing the set of candidate permutations.

GEN_PERMS is a recursive function generating the permutations of a node by compounding the permutations of its children. There are four cases depending on

the type of operator:

- PH: this is the base case of the function where placeholders (i.e., the leaves of FormulaStruct) are handled. It returns a $N \times 1$ matrix using function makeDomainMatrix (line 4). The matrix enumerates from 0 to N-1 the propositions of the corresponding domain (a, c, ac).
- NR: it returns the candidate permutations for the children ($ch[0]$ and $ch[1]$) of a non-reflexive operator by using the function makeNRMatrix (line 6).
- COM: it returns the candidate permutations of a commutative operator by using function makeComMatrix (line 10). Note that makeComMatrix requires as input only the permutations of the first child $ch[0]$ of the operator, as all the other children must yield the same permutations by construction.
- *: it returns the combination of the children's permutations (lines 12-16).

The right side of Fig. 5.4 shows the full list of PITs for template t_1, obtained by instantiating the permutations extracted by GEN_PERMS. Through this algorithm, we avoid generating, for example, the useless permutations $v5$ R $v5$ and $v6$ R $v6$ for the consequent of t_1. The effectiveness of the approach is more evident if we consider the template t_2 of Fig. 5.3: in this case, HARM generates only 15 non-redundant candidates instead of the full set of 256 permutations. The time complexity of this algorithm depends on the type of operators and the number of placeholders and propositions involved; in the worst case, where all permutations are generated through a '*' operator, the worst time complexity is $O(|PH|^P)$, given set of propositions P and the set of placeholders PH.

5.6 Evaluation function

In this section, we describe how HARM generates an evaluation for propositions, PITS and assertions (def. 8). This is a necessary step to implement the assertion mining procedure described in Section 5.7.

To generate an evaluation we apply an evaluation function to the input trace. The evaluation function for a proposition is trivial as its truth values depend only on a single time unit. For example, the evaluation function for proposition $v1$ && $v2$ is a function returning true if both $v1$ and $v2$ hold on a certain time unit, false otherwise. However, a template includes also temporal operators. As a consequence, the corresponding evaluation function must usually consider several time units before returning a truth value. The evaluation function we implemented in HARM for templates is based on an automaton-based representation of the corresponding temporal formulas. Therefore, we employ the framework spotLTL to translate LTL templates to deterministic complete Büchi automata. The advantage of this approach is that automata can be optimised by using state-of-the-art minimisation techniques, improving the performance of the whole evaluation process.

For each template, we generate two automata, one for the antecedent and one for the consequent. This is necessary for two reasons; first, the generation of the contingency table (def. 10) requires knowing the truth values of both the antecedent and the consequent; second, this schema speeds up the evaluation process in scenarios in which only the antecedent (or the consequent) needs to be re-evaluated, such as in our decision tree procedure.

An automaton is composed of states and edges; it always contains a *root* state, an accepting state and a rejecting state; each edge is associated with a proposition, if a

proposition p associated with an edge $e_k(s_{src}, s_{dst})$ connecting state s_{src} with state s_{dst} is true, and s_{src} is the current state, then s_{dst} is the next state. Given an LTL formula f, the corresponding Büchi automaton aut, and a trace tr, a trivial way to determine the truth value of f at time i consists of the following steps:

1. Current state is equal to the root state of aut;
2. Find edge $e_k(s_{cur}, s_{next})$ whose proposition is true at time i, if s_{next} is an accepting (rejecting) state, then f is true (false) at time i and the procedure ends, else go to the next step;
3. Current state is now equal to s_{next}, if $i+1$ is greater than the length of tr then f is unknown at time i, else repeat step 2 at time $i+1$, .

By repeating the previous procedure for all the time units of the trace, we obtain the evaluation of f with respect to tr. However, in the worst-case scenario, each execution of the above procedure starting at time i might have to consider all the subsequent time units $i+1$, $i+2$, ..., till the end of the trace to return a truth value; therefore, this algorithm is quadratic with respect to the length of the trace, which may result definitely inefficient for very long traces.

In Algorithm 2, we propose a more efficient linear-time function to generate an evaluation that we implemented in HARM. The idea of the algorithm is to group up evaluation units requiring the same operations and to execute such operations only once for the entire group. The inputs of function EVALUATE are a Büchi automaton aut where all the placeholders have been substituted with propositions and a trace tr. $curr$ and $next$ are two vectors of type $[(0, ddl_0), (1, ddl_1), ..., (n, ddl_n)]$, where (i, ddl_i) is a couple whose first element $i \in 0, 1, ..., n$ identifies one state of the automaton, while the second element ddl_i is a double linked list (DLL) of unsigned integers. DLLs are fundamental for keeping the algorithm linear, as they allow us to append a DLL to another DLL in constant time. Each element (i, DLL) contains the list of time units assigned to state i; $curr$ contains the time units evaluated in the current time frame ($time$), while $next$ contains the ones that will be evaluated in the following time frame ($time$+1). Both vectors are initialised with a list (containing $aut.nStates$ elements, that is, one element for each state of the automaton) of empty DLLs (lines 3-4). The algorithm is then, composed of three main parts repeated for each time unit of the trace (line 6):

- In the first part (line 8), the current time unit ($time$) is appended to the DLL containing the time units of the root state ($aut.root$ is the index of the root state). This is the engine of the algorithm, that is, where new time units are added to the data flow.
- In the second part (lines 10-26), the algorithm repeats the following procedure for each element of $curr$ (for each state of the automaton) with at least one time unit in the DLL. The algorithm finds an outer edge in which the corresponding proposition is true on the trace for the current instant of time (lines 13-14). The expression $aut[si.first].outEdges$ returns the outer edges of the state with ID equal to $si.first$, which is the first element of the couple (i, ddl_i). If the selected edge reaches an accepting (rejecting) state, then the time units in $si.second$ are used to generate true (false) evaluation units (lines 16-18 and 19-21), otherwise, they are appended to the DDL of the element of $next$ (lines 22-24) corresponding to the destination state of the selected edge. The expression $ret[tu] \leftarrow true$ stores in evaluation ret that the corresponding formula is true on trace tr at

time tu. After that, the DDL of si is cleared, as all the time units were either moved to *next* or used to generate evaluation units.

- In the third part (lines 28-30), all DLLs of *next* are moved to *curr*, precisely, the $i-th$ DLL of *next* is appended to the $i-th$ DLL of *curr*. This step is meant to load in *curr* the time units accumulated in *next*. This way, the algorithm is ready for the next time frame. The DDL of each si of *next* is cleared as all the time units were moved to *curr* (line 30).

Finally, the remaining time units, for which a truth value can not be inferred (the evaluation goes beyond the length of the trace), are used to generate unknown evaluation units (lines 32-34).

The worst-case time complexity of the algorithm is $(V+E)*traceLength$, where V and E are the numbers of states and edges of the automaton. However, since in all meaningful practical scenarios $((V+E)/traceLength) \approx 0$ (which means that *traceLength* is orders of magnitude larger than $V+E$), we can conclude that the time complexity of the algorithm is linear with respect to the length of the trace.

5.7 Assertion mining

In the second step of the methodology, we use PITs to generate assertions. The procedure differs depending on the content of the template.

If a PIT *pit* does not contain any DT operator (def.6) and the evaluation of *pit*, according to the EVALUATE function proposed in the previous section, does not contain any false values, then *pit* is an assertion that holds on the input trace.

If a PIT contains a DT operator, assertions are then mined through a DT algorithm as follows. As indicated after def. 6, an uninstantiated DT operator is equivalent to *true*. Since DT operators appear only in the antecedent, an evaluation of a PIT including an uninstantiated DT operator usually contains several evaluation units in which the antecedent is true and the consequent is false. Then, instantiating a DT operator consists of narrowing the antecedent pool of true values to make the implication hold on the input trace. This is achieved by substituting the *true* constant with a more constrained expression by using the DT algorithm. For example, in Fig.5.5 we show a PIT containing an uninstantiated DT operator (..##1..) in the antecedent. The evaluation of the PIT (by considering the trace in Fig. 5.3) shows that there are four evaluation instants in which the antecedent is true and the consequent is false. By applying the DT algorithm, the *true* constant represented by the DT operator is replaced by $!v4$ ##1 $v3$ ($!v4$ and $v3$ are the operands of this expression) making the assertion hold on the trace. This concept can be applied to any LTL operator, as long as each time a new operand is added to the expression by the DT algorithm, the number of true values of the antecedent is reduced. For instance, the operator || would not be a suitable DT operator, because, being an "or" operator, each new operand would widen the number of true values of the antecedent. On the contrary, && (being an "and" operator) would work perfectly for the opposite reason. In HARM, we have generalised and formalised this idea by classifying the LTL operators complying with the above constraint into "Prop", "Temp" and "Mixed" DT operators.

- *Prop* DT operators have only a propositional dimension; in our grammar we have ..&&.. $= \{o_1$ && o_2 && ... && $o_n\}$ as a *Prop* DT operator. Each o_i is a proposition.

Algorithm 2 Linear-time evaluation function

```
1:  function EVALUATE(aut, tr)
2:      Evaluation ret
3:      curr ← init(aut.nStates)
4:      next ← init(aut.nStates)
5:
6:      for time ← 0 to tr.length do
7:
8:          append(curr[aut.root].second, time)
9:
10:         for all si ∈ curr do
11:             if isEmpty(si.second) then
12:                 continue
13:             for all edge ∈ aut[si.first].outEdges do
14:                 if edge.evaluate(time, tr) then
15:                     switch aut[edge.dst].type do
16:                         case Accept
17:                             for all tu ∈ si.second do
18:                                 ret[tu] ← true
19:                         case Reject
20:                             for all tu ∈ si.second do
21:                                 ret[tu] ← false
22:                         case default
23:                             append(next[edge.dst].second, si.second)
24:                     break
25:             clear(si.second)
26:
27:         for all si ∈ next do
28:             append(curr[si.first].second,si.second)
29:             clear(si.second)
30:
31:     for all si ∈ next do
32:         for all tu ∈ si.second do
33:             ret[tu] ← unknown
34:     return ret
```

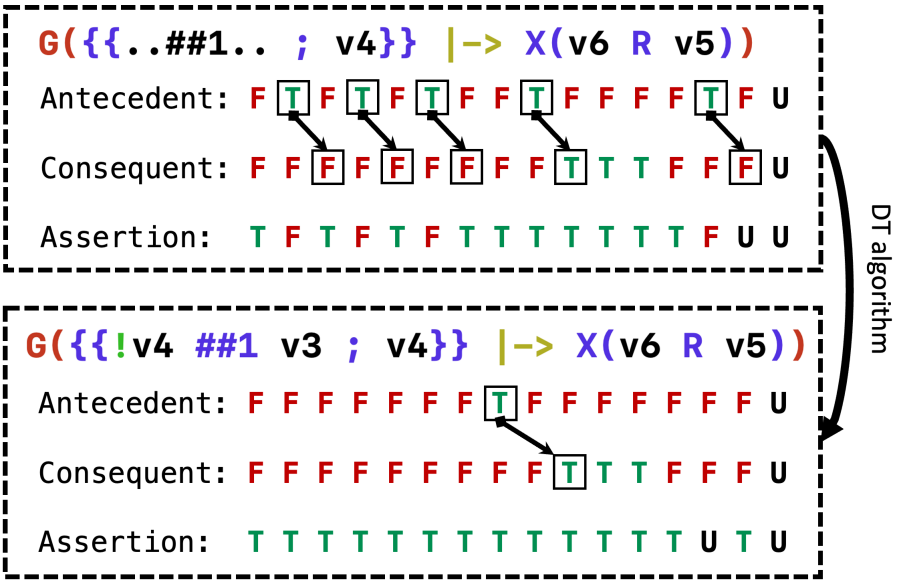

FIGURE 5.5: DT narrowing.

- *Temp* DT operators have only a temporal dimension; we have implemented $..\#\#N.. = \langle o_1\ \#\#N\ o_2\ \#\#N\ ...\ \#\#N\ o_n \rangle$ as a *Temp* DT operator. The user can define N to configure the temporal delay between o_i and o_{i+1}.
- *Mixed* DT operators have both the temporal and propositional dimension; we implemented $..\#N\&.. = \langle (..\&\&..)_1\ \#\#N\ (..\&\&..)_2\ \#\#N\ ...\ \#\#N\ (..\&\&..)_n \rangle$ as a *Mixed* operator. This operator behaves like the sum of the propositional and the temporal dimension of the previous two DT operators.

Each DT operator is associated with a configuration $(Size, CType, Range, Offset)$ involving several adjustable parameters:

- *Size* is a tuple $(TempSize, PropSize, AllSize)$ containing the maximum overall size $AllSize$ (in terms of the number of operands) of the generated expression, and the maximum number of *Temp* operands $TempSize$ and *Prop* operands $PropSize$.
- *CType* is a binary parameter stating if a DT operator with a temporal dimension must construct expressions following a sequential or not ordered approach. To understand this, consider the DT operator ..##2.. with $TempSize$ equal to 3; the resulting expression must follow the implicit template o_1 ##2 o_2 ##2 o_3; however, the order in which o_1, o_2, o_3 are substituted changes the outcome of the DT algorithm. A sequential DT operator substitutes the operands in order from o_1 to o_3 while a not ordered DT operator can substitute operands in any order. The first can only generate the expressions o_1, o_1 ##2 o_2, o_1 ##2 o_2 ##2 o_3, while the latter can generate expressions such as o_1 ##4 o_3 or ##4 o_3.
- *Range* is a numeric parameter to adjust the number of candidates selected by the DT algorithm to split the search space.
- $Offset$ is a binary parameter stating if the algorithm must return the assertions belonging to the offset; such assertions are obtained by negating the consequent of an implication that is false each time the antecedent is true ($G(ant \rightarrow !con)$), making the implication always true on the trace.

The main advantages of our approach with respect to earlier works are that:

- we offer a variety of DT operators;
- our DT algorithm is more configurable;
- our DT operators are more flexible as they appear in a template-based context. For instance, the user can apply the DT algorithm by using a complex and partially instantiated template such as $G(\{prop_1[=1]$ ##7 $prop_2$ ##1 ..&&..$)[*3]\} \rightarrow con)$.

5.7.1 The DT algorithm

In this section, we show how the DT algorithm generates assertions by replacing the DTO with a concrete expression. As implied by the name, the algorithm employs a DT procedure where each decision consists of adding a new operand (proposition) to the DT expression. Operands are chosen according to their information gain (IG), that is, the expected reduction in information entropy caused by adding them to the DT expression. Information entropy can be thought of as the amount of variance in a dataset; in our scenario of application, the entropy is maximum (1) when each time the antecedent is true, we have a 0.5 probability of having the consequent being also true; conversely, the entropy is minimum (0) when each time the antecedent is true than the consequent is always true (onset) or always false (offset). The algorithm always chooses the operand(s) that produces the highest reduction in entropy (the highest IG) in order to mine assertions using as few operands as possible, avoiding over-constrained expressions. Since an operand op_i can only be either true or false (boolean expressions), each decision involves both the operand "as it is" (op_i) and its negated version ($!op_i$). In algorithm 3 we report the functions implementing the above idea.

Function DT_ALGO takes as input a PIT *pit* containing a DT operator ($dtOp$) and a set of decisional propositions (dp) stored as couples (p, $!p$) that will be used in the DT. This function involves three main steps. First, it checks if the current instantiation of $dtOp$ is a solution already found on another path of the tree (lines 6-7), if that is the case, the current node is ignored and the function returns. Second, it finds the candidates used to perform a new decision in the tree (lines 9-15); the function FIND returns the information gain of choosing proposition $dp[j]$ to perform the next decision. Note that there are two scenarios depending on whether the dtOp must be constructed unordered or sequentially; in the first case, candidates are searched among all the temporal propositions ($depth$) in the range $[0, dtOp.maxDepth]$ (lines 11-13), in the latter case candidates are searched only for the last position (either temporal or propositional) denoted by -1 (line 15). After that, the found candidates are sorted in increasing order of IG and filtered of all elements not belonging to the range $[maxIG - Range, maxIG]$ (line 17); if $Range$ is < 0 then only the candidate with maximum IG $maxIG$ is kept. In the third step, a new branch of the tree is created for each remaining candidate (lines 19-27), and both versions of the decisional proposition ($cand[i]$ at line 25, where i is 0 or 1) are used. The *leaves* variable is used to keep track of the propositions that were used in the previous nodes of the current path to generate assertions, these propositions can not be used to generate a new branch. To understand this, consider a path in which proposition $v1$ was used to generate an assertion $G(v1 \rightarrow consequent)$ and the current DT expression is $v2$ && $v3$, if we added $v1$ to such expression, we would generate a correct assertion $G(v1$ && $v2$ && $v3 \rightarrow consequent)$ but we would also render decisions $v2$ and

Algorithm 3 DT algorithm

```
1:  function DT_ALGO(dp, pit)
2:      dtOp ← pit.getDT()
3:      leaves ← ∅
4:      igs ← ∅
5:
6:      if isKnownSolution(dtOp) then
7:          return
8:
9:      if (dtOp.nOperands < dtOp.allSize then
10:         for j ← 0 to dp.size do
11:             if isConstructedUnordered(dtOp) then
12:                 for depth ← 0 to dtOp.maxDepth do
13:                     igs.add(FIND(dp[j], pit,leaves,depth))
14:             else
15:                 igs.add(FIND(dp[j], pit, leaves, −1))
16:
17:     sortAndFilter(igs);
18:
19:     for all ig ∈ igs do
20:         cand ← dp[ig.id]
21:         for i ← 0 to 1 do
22:             if !dtOp.isTaken(ig.id, i, ig.depth) then
23:                 dtOp.addLeaf(cand[i], depth)
24:                 leaves.add(cand[i], depth)
25:                 dtOp.addItem(cand[i], ig.depth)
26:                 DT_ALGO(dp, t)
27:                 dtOp.popItem(ig.depth)
28:
29:     removeLeaves(leaves,dtOp)
30:
31: function FIND(cand, pit, leaves, depth)
32:     dtOp ← pit.getDT()
33:
34:     for i ← 0 to 1 do
35:         if dtOp.isTaken(cand, i, depth) then
36:             continue
37:
38:         dtOp.addItem(cand[i], depth)
39:         eval ← pit.evaluate()
40:         evalAnt ← pit.evaluateAnt()
41:
42:         if getATCT(eval) == 0 || getATCT(eval) == getAT(evalAnt) then
43:             dtOp.addLeaf(cand[i], depth)
44:             leaves.add(cand[i], depth)
45:             store(pit, dtOp)
46:         dtOp.popItem(depth)
47:
48:     if ((!dtOp.isTaken(cand, 0, depth) ||
49:         !dtOp.isTaken(cand, 1, depth)) && ig < 1) then
50:         ig ← computeIG(pit)
51:         return {cand, ig, depth}
52:     return ∅
```

$v3$ pointless as $v1$ was already enough to make the antecedent always implying the consequent. The *leaves* variable is used in conjunction with function *isTaken* (line 22) to determine if a proposition should be used or not. Function $FIND$ is responsible

for finding the candidates and storing the found assertions. The function consists of two main parts. First, each time the DT expression makes the implication hold on the whole trace (getATCT(*eval*)==getAT(*eval Ant*)) or it makes the implication always false (getATCT(*eval*)== 0), a new onset or offset assertion is respectively generated using the *store* function (lines 38-47). The *store* function is also responsible for minimising the DT expression by keeping only the minimum subset of operands making the assertion hold on the trace. Second, if the current candidate contains at least one of the two propositions of *cand* that is not already used to generate an assertion in the current path (lines 49-50), the candidate's information gain *ig* is computed (line 51) and returned together with *cand* and *depth* (line 52).

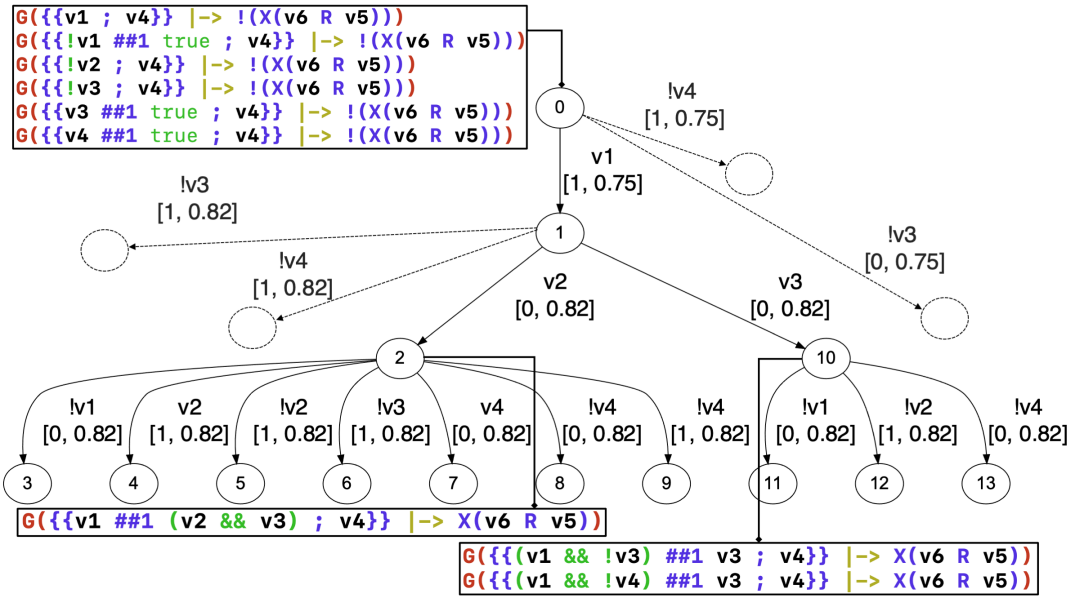

FIGURE 5.6: DT of the running example

Fig. 5.6 shows the resulting DT and mined assertions obtained by applying the DT algorithm to pit_2; the configuration of the DTO is $((2,2,3),0,Unordered,1)$. The branches that do not yield any assertions were removed from the tree to make the picture fit in the paper, these branches are represented by edges connecting to "empty" nodes. The black boxes contain the assertions generated at the corresponding node. Node 0 generates assertions with only one proposition replacing the DT expression as no decisions were already made on the corresponding path. Likewise, node 2 generates assertions containing $v1$ and $v2$ as these are the decisions made on the corresponding path. Each edge is labelled with the used proposition, the temporal position in which the operand was added and its information gain. The given DTO configuration influences the resulting DT in the following ways: the tree has only 3 levels because *allSize* was set to 3; all decisions made on the same node have the same IG value because *Range* was set to 0; the algorithm is allowed to generate assertions with empty spots represented by *true* constants because the construction policy is set to *Unordered*; the algorithm generates offset assertions (see the box connected to node 0 in Fig. 5.6) because the corresponding flag was set to 1. Note that proposition $!v1$ $[1, 0.75]$ is not used in the first decision at node 0 because it was already used to generate assertion $G(!v1$ ##1 $true$; $v4 \mid \rightarrow !X(v6\ R\ v5))$; the same idea is true for $v3$ $[1, 0.82]$, $v4$ $[1, 0.75]$ on the same node and for several other propositions in the tree.

The computational cost of the DT algorithm is dependent on the number of propositions, the employed DT operators, the length of the trace and the aforementioned parameters. The temporal complexity is summarised as follows: ..&&.. operator: $2^{AllSize} * traceLength$; ..##.. operator: $|P|^{AllSize} * traceLength$, where P is the number of "a" propositions in the context; ..#&.. operator: $2^{TempSize+PropSize} * traceLength$. Note that in any meaningful practical scenario, the worst-case scenario never occurs as the entropy-based heuristic only selects a small subset of the possible decisions for every level of the tree.

5.7.2 3-level parallelisation

The formalisation of the mining problem adopted in HARM allowed us to heavily parallelise the generation of assertions. Particularly, we have implemented a 3-level parallelisation algorithm capable of exploiting the additional cores of a CPU to speed up the computation. The idea behind this procedure is that the following are independent processes that can be parallelised:

Level 1: Generation of an evaluation by dividing the evaluation units among different threads. E.g., in the running example one thread can handle evaluation units from t_0 to t_7 while another thread can operate from t_8 to t_{15}.

Level 2: Generation of assertions from different permutations of the same template.

Level 3: Generation of assertions from different templates.

The algorithm starts with an initial number of available threads. First, it divides them among the templates defined in the user context (Level 3); the templates that received at least one thread are elaborated in parallel. After that, the distribution of threads continues for each template (level 2); a template with n threads will generate assertions for n permutations in parallel. If a template has more threads than permutations, the additional threads are distributed among the evaluation functions in each permutation (level 1). Each time the generation of assertions for a permutation or a template is concluded, the assigned threads are returned to the upper levels. Fig.5.7 shows an example of this process where 32 threads are used to generate assertions for the running example. There are only two templates; therefore, each template initially receives 16 threads at level 3. At level 2, the template on the left side produces 8 permutations and each one receives 2 threads. At level 1, each permutation can produce evaluations using 2 threads as mentioned before.

5.8 Qualification

The last step of the methodology consists of filtering and measuring the quality of the generated assertions. Mined assertions are labelled with a ranking score by using the metrics provided by the user in his/her input context. After that, they are filtered and sorted in increasing order according to their scores. The employed metrics are user-defined numeric expressions; the user can define each metric by using various built-in assertion features. For example, in Fig. 5.3, the metric m_3 combines arithmetic operators with *pRepetitions*, which is a built-in assertion feature measuring the number of repeated propositions. Through this mechanism, the user is free of defining the metrics by measuring the characteristics of an assertion he/she is interested in. In this process, metrics can be used either to filter or sort the assertions. The running example contains two sorting metrics (m_2, m_3) and one filtering metric

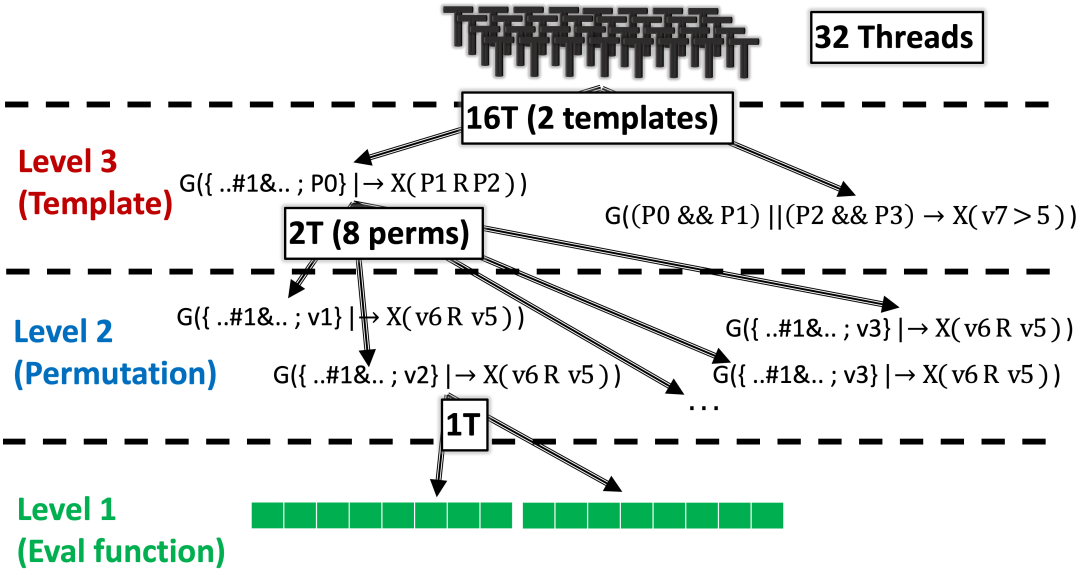

FIGURE 5.7: 3-level parallelization of the running example

(m_1). Filtering metrics are associated with a threshold; assertions with a score below the threshold of any filtering metric are directly discarded. In the running example, all assertions with a score of m_1 less than 0.45 are ignored.

Sorting metrics are used to perform the ranking. The ranking is computed according to an overall score. This is calculated, for each mined assertion a, through the following formula $\prod_{i=1}^{n} calibrate(sm_i(a)/sm_i(a_{max\_i}))$, where $sm_i(a)$ is the score of a by using the $i-th$ sorting metric, a_{max_i} is the assertion that yields the maximum score by using metric sm_i, and *calibrate* is a procedure that "calibrates" the input score by using function $R = 1/(1 + e^{(z-kx)})^2$. In the chart on the left side of Fig. 5.8, we show the graphical representation of function R with z equal to 3.3 and k equal to 10.62. This function is a modified version of the Richards' curve that ranges from 0 to 1. The intuition behind this ranking formula is that we want to allow the simultaneous employment of multiple sorting metrics in a single ranking procedure. Furthermore, we want to give more importance to assertions presenting a higher score for all sorting metrics, while penalising assertions that score well only in a subset or in none of the given metrics. The *calibrate* function is capable of making lower scores greatly undermine the final ranking (1) while preventing high scores from compensating for lower scores (2).

The aforementioned values of k and z yield the standard *calibrate* function used in the miner; however, we allow the user to choose between 55 different configurations of k and z, to adjust the effect of (1) and (2) on the final ranking. In the chart on the right side of Fig. 5.8, we show 10 configurations of the *calibrate* function, where the first function from the left returns values greater than 0 from 0 to 1, the second does the same thing from 0.1 to 1, the third from 0.2 to 1, ..., the tenth from 0.9 to 1. Other configurations may include functions ranging from 0.3 to 0.8, from 0.5 to 0.6, et cetera. The computational complexity of performing the ranking is linear with respect to the number of ranked assertions.

Fig. 5.9 shows the final ranking of the assertions generated for the running example. Note that all assertions belonging to the offset of template t_1 were discarded by the filtering metric. As expected, assertions not presenting a good score for all sorting metrics have received a low ranking (assertions 1-12). Assertions 1-4 received the maximum ranking in one metric (pRepetitions), however, it was not enough to

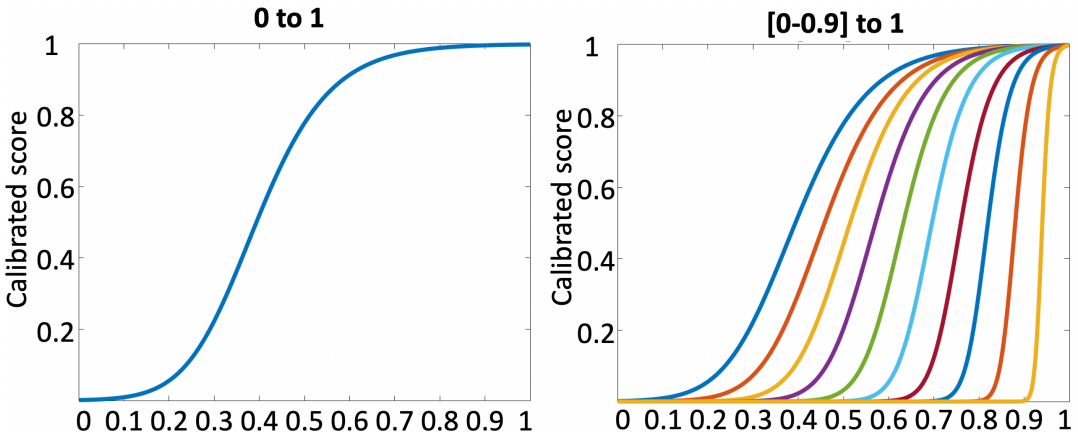

FIGURE 5.8: Score functions

N	Assertion (Context : default)	final	frequency	pRepetitions
0	G(((v1 && v2) || (v3 && v4)) -> X(v7 > 5))	1.00	1.00	1.00
1	G(((v1 && v3) || (v2 && v4)) -> X(v7 > 5))	0.20	0.33	1.00
2	G({{(v1) ##1 (v2 && v3) ; v4}} |-> X(v6 R v5))	0.20	0.33	1.00
3	G({{(!v3 && v1) ##1 (v3) ; v4}} |-> X(v6 R v5))	0.20	0.33	1.00
4	G({{(!v4) ##1 (v3) ; v4}} |-> X(v6 R v5))	0.20	0.33	1.00
5	G(((v1 && v2) || (v1 && v3)) -> X(v7 > 5))	0.19	0.67	0.33
6	G(((v1 && v2) || (v1 && v4)) -> X(v7 > 5))	0.19	0.67	0.33
7	G(((v1 && v3) || (v3 && v4)) -> X(v7 > 5))	0.19	0.67	0.33
8	G(((v1 && v4) || (v3 && v4)) -> X(v7 > 5))	0.19	0.67	0.33
9	G(((v2 && v4) || (v3 && v4)) -> X(v7 > 5))	0.19	0.67	0.33
10	G(((v1 && v2) || (v2 && v4)) -> X(v7 > 5))	0.04	0.33	0.33
11	G(((v1 && v3) || (v1 && v4)) -> X(v7 > 5))	0.04	0.33	0.33
12	G(((v1 && v4) || (v2 && v4)) -> X(v7 > 5))	0.04	0.33	0.33

FIGURE 5.9: Running example final ranking

compensate for the low score in the other (frequency); furthermore, they received a final score very close to assertions 5-9 (0.20 vs 0.19) even though having a far higher score with the frequency metric (1 vs 0.67).

With respect to earlier techniques, our ranking approach has the following advantages:

- It is independent from the source code of the DUV;
- It is flexible and configurable;
- It allows the identification of interesting assertions by using multiple metrics at the same time.

On the other hand, our approach presents the drawback of being less user-friendly, since choosing the correct set of metrics for a certain use case might not always be trivial.

5.9 Mining assertions containing non-boolean expressions

In this section, we show how to automatically mine assertions containing non-Boolean expressions. In the last few decades, several tools have been proposed to mine assertions with various heuristics; however, these approaches generally have to deal with

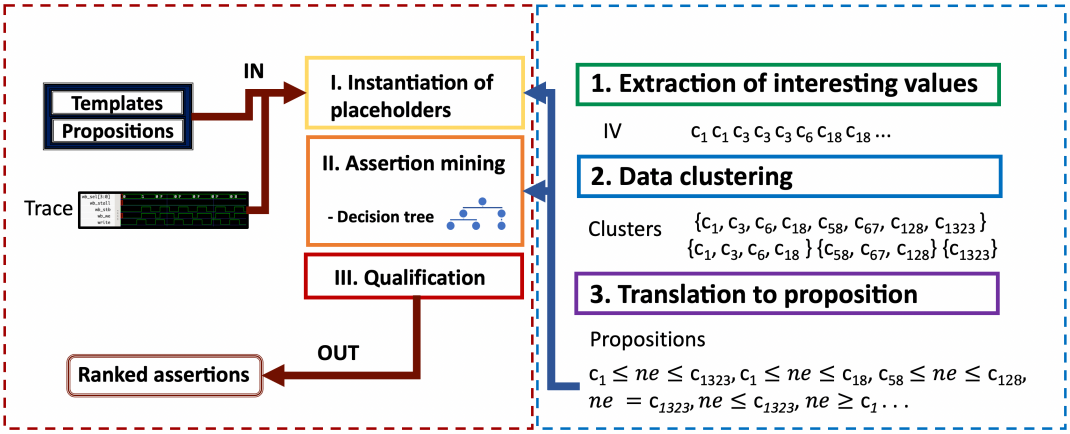

FIGURE 5.10: Overview of the methodology: Original HARM implementation (on the left) and the proposed extension (on the right).

the following dilemma: generating assertions that describe the temporal relation between Boolean expressions (e.g., $always(x \rightarrow next(y))$, where x, y are Boolean variables) or generating assertions outlining non-temporal relations between more complex propositions involving arithmetic and/or relational operators (e.g., $(a + b > c)$, where a, b, c are numeric variables whose arithmetic/logic relation is evaluated as invariant). In general, it is considered difficult to mine assertions presenting both characteristics (e.g., $always(x \rightarrow next(a + b > c))$). Intuitively, an algorithm that commits to such a goal would have to solve two orthogonal problems at the same time, that is, computing both the temporal and the propositional layers of an assertion. State-of-the-art approaches try to outflank the problem by prompting the user to define one of the two layers (temporal or propositional); then, they exploit an automatic tool to generate assertions by completing the missing layer.

In this section, we propose a method to generate LTL assertions by automatically extracting both the temporal and the propositional layers containing non-Boolean expressions.

In particular, starting from a set of simulation traces of the DUV, our method is capable of generating LTL assertions in the form $always(antecedent \rightarrow consequent)$, where *antecedent* and *consequent* are composed by combining temporal operators with propositions predicating over arithmetic expressions, like $c = ne$, $c \leq ne$, $c \geq ne$, $c_l \leq ne \leq c_r$, with c, c_l, c_r representing constants of numeric type, and ne indicating numerical expressions involving DUV variables. The mining technique exploits a clustering algorithm to determine a meaningful set of propositions following the aforementioned structure. After that, the propositions are used to mine temporal assertions by exploiting a decision tree algorithm. It is then possible to automatically mine assertions like $always(x \rightarrow next(a + b > c))$[2].

In addition to the execution trace of the DUV, the set of templates, and the set of propositions, which represent the original input of HARM, our methodology requires the user specifies also a set of tuples $N = \{(ne_i, loc_i, th_i) | i = 1, \ldots, k\}$, where ne_i is a numeric expression, loc_i is a location label (among "a", "c", "ac", and "dt") as described below Def. 2, and th_i is a numeric threshold (from 0 to 1) that is used to specify how much effort the tool must put in to generate propositions including the numeric expression ne_i. For instance, in the running example reported in

[2]This is just an example of assertions that we can extract. The tool does not have any limitation in the number and kind of LTL temporal operators that can be instantiated in the mined assertions.

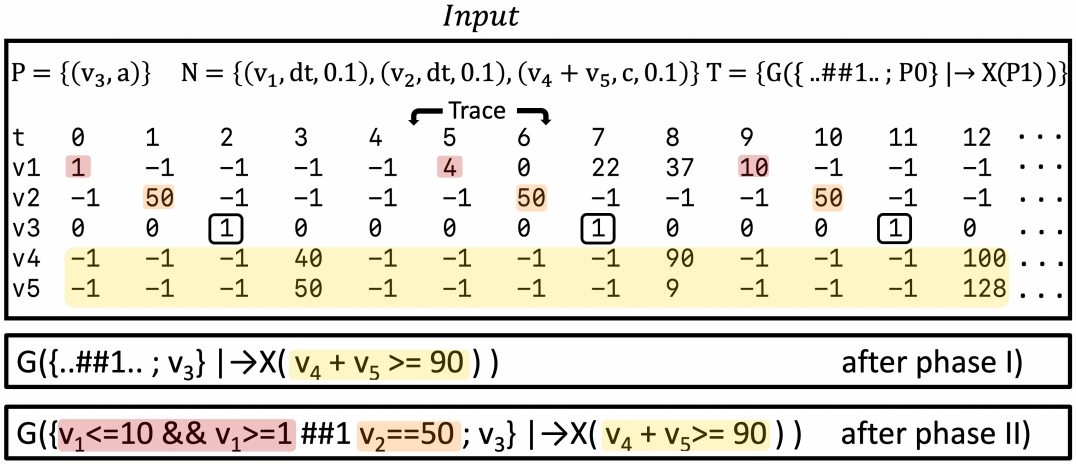

FIGURE 5.11: Running example: P, N and T, together with the simulation trace of the DUV, are the input of the tool and represent, respectively, the set of propositions, the set of numeric expressions, and the set of templates to be used for mining temporal assertions on non-Boolean expressions.

Fig. 5.11, we include $(v_4 + v_5,\ c,\ 0.1)$ in the set N to specify that we want to substitute placeholders in the consequent with a proposition constructed by using the numeric expression $v_4 + v_5$, like, for example, $v_4 + v_5 \geq 90$, or $v_4 + v_5 \leq -2$.

The architecture of our approach is shown on the right side of Fig. 5.10 and it is composed of the following 3 steps, which extend phase I (instantiation of placeholders) and phase II (assertion mining) of the original HARM implementation:

1. First, given a target template, we extract a set of interesting numeric values (IVs) to be used in accordance with each numeric expression ne_i provided as input.

2. After that, we apply a clustering algorithm to generate a set of numerical ranges for each set of IVs.

3. Finally, we translate the ranges into a set of propositions of the form $c = ne_i$, $c \leq ne_i$, $c \geq ne_i$, $c_l \leq ne_i \leq c_r$, with c, c_l, c_r representing constants of numeric type.

5.9.1 Extraction of interesting values

In the first step, the tool gathers a set of IVs for each target numeric expression ne. To obtain an IV at time t_i, we evaluate ne on the input trace at time t_i. If we are generating propositions for phase I (instantiation of placeholders) of HARM, we extract IVs by exploiting the whole trace, that is, by using the values of ne for every time t_i. However, if we are generating propositions for phase II (assertion mining), then we proceed differently: given a location inside a DT operator, that is, a place where a new proposition p_{new} must be inserted by means of the decision tree algorithm, we consider only the time units t_i in which a true evaluation of p_{new} at t_i would be "necessary" to make the antecedent of the target template hold at t_i. To determine if an instant t_i is necessary to extract an IV, we perform the following steps. First, we substitute the next location in the DT operator with a *true* constant and we evaluate the antecedent at time t_i. If the antecedent evaluates to true, we substitute the *true* constant with the $false$ constant and evaluate again the antecedent. If the antecedent

is no longer true at time t_i, then we can conclude that the value of the numeric expression *ne* at t_i is an IV, since it is necessary to satisfy the antecedent. With the above procedure, in phase II, we are able of generating IVs by considering a reduced number of time units. This is possible because, in phase II, all the placeholders are instantiated and the decision tree operator (Def. 6) is the last remaining location to be substituted, which represents a single variable problem independent from other factors. On the contrary, in phase I, we may have multiple dependent placeholders: one can not be substituted without affecting the substitution of the others.

To make things clearer, consider the template $G($ ..##1.. ; $P0 \mid \rightarrow X(P1))$ of the running example of Fig. 5.11. In phase I of HARM, $P0$ and $P1$ must be substituted with propositions. $P0$ is an antecedent placeholder, and then it can be substituted only with $v3$. On the contrary, $P1$ is a consequent placeholder, thus it can be substituted only with a proposition originating from $v_4 + v_5$, like, for example, $v_4 + v_5 \geq 90$. To achieve that, we extract IVs by using all the time units of the trace involving variables $v4$ and $v5$, thus providing candidate values -2 from -1 + -1, 90 from 40 + 50, 99 from 90 + 9 and 228 from 100 + 128, (repetitions are considered) to be combined with $v_4 + v_5$ by using relational operators $==, \leq, \geq, \ldots$.

In phase II of HARM, instead, the DT operator ..##1.. is instantiated by using a decision tree algorithm (DTA). Let us assume that the DTA produced the partial instantiation $G($ *nl* ##1 $v2 == 50$; $v3 \mid \rightarrow X(v_4 + v_5 >= 90))$, where *nl* is the next location that the DTA will try to substitute with a new proposition. To do that, our approach generates IVs considering all the instants of the trace in which *nl* must evaluate to *true* to make the antecedent hold on the trace; these values are 1, 4 and 10 for variable $v1$ at time t_0, t_5 and t_9 respectively (we report only the values observable in the partial trace of Fig. 5.11). Note how values 22 and 37 for $v1$ at time t_7 and t_8 are not considered, as the antecedent would be *false* at time t_7 and t_8 regardless of the value of $v1$.

5.9.2 Clustering of interesting values

In the second step of the methodology, we generate a set of ranges $[left, right]$ (with $left \leq right$) by applying a clustering algorithm to the IVs retrieved in the previous step. We apply Lloyd's version of the k-means algorithm [56], whose worst-case time complexity is $O(nkid)$, which becomes $O(n)$ if we fix the number of clusters k and the maximum number of iterations i, and we apply the algorithm to uni-dimensional data ($d = 1$). Therefore, it can be easily applied to sets of data with millions of values. The k-means algorithm requires the number of clusters as input; however, for a generic set of IVs, this information is usually not available beforehand. To solve this issue, we apply the elbow method, a well-known cluster analysis heuristic used to determine the number of clusters in a data set. When using the elbow method, the k-means algorithm is executed multiple times: the idea consists of measuring how the variance v inside the generated clusters diminishes for an increasing k. In most cases, if we plotted v for every value of k in a line plot, we would observe an "elbow-like" line as in Fig. 5.12. The value k, at which the reduction of variance plateaus, is considered a good candidate. In our approach, to algorithmically identify the elbow of the variance, we use the threshold *th* associated with the target numeric expression; it specifies the minimum percentage reduction of variance below which we consider the procedure to have reached the elbow. We execute the k-means algorithm until the variance plateaus according to *th*; for each execution, we keep track of generated clusters (we do not keep multiple copies of the same cluster). When

the above procedure has finished, we transform each cluster to a range $[left,\ right]$, where $left$ and $right$ are respectively the smallest and largest value in the cluster.

For example, Fig. 5.12 refers to the clustering procedure applied to element $(v_1, dt, 0.1) \in N$ of Fig. 5.11. As the target threshold for v_1 is 0.1, the elbow is reached if the reduction of variance goes below 10% (note that in our implementation, we decided to use the standard deviation instead as it is easy to handle numerically and provides a better intuition of the approach), which happens after running the k-means algorithm with $k = 4$, as the reduction of standard deviation from $k = 3$ to $k = 4$ is only 6.6%, i.e., (1.5-1.4)/1.5 (see Fig. 5.12 on the right). The upper-left part of Fig. 5.12 shows, in particular, the ranges obtained through the clustering procedure at varying of k (numbers over black background are the minimum and the maximum per each range).

In the final step of the methodology, the ranges found in the previous step are used to generate propositions (see bottom-left part of Fig. 5.12). For each range $[left,\ right]$, we generate propositions $ne \leq right$, $ne \geq left$, $ne \leq right$ && $ne \geq left$; if $left == right$ then we generate proposition $ne == left$. Even though this final step looks simple, it requires some clarifications.

Why do we generate propositions of the form $ne <= right$, $ne >= left$? Since the clustering procedure generates a set of finite ranges, it would be reasonable to generate only propositions of the form $ne <= right$ && $ne >= left$. However, there are scenarios in which that approach would not work. Consider an input trace that contains the following functional behaviours formalised through two assertions: $G(v1 >= 1 \text{ \&\& } v1 <= 100 \mid \rightarrow X(out1))$ and $G(v1 >= 50 \text{ \&\& } v1 <= 150 \mid \rightarrow X(!out2))$. Let us assume that the expression $out1$ && $!out2$ is true for a large number of units of time and that we want to mine the aforementioned assertions starting from templates $G(..\&\&.. \mid \rightarrow X(out1))$ and $G(..\&\&.. \mid \rightarrow X(!out2))$. If we apply the clustering algorithm to instantiate ..&&.. for one of the two templates, the data used to perform the clustering contains two overlapping ranges [1, 100], [50, 150] of IVs; as a result, the data has a high density of values in the range [50, 100] (the overlapping part). Therefore, if we allow only propositions of the form $ne <= right$ && $ne >= left$, the clustering algorithm does not generate the expected propositions as it is not able to untangle the overlapping data; it would probably generate the proposition $p = v1 >= 50 \text{ \&\& } v1 <= 100$. However, p contains exactly the expressions we are looking for. If we break down p into two propositions $v1 >= 50$, $v1 <= 100$, then we are capable of completing the ranges in the next clustering cycle. For instance, if the tool generates the partial instantiation $G(v1 <= 100 \text{ \&\& } nl \mid \rightarrow X(out1))$, the next time the DTA tries to substitute nl with a proposition, it will only consider the values of $v1$ that are less than 100, allowing the clustering algorithm to generate the range [1, 100] and translate it to $v1 >= 1$, thus mining the expected assertion with $v1 >= 1$ && $v1 <= 100$ as antecedent.

5.10 Experimental results

Evaluating the effectiveness and efficiency of assertion miners is not a trivial task as the quality of the generated assertions is often a subjective matter; additionally, these tools are heavily influenced by their initial configuration, further complicating their overall evaluation. Nonetheless, there are objective measures we can exploit such as fault coverage and execution performances. Our experiments are divided into five parts; in the first and second parts we compare HARM against the well-known Goldmine [29] and A-TEAM [51] miners, with respect to fault coverage and scalability; in

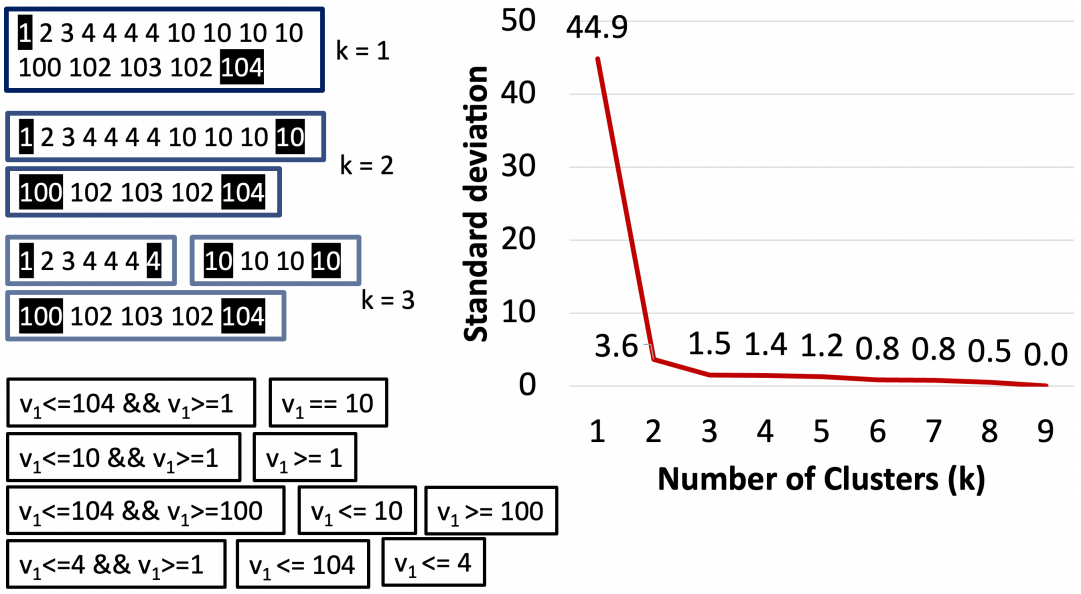

FIGURE 5.12: Example of the clustering procedure.

the third part, we evaluate the behaviour of HARM operating in a multi-threading scenario; in the fourth part, we show the effectiveness of our context-based approach through a concrete use case; in the last part, we evaluate the extension for mining assertions with non-boolean variables.

The experiments have been carried out on a 3.5 GHz 16-core AMD Ryzen 3950x processor equipped with 32 GB of RAM (3600 MHz) and running Ubuntu 20.04 LTS.

5.10.1 Fault coverage

The first set of experiments compares the fault coverage achieved by assertions mined using HARM, Goldmine and A-TEAM on five RTL designs. These designs are used by the developers of Goldmine to showcase the tool. They are available at [57]. For each design, we have inserted a set of mutants and re-simulated the mutated design by activating only one mutant for each simulation. This way, each generated faulty trace is affected at most by a single fault. Note that we have kept only the faulty traces in which the fault was observable, that is, in which the values on the outputs were different compared to the original unaltered trace. The considered mutants are: 1) “bit flip" for bit-vector variables, where a bit of a variable is negated; 2) “operator swap" for relational, arithmetic, boolean and bit-wise operators, where the original operator is changed with a compatible one. For example, the original expression $v1$ && $v2$ is changed to $v1$ || $v2$ to generate a mutant.

We measure the fault coverage by counting how many faults are covered by the mined assertions (an assertion covers a fault if it fails on the corresponding faulty trace). To enable a fair evaluation of the two tools, we have run the experiments under the following constraints, which are due to the characteristics of Goldmine:

- Goldmine can extract only assertions compliant with the template $G(\{..\#1\&..\}| \rightarrow P0)$. Thus, the same template has been provided to HARM and A-TEAM;
- Goldmine does not support non-Boolean data types, thus only Boolean variables have been considered;

Design	Complexity		
	Lines	I/O	Faults
(1) arb2	28	6	7
(2) id_stage	484	82	546
(3) decoder	97	4	368
(4) controller	459	57	602
(5) multdiv	230	15	1780

TABLE 5.2: Complexity of the designs

Design	Assertions			Coverage			Min subset			AVG Coverage			Time to mine		
	H	G	A	H	G	A	H	G	A	H	G	A	H	G	A
(1)	12	6	6	100%	100%	100%	2	2	2	3.5	3.5	3.5	0.3s	2.4s	2s
(2)	3160	2803	2405	82%	41%	68%	182	102	201	2.5	2.2	1.8	8.3m	23.5m	13.4m
(3)	701	431	501	52%	39%	40%	54	39	58	3.5	3.7	2.5	3s	19s	5s
(4)	4909	1010	2079	84%	48%	56%	132	79	93	3.8	3.7	3.6	5.8s	12m	17s
(5)	1722	682	1255	92%	51%	79%	387	222	344	4.2	4.1	4.1	22s	1.3m	49s

TABLE 5.3: Comparison between Goldmine, A-TEAM and HARM: fault coverage

- the maximum depth of the assertion has been set to 3 cycles, and the maximum number of propositions in the antecedent has been set to 5 for all tools;
- the tools are all single-threaded.

Table 5.2 reports the complexity of the five benchmarks in terms of the number of code lines (*Lines*) and the total number of primary inputs/outputs (I/O) of the design, and the number of injected observable faults (*Faults*) Table 5.3 reports the results for the five benchmarks using HARM (H), Goldmine (G) and A-TEAM (A), in terms of the number of generated assertions (*Assertions*); the percentage of faults covered by the mined assertions (*Coverage*); the minimum number of assertions covering the maximum number of faults (*Min subset*); the value ((N_faults * Coverage)/ Min_subset), higher is better, to describe how effective the mined assertions are at covering the faults (*AVG coverage*); the time spent by the tools for mining the assertions (*Time to mine*). For all benchmarks, traces composed of 1000 clock cycles have been provided to the tools. Overall, HARM consistently managed to reach a higher fault coverage for all benchmarks except for (1), where all tools achieved the same coverage. In terms of *AVG coverage*, HARM generally produced an equal or better covering set. This suggests that HARM is generally able of generating a less constrained set of assertions with higher coverage. One noticeable difference is that HARM is faster than the other tools at mining assertions.

5.10.2 Scalability

In this section, we compare the scalability of the three tools with respect to the length of the input trace. The tests are executed under the same constraints defined in the previous section but with a progressively increased length of the input trace; in particular, we executed all five designs by using four traces that are 1000, 10000, 100000, 1000000 clock cycles long. We have set 12 hours (43200s) as the time limit. The chart in Fig. 5.13 reports the results of this evaluation. The x-axis reports the four lengths of the trace while the y-axis reports the time in seconds (logarithmic scale).

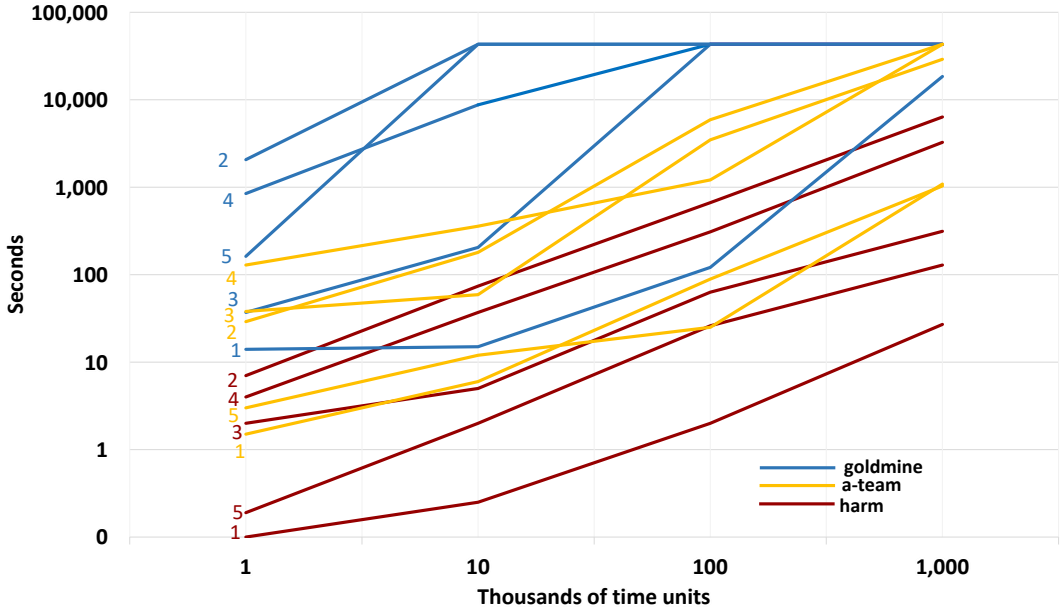

FIGURE 5.13: Comparison between Goldmine, A-TEAM and HARM: Scalability

It is clear by analysing the data that HARM is remarkably faster than the other tools. *arb*2 was the only design on which Goldmine terminated before the time limit by considering a 1 million clock cycle-long trace, taking over 5 hours to complete, whereas HARM took only 34 seconds. A-TEAM appears to behave as a middle ground between the faster HARM and the slower Goldmine. Overall, HARM proves to be extremely scalable with respect to the length of the trace while Goldmine and A-TEAM look reasonably fast only for short traces.

We do not report results in terms of memory usage because they are dominated by the amount of memory required to hold the input trace; besides, the amount of memory required by HARM can not cause scalability issues as it is always a linear function of the input.

5.10.3 Multi-threading evaluation

In this section, we analyse the speed-up guaranteed by applying the 3-level parallelisation described in Section 5.7.2. Since this procedure has three dimensions, we have analysed the speed-up provided by each level separately.

In Fig. 5.14 we report the average results of parallelising all the designs. The x-axis reports the number of threads available in each test while the y-axis reports the speed up of the test. The green, blue and red lines correspond to the results of applying levels one, two and three respectively. Since the achieved speedup is also dependent on the length of the trace, we have included the results of using a 1k, 10k and 100k long traces, these correspond to the dotted, dashed and solid lines respectively. It is important to note that all speedups start deteriorating after 16 threads as all the tests have been executed on a 16-core machine.

For the first level (green lines), we have executed HARM with a single template producing a single permutation, making the evaluation function the only element benefiting from multi-threading. The results at this level are heavily affected by the length of the trace: at 1k (dotted line), we obtain a negative speed up (< 1) as the

overhead of multithreading is not compensated by dividing the work among several threads; at 10k (dashed line) the overhead starts to stabilise and we obtain a 1.5x speedup; at 100k (solid line) the overhead completely stabilises and we obtain a 3x speed-up with 16 threads. The results of applying the second and third levels are quite similar (blue and red lines) as their parallelization follows the same principle. In this case, we have executed the tests with a single template generating 16 permutations for level 2, and with 16 templates generating a single permutation for level 3. As shown in the chart, the best results are obtained at 100k (blue and red solid lines) where we achieve a 9x speed-up with 16 threads. We also include the results of using the "fault-coverage configuration" (black lines) as a more practical scenario where we use all three levels together.

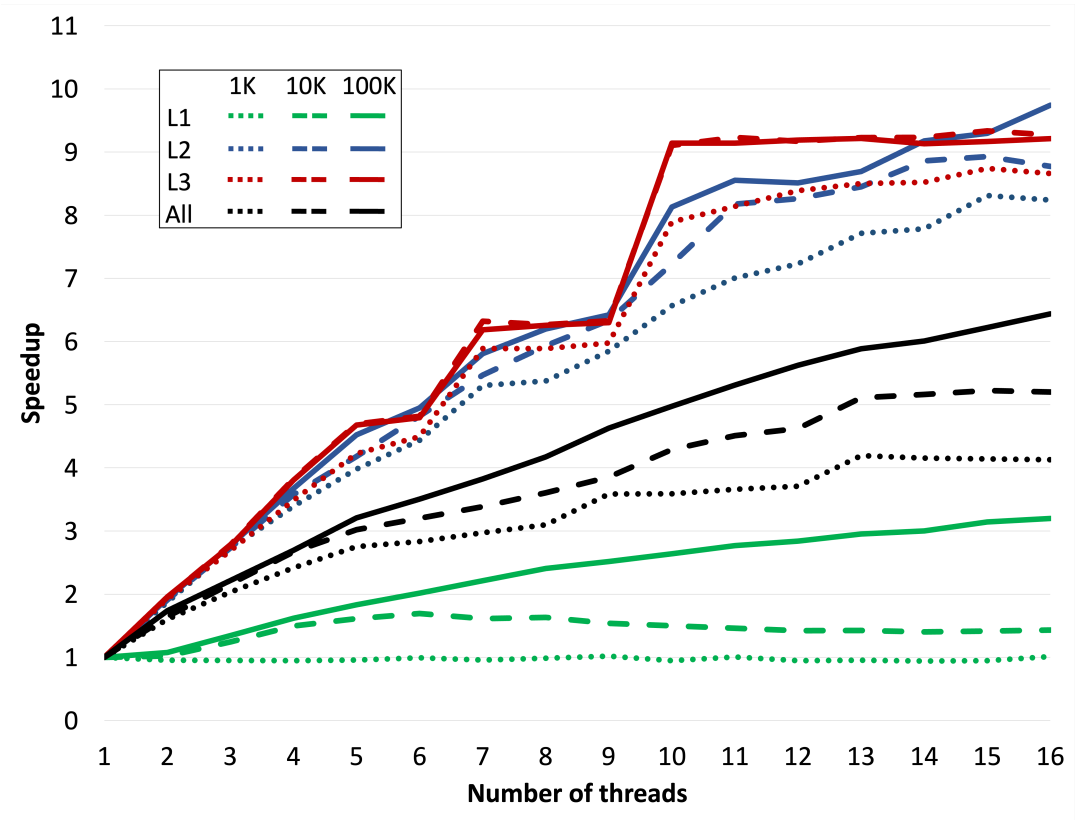

FIGURE 5.14: Speedup of the 3-level parallelisation

5.10.4 Applying the context-based approach

In this section, we show the effectiveness of applying our context-based ranking approach to a concrete use case. The use case consists of mining assertions for an RTL design containing a hardware trojan (HT). Our objective is to show that the assertions identifying the behaviour of the HT are ranked higher than the other assertions if the user provided the correct metrics. To perform this test, we have selected the designs listed in table 5.4 implementing the RSA encryption algorithm. These designs are available at [58]. According to the HT taxonomy, *rsa*100 and *rsa*300 contain a "leak-information" HT while *rsa*200 contains a "denial of service" HT; all three Trojans are activated through the user input. The trace used to mine the assertions is

Design	Trace length	N assertions	Position of interesting assertion	Time to mine
rsa100	834k	689	1st	19m
rsa200	1074k	1440	1st	116m
rsa300	840k	227	1st	30m

TABLE 5.4: Results of applying the context-based approach

generated by simulating the design with a test bench performing thousands of encryptions; additionally, the generated trace contains at least one activation of the HT. To rank the generated assertions, we have provided the following sorting metrics:

- Causality: $(1 - AFCT/traceLength)$,
- Frequency: $(1 - ATCT/traceLength)$,
- Complexity: $(complexity)$.

The first metric is to reward assertions representing a real correlation between antecedent and consequent, as assertions with a high $AFTC$ in the contingency matrix, correspond to random correlations or partial behaviours. The second and third metrics reward assertions with low frequency and high complexity respectively, where *complexity* is the number of variables in the assertion.

In this context, we assume that an assertion identifying an HT would have low frequency and high complexity as it shows a rare and hidden behaviour with several constraints. We used template $G(..\&\&.. \rightarrow P0)$ for the first and second designs; template $G(..\#\#100..| \rightarrow P0)$ was employed for the third design. In the first and second designs, we mine the assertions by using 931 propositions, while in the third we have selected 212 propositions. In this scenario, we have a huge amount of propositions; this happens because all variables of bit-vector type are split into many single-bit variables, allowing the miner to generate complex expressions corresponding to bit configurations. HARM was capable of generating assertions catching the behaviour of the injected HT in all three designs. Furthermore, as shown in table 5.4, such assertions are ranked in the first position by using our context-based approach, greatly simplifying the work of the verification engineer, who no longer has to read thousands of assertions to identify a single interesting assertion.

5.10.5 Evaluation of assertions with non-boolean variables

5.10.5.1 Assertion effectiveness

To evaluate the effectiveness of the assertions mined by our approach, we compared the fault coverage achieved by the assertions generated with the original implementation of HARM and with our extension by using five RTL designs available at [59], [60]. The experiments were set up as follows:

- Templates: for both the original HARM and our extended version we use the template $G(\{..\#1\&..\}| \rightarrow X(P0))$.
- Propositions: for the extended HARM, we created a numeric expression with a threshold equal to 0.1 for each non-Boolean variable of the DUV, the rest of the variables were used as atomic propositions; for the original HARM, all variables of the DUV were used as atomic propositions and bit-vectors were split into single bit variables.

Design	Trace length	Time to mine (s.)		N assertions		N faults	Coverage%		Cov. Subset	
		ext-h	h	ext-h	h		ext-h	h	ext-h	h
wishbone	1000	<1	< 1	1711	2122	151	71.0	47.6	18	24
camellia	1000	3	11	207923	8357	718	99.8	36.0	67	24
vgafb	10000	75	11	75167	4177	359	83.2	66.2	33	40
sdram16bit	50000	459	329	53539	47379	1047	88.7	84.3	121	146
pid_controller	16035	84	320	28682	118840	1880	67.8	48.4	102	151

TABLE 5.5: Assertion effectiveness by comparing the fault coverage achieved by the assertions mined with the original version of HARM, predicating over Boolean variable (*h*), and with the proposed extension, which extracts also non-Boolean expressions (*ext-h*).

Design	Time to mine (s.)			N assertions			Coverage %			Cov. Subset		
	k-means	KDE	HC	k-means	KDE	HC	k-means	KDE	HC	k-means	KDE	HC
wishbone	< 1	1	1	1711	2105	5113	71.0	71.0	71.0	18	10	16
camellia	3	10	2	8357	12232	14567	99.8	100.0	100.0	67	56	55
vgafb	75	196	9	75167	38836	8073	83.2	83.2	65.4	33	30	40
sdram16bit	459	>12h	>12h	47370	NA	NA	88.7	NA	NA	121	NA	NA
pid_controller	84	>12h	>12h	28682	NA	NA	67.8	NA	NA	102	NA	NA

TABLE 5.6: Comparison among different clustering algorithms. The value NA refers to cases where the tool exceeded the time limit (12 hours). The best results have been achieved with k-means.

- The rest of the configurations were the same for both scenarios.

Table 5.5 reports the results for the five designs. Columns *h* (harm) and *ext-h* (extended harm) refer, respectively, to the original HARM implementation and the extended version proposed in section 5.9.

The table shows the trace length in terms of the number of clock cycles (*Trace length*), the mining time in seconds (*Time to mine*), the number of mined assertions (*N assertions*), the number of injected faults *N faults*, the percentage of covered faults by the generated assertions (*Coverage*), and the minimum number of assertions covering the maximum number of faults (*Cov. subset*). The reported results highlight a clear trend: our approach achieves the highest fault coverage for all benchmarks. Furthermore, the smaller covering subsets in the *ext-h* column suggest that our approach is able of generating a more meaningful and readable set of assertions, as a lower number of assertions covers a higher number of faults. The time required to mine the assertions is similar for both *ext-h* and *h*, thus suggesting that our approach did not worsen the performance of the tool.

5.10.5.2 Comparison of the clustering algorithms

In the second part of the experiments, we compare the performances of applying our approach with three different clustering algorithms: k-means, kernel density estimation (KDE) and hierarchical clustering (HK).

The experimental setup is the same as reported in Section 5.10.5.1, the only variable is the clustering algorithm. Table 5.6 reports the results for the five designs. Overall, the three clustering algorithms provide similar fault coverages and covering subsets. However, k-means seems more reliable in terms of temporal performances; in fact, both KDE and HK execution time substantially increased for sdram16bit and pid_controller preventing the termination of the mining. This is not surprising considering the higher worst-case temporal complexity of KDE and HK.

Chapter 6

Bug explanation: COME & BECAUSE

6.1 Introduction

Early identification and correction of bugs is a key point in order to save money and speed up the time-to-market of modern embedded systems. In this context, while designers focus on generating a bug-free implementation that meets the specifications, verification engineers check that such an implementation satisfies the initial specifications without including unexpected behaviours.

Thus, many approaches have been developed both from the point of view of the designers and the verification engineers to detect bugs and, more generally, unexpected behaviours in system-level descriptions before they are propagated throughout the lower design levels.

However, when such behaviours are found, the verification engineer still has to understand their cause through a tedious manual inspection of the execution traces of the DUV. In most cases, this process is unnecessarily long since only a few instructions of the execution traces are relevant for understanding and fixing unwanted behaviours.

To fill in the gap, we present two new methodologies and related tools called COME and BECAUSE to automatically remove irrelevant instructions from the execution traces of unexpected behaviours, such that verification engineers can focus on the real cause of the problem when debugging their DUV.

6.2 Related work

In the last decades, several methodologies, mainly in the software field, have been proposed to tackle the aforementioned problem.

A well-known technique to perform fault localisation and bug explanation is, in particular, program slicing.

The original notion of a program slice was proposed by Weiser [61]. Weiser defined a program slice as a reduced program obtained from a program p by removing statements, such that the slice replicates part of the behavior of p. Program slicing techniques fall into two main categories: static and dynamic program slicing. A static slice is computed without making assumptions regarding the input of the program while a dynamic slice relies on some specific test case. Several techniques have been proposed to produce a static slice using reachability algorithms on program dependency graphs (PDG) [62]–[66]. A PDG is an intermediate program representation to make explicit both data and control dependencies in a program.

Dynamic program slicing was first introduced by Korel and Laski in [67], which allows extracting a (small) executable section of the original program that preserves part of the program's behaviour for a specific input with respect to a subset of selected variables, rather than for all possible computations. One of the most popular applications of dynamic program slicing consists of comparing two or more slices to identify differences or similarities. In [68], the authors present a technique to isolate the region of the bug by computing the difference between a correct slice and the faulty one; likewise, [69] propose an approach to find a correct slice that is the nearest to a related faulty slice. Similar techniques based on intersections and unions between dynamic slices are reported in [70]. In[71] and[72], the authors proposed a solution to error explanation by identifying the differences between an error trace, representing an undesired behavior, and an error-free trace. They employed distance metrics to single out error-free traces that are as similar as possible to the error trace. Then, they use bounded model checking and SAT solving to extract a counter-example trace violating the given specification.

A dynamic program dependency graph is usually employed in conjunction with program slicing as a dynamic variant of a PDG. In a DPDG, dependencies consider a specific occurrence of a certain instruction as there may be several repetitions in a single execution trace. The paper in [73] describes several techniques to exploit a DPDG while performing dynamic program slicing.

Several approaches have been proposed to generate slices by exploiting both static and dynamic information [74]–[79].

Other approaches rely on statistical methods to perform fault localisation [80],[81]. These techniques aim at gathering coverage details of correct and faulty executions over a bugged program, then they rate each programming element in terms of their suspiciousness. In [82] the authors combine dynamic program slicing with statistical methods to build program slicing spectra to rank suspicious elements.

Other approaches involve the use of symbolic simulation.

Symbolic and concolic simulation are techniques exploiting symbolic values to maximize the execution path coverage of a program under validation. KLEE[21] is an approach to symbolically executing Low-Level Virtual Machine (LLVM) code [83]. It can generate high-coverage test cases and it has been successfully applied to find out software errors such as memory overflow in GNU COREUTILS[84]. Different approaches extending or specializing KLEE are then proposed in the literature. FIE[85] is a platform built on top of KLEE for detecting bugs in programs for the MSP4030 family of microcontrollers. FIE can identify two types of errors: memory safety violations, such as buffer overruns, and out-of-bounds accesses to memory objects like arrays, as well as peripheral-misuse errors in which the software under validation writes to a read-only memory location. KleeNet[86] and T-Check[87] are KLEE customization to generate test cases for sensor networks and find safety and liveness errors in sensor network applications running on TinyOS. S2E[88] is a platform built on top of QEMU[89] and KLEE. It can scale to large complex programs, such as a real application on Linux, by exploiting a selective symbolic execution. Moreover, it allows the analysis of properties and behaviors in each simulated execution path (e.g. the number of cache misses) through an external user-defined plug-in. BitBlaze[90] is an early representative work on binary analysis for computer security focused on optimizing the close coupling of concrete and symbolic execution to detect exploitable software bugs. ANGR[91] is a binary analysis framework that implements concrete execution techniques based on fuzz testing[92] and symbolic execution techniques to perform test-case generation and software bug detection. CRETE[93] is another versatile binary-level concolic testing framework. Decoupling

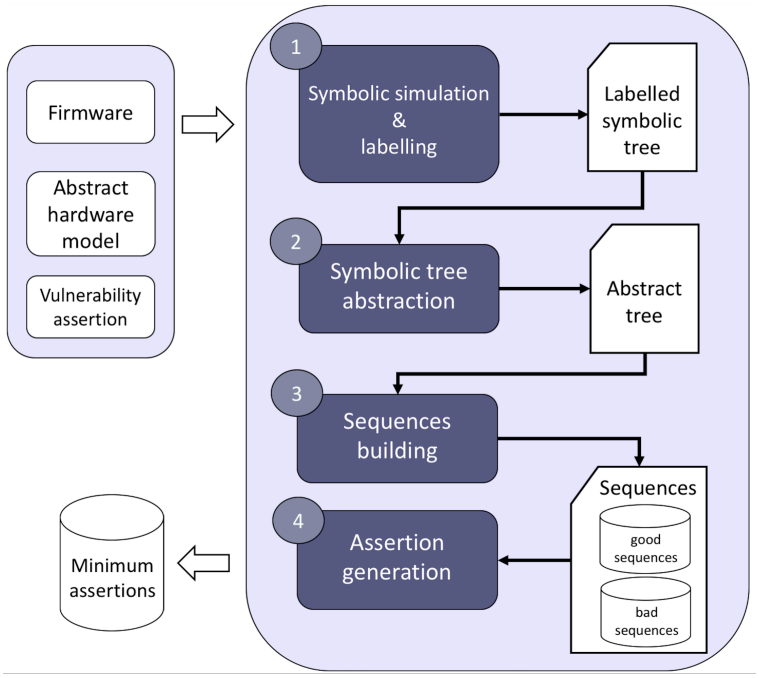

FIGURE 6.1: Execution flow of COME.

between tracing and symbolic simulation, CRETE can dramatically reduce memory usage for test-case generation. Experimental results performed with GNU COREUTILS show that CRETE achieves a comparable code coverage compared to KLEE, and it generally outperforms ANGR. DOVE [94] aims at providing behaviors that may represent firmware vulnerabilities. It exploits KLEE to symbolically execute the binary code of the firmware in an instruction set simulator. Through a model counting-based approach, the execution paths leading to rare events are identified and provided to the user in terms of temporal constraints on firmware input values.

6.3 COME

In this section, we present the first tool called COME. COME focuses on explaining vulnerabilities affecting firmware implementations.

Fig. 6.1 shows an overview of the tool. Given a vulnerability in the form of a propositional logic assertion, which represents the effect of an unwanted behavior of the IP, the framework first relies on the symbolic simulation of the firmware to search for computational paths that satisfy the assertion, thus triggering the vulnerability. Then, it analyzes these computational paths using alignment algorithms to extract the essential instructions actually characterizing the vulnerability, removing unnecessary elements. For each vulnerability, the output is a set of assertions representing the minimum sequence of firmware instructions that trigger the vulnerability, thus allowing the designer to effectively identify its cause.

The rest of the section is organized as follows. In Section 6.3.1 we describe the threat model of the considered vulnerabilities. In Section 6.3.2 we illustrate some helpful preliminary definitions. Section 6.3.3 explains the methodology implemented in COME. Section 6.3.4 reports experimental results.

6.3.1 Threat model

We assume firmware inputs and outputs are any memory location or memory-mapped register addressable by the CPU. We consider two different threat scenarios. In the first scenario, we assume that the attacker is the designer of the application-level software. He/she has full control of the firmware's inputs, therefore an IP vulnerability could be exposed by an inaccurate sanity check of firmware input values by the firmware itself. In the second scenario, the attacker is the designer of the firmware. In this case, we assume he/she intentionally embeds malicious code into the firmware that is eventually triggered through specific sequences of inputs. In both cases, the attacker aims to undermine the integrity of the system. We make no assumptions about the hardware platform executing the firmware. On the basis of this threat model, our approach simulates the binary code of the firmware by using an instruction set simulator of the target hardware. We finally assume that the verification engineer succeeds in detecting the vulnerability, but he/she is not able to easily identify its cause. This is the point where COME begins to operate.

6.3.2 Preliminaries

Our tool exploits a symbolic tree model, whose definition is reported hereafter to simplify the understanding of the following sections.

Definition 13. *A **tree entry** is a triplet $e = \langle O, adr, val \rangle$, where $O \in \{Load, Store\}$, adr and val represent respectively a memory address and a data value. Elements of O define the following operations:*

- *Load represents the load of the value val from the memory location adr into CPU;*
- *Store represents the store of the value val from CPU to the memory location adr.*

Definition 14. *A **tree node** $n = \langle e_1, e_2, \ldots, e_n \rangle$, is a finite sequence of tree entries.*

Definition 15. *A **tree arc** is a triplet $a = \langle n_s, n_d, c \rangle$, where n_s, n_d are, respectively, the source and the destination nodes, and c is a constraint satisfied by the firmware moving from n_s to n_d.*

Definition 16. *A **symbolic tree** is defined as $ST = \langle N, A, n_0 \rangle$, where N is a set of tree nodes, A is a set of arcs defining the transition functions among nodes, and n_0 is the root node of the tree.*

Fig. 6.2 shows an example of symbolic tree. Starting from node 0, we can observe the first entry $\langle Load, 0x4000, 0x1 \rangle$ which describes a load operation where the value 0x1 is read from address 0x4000. After that instruction, the execution proceeds linearly until a branch is formed in which different constraints are given to the same symbolic variables generating different possible execution paths. In this example, the first branch generates only two possible paths, in the first path the symbolic variable X_5 can assume only the value 1 while in the second path it can assume all values other than 1. Symbolic variables are shown in the form X_i with i $\in \mathbb{N}^*$.
We introduce hereafter the function *getSource* for a symbolic tree and a tree node that will be used in Algorithm 4 of section 6.3.3.

Definition 17. *Given a symbolic tree ST, and a node n_d, the function $getSource(ST, n_d)$ returns the arc $\langle n_s, n_d, c \rangle$ of ST. For the root node n_0, the value nil is returned as a result.*

We introduce hereafter the concept of *equivalence* between arithmetic and logic expressions, which will be then recalled in definition 19 concerning the *equivalence* between tree entries. Intuitively, two expressions r_1 and r_2 (*e.g.* $r_1 = x_1, r_2 = x_2 + x_3$) with constraints c_1 and c_2 (*e.g.* $c_1 : \ 0 \leq x_1 \leq 2, c_2 : \ (0 \leq x_2 \leq 1) \wedge (0 \leq x_3 \leq 1)$), are equivalent when the set of feasible values for r_1 and r_2 is the same (*e.g.* $r_1, r_2 \in \{0, 1, 2\}$). Formally:

Definition 18. *Let r_1 and r_2 be two arithmetic and logic expressions, $X = (x_1, x_2, \dots x_k)$ the set of variables involved in r_1, $Y = (y_1, y_2, \dots y_l)$ the set of variables involved in r_2, c_1 and c_2 constraints, respectively, over X and Y, r_1 and r_2 are **equivalent** if and only if:*

$$\forall X(c_1(X) \rightarrow \exists Y(c_2(Y) \wedge r_1(X) = r_2(Y))) \quad \wedge \tag{6.1}$$

$$\forall Y(c_2(Y) \rightarrow \exists X(c_1(X) \wedge r_1(X) = r_2(Y))) \tag{6.2}$$

Formula 6.1 (6.2) affirms that for each assignment of the symbolic variables X (Y) satisfying the constraints c_1 (c_2) there exists an assignment of the symbolic variables Y (X) satisfying the constraints c_2 (c_1) such that r_1 and r_2 have the same evaluation. When both Formula 1 and Formula 2 are satisfied, then r_1 and r_2 are said equivalent in the proposed approach.

Definition 19. *Given a tree entry e_1 and a constraint c_1, a tree entry e_2 and a constraint c_2, we assert that e_1 and e_2 are equivalent when $e_1.O = e_2.O$, and the arithmetic and logic expressions $e_1.adr$ and $e_1.val$ are respectively **equivalent** to $e_2.adr$ and $e_2.val$.*

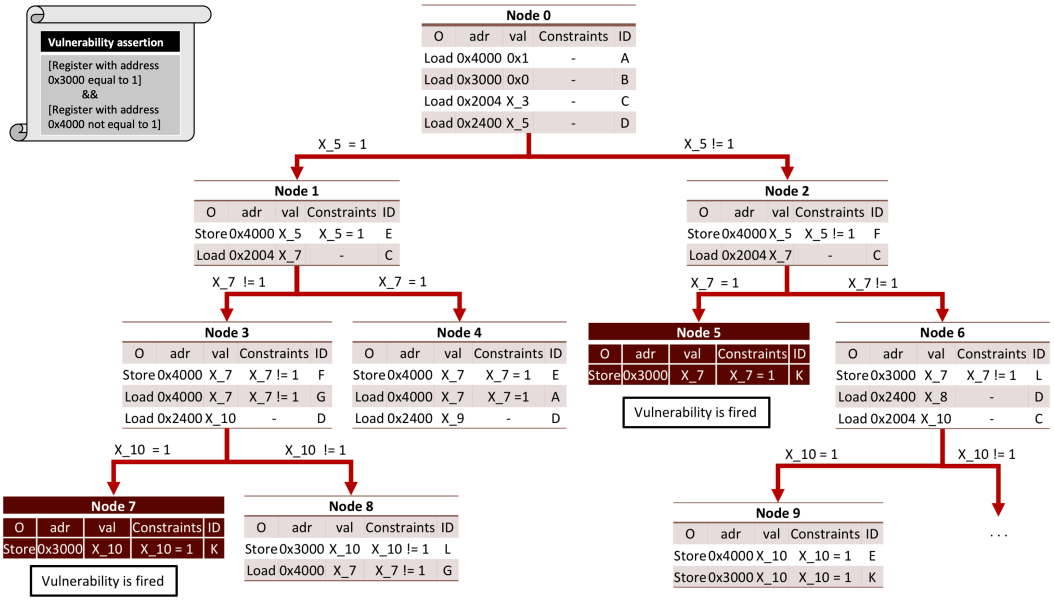

FIGURE 6.2: Example of abstract symbolic tree. Nodes 5 and 7 are marked with the label "Vulnerability is fired" since when they are reached, the input assertion is satisfied.

6.3.3 Methodology

As shown in Fig. 6.1, the proposed framework is composed of four main functions that are executed in a cascade fashion. The inputs of COME are a *firmware* in binary code, an *abstract model* of the hardware where the firmware is executed, and

a propositional logic *assertion*, which represents the effect of an unwanted behavior of the IP. COME is independent from the origin of such an assertion. It can be extracted automatically by assertion mining (e.g., by using DOVE) or manually defined by the verification engineer starting from counterexamples generated during a previous verification process. The goal of COME is to understand why such an unexpected behavior occurs during some execution flows to guide the designer in fixing the related vulnerability. The output is then the set of temporal assertions that represent the minimum sequence of firmware instructions triggering such behavior.

(1) Symbolic simulation & labelling: The first phase consists of identifying the different firmware execution flows in which the vulnerability assertion provided as input is satisfied. This goal is achieved by using a symbolic simulation engine to concurrently execute different firmware execution paths in the hardware abstract model. If an execution path fires the vulnerability assertion then the symbolic simulation ends for that path by marking its last node in the symbolic tree with the label "vulnerability is fired". The result of this step is then a labelled symbolic tree as shown, for example, in Figure 6.2, where nodes 5 and 7 are labelled, since the paths ending with them satisfy the vulnerability assertion.

(2) Symbolic tree abstraction: As a second step, the previously generated symbolic tree is analysed to identify the equivalent tree entries according to Definition 19. In particular, COME applies a semantic equivalent checking strategy to compare data and address values involved in load/store operations. Operations managing the same set of values are marked by the same identifier. The result of this step is an abstracted symbolic tree, in which each operation is mapped with a unique identifier.

(3) Sequence building: In this phase, each path of the abstract symbolic tree is visited, from the leaf to the root, to describe each simulated firmware execution path as a sequence of identifiers. A "good-sequences" set is generated by including paths in which the vulnerability assertion is never satisfied. Instead, a "bad-sequences" set is generated by including the paths in which the vulnerability assertion is satisfied.

(4) Assertion generation: The last phase is intended to generate assertions identifying the shortest sequences of operations, which lead the firmware to satisfy the provided vulnerability assertion. This goal is achieved by a pairwise alignment algorithm identifying the shortest subsequence of identifiers included in "bad-sequences" but not in "good-sequences".

To intuitively clarify the idea underpinning our methodology, before entering the technical details reported in the next subsections, let us consider the labelled symbolic tree shown in Fig. 6.2. It corresponds to the output of phase (1) applied to a simple example. In this example, the vulnerability is triggered by satisfying the condition "Register with address 0x3000 equals 1 and register with address 0x4000 is different from 1". The symbolic simulation represented in the tree of Fig. 6.2, reaches two states (nodes 5 and 7) where such a vulnerability is fired. Thus, the two paths from node 0 till, respectively, node 5 and node 7 represent two "bad sequences". The purpose of the following phases of the methodology is then to reduce the number of instructions contained in such bad sequences to understand the actual cause that fires the vulnerability. By executing phase (2) of our approach, all entries of the symbolic tree are marked with an identifier. Then, in phase (3), a sequence of such identifiers is generated for each path from the root to the target node. For example, the bad sequence from node 0 to node 7 corresponds to the sequence of identifiers {A,B,C,D,E,C,F,G,D,K}. In the last phase, each bad sequence is filtered to extract the shortest sequence of instructions able to satisfy the target vulnerability. In Fig. 6.2, the output of step 4 for both the bad sequences is then {F,K}. Indeed, those identifiers

coincide with the instructions $\{\langle Store, 0x4000, X\_7 \neq 1\rangle, \langle Store, 0x3000, X\_10 = 1\rangle\}$ which correspond to the shortest sequence of instructions satisfying the target vulnerability.

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

6.3.3.1 Symbolic simulation and labelling

In the first step of the proposed methodology, COME detects in which firmware execution paths, a vulnerability, represented by a propositional logic assertion, can be exploited. A vulnerability assertion is an arithmetic/logic proposition describing an undesired value for a memory location or a CPU register. The goal is achieved by a symbolic simulation engine, which explores all the execution paths of the firmware to label as "vulnerable" the nodes of the symbolic tree where the assertion holds. We integrated COME within an in-house modified version of Klee to symbolically simulate the firmware and to label the nodes. As symbolic simulation is a well-known concept, we do not report further details for this step of our approach. Unfamiliar readers can refer to [95] for a survey on symbolic simulators. The vulnerability represented by the assertion can be manually defined by the user, or automatically extracted by a verification tool, like, for example, the one proposed in [94]. The result of this step is a symbolic tree called labelled symbolic tree in which nodes exposing the firmware to a given vulnerability are labelled as "vulnerable nodes". Intuitively, a labelled symbolic tree is a symbolic tree where leaves marked as "vulnerable" describe execution paths that satisfy the vulnerability assertion as more formally specified in the next definition.

Definition 20. *Given a vulnerability assertion va, a* ***labelled symbolic tree*** *is defined as* $LST = \langle N, N_{va}, A, n_0\rangle$*, where N is a set of tree nodes,* $N_{va} \subseteq N$ *is a set of tree nodes where the assertion va is satisfied, A is a set of arcs defining the transition function among nodes, and* n_0 *is the root node of the tree.*

The symbolic simulation engine explores all feasible execution paths for the given firmware inputs. Each firmware's input is seen as a symbolic variable that represents a range of possible values that could be received from the firmware primary inputs, leading the execution to different paths. We won't describe in detail how the symbolic simulation is performed which is considered a well-known approach in the verification community. In addition to the papers already mentioned in Section 6.2, the reader is free to go deeper into the topic by reading [95] which provides a clear summary of symbolic and concolic simulation and their applications.

6.3.3.2 Symbolic tree abstraction

In the second step of the proposed methodology, COME aims at characterizing the equivalent operations performed by the firmware through a unique identifier. The proposed approach works in two steps: first an augmented symbolic tree is defined to augment each tree entry *e* with the conjunction of the constraints satisfied along the arcs belonging to the execution path traversed to reach *e*. Then, an abstract symbolic tree is defined by marking each *tree entry* with an identifier. Tree entries satisfying the same constraints are equivalent (see Definition 19), and are therefore marked with the same identifier.

6.3.3.3 Generation of the augmented symbolic tree

Informally, an augmented symbolic tree aims at mapping a tree entry e of a node n with a constraint $c^\wedge$ which is the logical conjunction of constraints c_i belonging to tree arcs connecting, from the root, each node of the path where e occurs. For example, by collecting the constraints for the first entry of node 8 in Fig. 6.2, we obtain $c^\wedge = \{X\_5 = 1 \wedge X\_7 \neq 1 \wedge X\_10 \neq 1$. COME repeats this operation for each entry of the symbolic tree to obtain the augmented one. We report hereafter the formal definition of augmented symbolic tree.

Definition 21. *An **augmented symbolic tree** is defined as $AG\_ST = \langle ST, C \rangle$, where ST is a symbolic tree, and $C : N \times E \times N \rightarrow C^\wedge$ is a function that maps a tree entry e of a node n and a leaf m with a constraint $c^\wedge = c_1 \wedge c_2 \wedge ... \wedge c_k$.*

Note that $c^\wedge$ could be equal to the constant *true* if the address (*e.adr*) and the value (*e.val*) in entry e contains only constant values or if no relevant constraints are found for the symbolic variables inside those registers. Algorithm 4 shows how an augmented symbolic tree is generated from a symbolic tree. At the beginning, set C_{set} representing the input-output relation corresponding to function C of Def. 21 is empty. Then, for each leaf node m of ST, the algorithm calls the recursive function *propagateConstraints*. This function takes as input a symbolic tree ST, a leaf node m, a node n which is initially equals to m, and a set of constraints T, which is initially set to $\emptyset$ (empty). *propagateConstraints* collects T, the set of all constraints satisfied by the firmware from leaf-node m to root-node n_0. In lines 12-16, it first gets the source node n_s of n by using the function *getSource* (line 13), then it enlarges T with the satisfied constraint c. Afterwards, it recursively calls itself (line 15) to climb back the tree from node n_s. When n is eventually equal to the root-node n_0 (line 12), all constraints are gathered for this path. In lines 17-20, *propagateConstraints* iterates over each entry e of current tree node n. In this phase the function aims to add the element $\{\langle n, e, m \rangle, c^\wedge\}$ (line 19), which maps n, m and e with a constraint $c^\wedge$, to the set C'_{set}. The generation of the constraint $c^\wedge$ relies on the procedure $simplify(e, T)$, which takes as input a tree entry e and a set of constraints $T = \{c_1, c_2 ... c_n\}$. $simplify(e, T)$ aims to return a constraint $c^\wedge$ in propositional logic of the form $c_1 \wedge c_2 \wedge ... \wedge c_k$ where $T_{min,e} = \{c_1, c_2, \ ... \ , c_k\} \subseteq T$ is the minimum set of relevant constraints for entry e. Intuitively we keep only the constraints that can have an effect on the symbolic variables contained in e. Relevant constraints are then reduced in order to decrease their quantity while keeping the same logical meaning. If $T_{min,e} = \{\emptyset\}$ then $simplify(e, T)$ returns the constant *true*. To clarify this procedure, we will complete the example where we collected the constraints for the first entry of node 8 which resulted in the set $\{X\_5 = 1, X\_7 \neq 1, X\_10 \neq 1\}$. In this scenario, the reduction step would result in $c^\wedge = \{X\_10 \neq 1\}$ as all the other constraints do not affect symbolic variables contained in this specific entry.

6.3.3.4 Generation of the abstract symbolic tree

Informally, an abstract symbolic tree (AB_ST) aims at mapping a tree entry e with an identifier k. The same k is applied for equivalent tree entries. We report hereafter the formal definition of AB_ST.

Definition 22. *An **abstract symbolic tree** is defined as $AB\_ST = \langle AG\_ST, M \rangle$, where AG_ST is an augmented symbolic tree, and $M : N \times E \times N \rightarrow K$ is a function that maps a tree entry e of a node n and a leaf node m with an identifier k.*

Algorithm 4

```
1: function augmentedSymbolicTree(ST)
2:     C_set = ∅
3:     for all leaf nodes m in ST do
4:         C_set = C_set ∪ propagateConstraints(ST, m, m, ∅)
5:     AG_ST = ⟨ST, C_set⟩
6:     return AG_ST
7:
8: function propagateConstraints(ST, n, m, T)
9:     T = ∅
10:    if n ≠ ST.n_0 then
11:        ⟨n_s, n, c⟩ = getSource(ST, n)
12:        T = T ∪ {c}
13:        C'_set = propagateConstraints(ST, n_s, m, T)
14:    for all e ∈ n do
15:        c^∧ = simplify(e, T)
16:        C'_set = C'_set ∪ {⟨n, e, m⟩, c^∧}
17:    return C'_set
```

Algorithm 5

```
1: function abstractSymbolicTree(AG_ST)
2:     M_set = ∅
3:     for all {⟨n_i, e_i, m_i⟩, c_i^∧} ∈ AG_ST.C_set do
4:         k = newIdenfier()
5:         for all {⟨n_j, e_j, m_j⟩, k_j} ∈ M_set do
6:             c_j^∧ = AG_ST.C_set(⟨n_j, e_j, m_j⟩)
7:             if equal(e_i, c_i^∧, e_j, c_j^∧) then
8:                 k = k_j
9:                 break
10:        M_set = M_set ∪ {⟨n_i, e_i, m_i⟩, k}
11:    AB_ST = ⟨AG_ST.ST, M_set⟩
12:    return AB_ST
```

Algorithm 5 shows how an abstract symbolic tree is generated from an augmented symbolic tree. At the beginning the set M_{set} representing the input-output relation corresponding to function M of Def. 22 is set to $\emptyset$ (line 2). Then, for each element $\{\langle n_i, e_i, m_i\rangle, c_i^\wedge\}$ of the set C_{set} in the augmented symbolic tree AG_ST (lines 3-13), the algorithm generates a new identifier k (line 4). Afterwards, it checks whether an element $\{\langle n_j, e_j, m_j\rangle, k_j\}$ exists in M_{set} such that e_i and e_j are equivalent (lines 5-11). In detail, at line 6 the algorithm extracts from C_{set} the constraint $c_j^\wedge$ of $\langle n_j, e_j, m_j\rangle$. Through the procedure $equal(e_i, c_i^\wedge, e_j, c_j^\wedge)$ (line 7), it checks if e_i and e_j are equivalent. If the equivalence test succeeds then the identifier k_j is assigned to k (line 8). At line 10 the element $\{\langle n_i, e_i, m_i\rangle, k\}$ is added to the set M_{set}. Eventually, an abstract symbolic tree is generated as $\langle AG\_ST.ST, M_{set}\rangle$ at line 11, and returned as a result at line 12.

The procedure *equal* implements the formula at Definition 18 using a SAT solver that checks the satisfiability of the given instance. It gets as input an entry e_i with a constraint $c_i^\wedge$ and an entry e_j with a constraint $c_j^\wedge$, it returns a boolean value. *equal* compares the entries e_i and e_j and returns *true* if they are considered equivalent according to Definition 19, otherwise it returns $false$.

Fig. 6.2 shows an example of abstract symbolic tree where each entry has an identifier abstracting the relative operation. Equivalent entries received the same identifier; e.g., the first entry of node 1 $\langle Store, 0x4000, X\_5\rangle$ and node 9 $\langle Store, 0x4000, X\_10\rangle$ were both assigned with the identifier E as they are equivalent according to Def. 19. In fact, they represent the same type of operation *Store* on the same address $0x4000$ and of the same value 1 forced by the constraints $X\_5 = 1$ and $X\_10 = 1$ on symbolic variables X_5 and X_10.

6.3.3.5 Sequences building

In the third step of the proposed methodology, COME aims at partitioning the simulated firmware execution paths into two sets. “good-sequence" set collects the firmware execution paths where the initially provided assertion was never satisfied. On the contrary, the “bad-sequence" set collects the firmware execution paths eventually ending with the provided assertion satisfied. The proposed approach works in one phase in which each path of the abstracted symbolic tree is visited to describe each simulated firmware execution path as either a “good" or a “bad" sequence of identifiers.

6.3.3.6 Generation of “good" and “bad" sequences

We report hereafter the formal definition of sequence.

Definition 23. *A **sequence** is defined as $\sigma = \langle k_1, k_2, \ldots, k_n\rangle$, where k_i is the identifier of a tree entry.*

Algorithm 6 shows how a sequence is generated. The algorithm takes as input an abstract symbolic tree AB_ST, and a leaf node m. As initialization steps, it sets the sequence s to empty, and node n equal to m. Afterwards, the algorithm iterates for all the tree entries of n. For each node entry e, it first get the identifier k of e through the set M_{set} (line 6). Then, it extends s by adding k as a prefix (line 7). When s is extended with the identifiers of all tree entries of the current tree node n, then $getSource(AB\_ST.AG\_ST.ST, n)$ is applied to replace n with its source node (line 8). In detail, *getSource* returns the tree node n_s of $AB\_ST.AG\_ST.ST$ such that an arc from n_s to n exists in $AB\_ST.AG\_ST.ST.A$. If n is the root node of $AB\_ST.AG\_ST.ST$, then a source code cannot be found, and the value *nil* is returned as a result. At line 10 the generated sequence s is returned as a result of Algorithm 6. COME applies Algorithm 6 to generate “good-sequence" set and “bad-sequence" set. In particular, the “good-sequence" set is defined by collecting the sequences generated by the algorithm for each leaf node m of $AB\_ST.AG\_ST.ST$ not belonging to $AB\_ST.AG\_ST.ST.N_{va}$ while the “bad-sequence" set is defined by collecting sequences generated by the algorithm for each leaf node m belonging to $AB\_ST.AG\_ST.ST.N_{va}$. Informally, sequences of entries in the original symbolic tree that satisfy the given vulnerability assertion become “bad sequences" while all the other sequences are mapped to “good sequences". Observing Figure 6.3 we can see the sequences of identifiers generated from the abstract tree of Figure 6.2. Column “leaf" uniquely identifies the sequence ending on that node.

6.3.3.7 Assertions generation

In the last step of the proposed methodology, given the good and bad sequences generated in the third step, COME is able to obtain as output a set of assertions identifying the cause of activation of the given vulnerability. By exploiting the alignment

Leaf	Good sequences										
4	A	B	C	D	E	C	E	A	D		
8	A	B	C	D	E	C	F	G	D	L	G
9	A	B	C	D	F	C	L	D	C	E	K

Leaf	Bad sequences									
7	A	B	C	D	E	C	F	G	D	K
5	A	B	C	D	F	C	K			

FIGURE 6.3: Good and bad sequences.

of sequences, we will initially get the set of essential subsequences by filtering bad sequences. Each one of these subsequences underlines a different way of triggering the vulnerability and it is composed of the essential elements needed to trigger such a vulnerability. Once we generated these subsequences, we can easily map back the identifiers to their original meanings (read/write instructions) and obtain a set of temporal assertions. This section is organized as follows: first, we will define additional constraints called “disabled behaviors" necessary to obtain consistent alignments of sequences. Then, we will describe the algorithm responsible for generating the set of shortest bad subsequences capable of triggering the vulnerability. Finally, the generated subsequences are mapped back to read/write operations and a set of temporal assertions is returned as result.

6.3.3.8 Generation of disabled behaviors

In this phase, we employ techniques based on alignments of sequences. We refer to the classical pairwise alignment approach that takes as input a couple of sequences of symbols and gives as output a subsequence of symbols contained in both the input sequences. In particular, we are not concerned with the resulting subsequence but with testing if the shortest input sequence is contained in the other one, or in other words, if it is alignable. We report hereafter the formal definition of “alignable" used in this methodology.

Definition 24. *Given two sequences of identifiers* $\sigma_1 = \langle k_1, k_2 ... k_n \rangle, \sigma_2 = \langle l_1, l_2 ... l_m \rangle$ *with* $n \leq m$, σ_1 *is* ***alignable*** *to* σ_2 *if* $\exists$ *a set of identifiers* $\{l_{i_1}, l_{i_2} ... l_{i_n}\} \subseteq \sigma_2$ *with* $l_{i_1} = k_1$, $l_{i_2} = l_2$, ... , $l_{i_n} = k_n$ *and* $i_1 < i_2 < ... < i_n$.

When dealing with alignments of sequences, we are basically selecting a subsequence of the original sequences. We have to remember that each element of a

Algorithm 6

```
1:  function getSequence(AB_ST, m)
2:      s = ∅
3:      n = m
4:      do
5:          for all e ∈ n do
6:              k = AB_ST.M_set(m, e, n)
7:              s = append(k, s)
8:          n = getSource(AB_ST.AG_ST.ST, n)
9:      while n ≠ nil
10:     return s
```

sequence is an identifier that describes an instruction of the firmware. Each not selected instruction in the alignment is essential in guiding the execution toward the path described by the original sequence. In fact, choices made during the symbolic simulation were embedded inside the instructions during the propagation of constraints, therefore not selecting an element in the alignment is equivalent to ignoring a choice. On this basis, we conclude that a subsequence formed by the selected instructions in the alignment could not describe a feasible succession of instructions. To ensure consistent alignments, sequences have to be equipped with additional constraints that forbid certain types of alignments. We call this type of constraints "disabled behaviors". We report hereafter the formal definition of "disabled behavior"

Definition 25. *Given a good sequence* $gs = \langle k_1, k_2, ...k_n \rangle$ *a couple of symbols* $\langle k_i, k_j \rangle \in gs$, *such that* $i < j$ *and both* k_i *and* k_j *represent a read(write) operation,* $k_i \leftarrow k_j$ *is a* ***disabled behavior*** *if* k_j *represents a read(write) operation on the same address as the operation described by* k_i.

Intuitively, a disabled behavior $k_i \leftarrow k_j$ of a good sequence gs imposes that if $k_i \in gs$ is aligned then also $k_j \in gs$ must be aligned, otherwise the alignment is considered inconsistent.

Fig. 6.4 shows the disabled behaviors extracted from good sequences of Fig. 6.3.

6.3.3.9 Sequences filtering

The idea behind the filtering process consists of finding the shortest bad subsequence not contained in every good sequence. A bad subsequence contained in a good sequence cannot be a minimum sequence that triggers the vulnerability and hence a solution, otherwise also the good sequence that contains it would be a bad sequence. Algorithm 7 shows how the minimum bad subsequences are generated. The algorithm takes as input parameters the set of good sequences GS and the set of bad sequences BS. As initialization step, it assigns the set of filtered sequences FS to empty. After that, the algorithm enters a loop and keeps cycling for all "bad sequences". Then, the execution proceeds in a second loop that keeps cycling until the sequence bs_{min} is completely filtered. Inside this second cycle, we select a subsequence bs_{sub} from the sequence bs_{min} using the function $selectNextSubsequence(bs_{min})$ (line 5). It returns a new subsequence of bs_{min} each time it is called. Starting from sequences of size 1, it keeps returning sequences of longer sizes until a solution is found. Then, we initialize $solutionFound$ to $true$. This variable will serve as a flag to exit the cycle when a solution is found for the current bs_{min}. Afterwards, the algorithm iterates for all good sequences of GS. For each good sequence gs, it checks if bs_{sub} is alignable (Definition 24) to gs by using the function $isAlignable(bs_{sub}, gs)$ which returns a boolean value. If $isAlignable$ returns $true$ for at least a gs then we can conclude that bs_{sumb} is not a solution and therefore we can directly discard it by exiting the for cycle with the command $break$ (line 10). If $isAlignable$ returns $false$ for

Leaf	Disabled behaviors
4	A←A C←C D←D E←E
8	A←G A←M C←C D←D E←F G←G
9	C←C D←D F←E L←K

FIGURE 6.4: Disabled behaviors.

all gs then bs_{sub} is a solution, and hence, a filtered bad sequence. In this case, the flag $solutionFound$ is kept to $true$, and therefore the algorithm exits from the second while cycle (line 11). Then, the found solution bs_{sub} is added to the set FS of filtered sequences (line 12). Finally, the algorithm returns as output the set of minimum filtered sequences FS (line 13).

Algorithm 7

```
1:  function filterSequences(GS,BS)
2:      FS = ∅
3:      for all bs ∈ BS do
4:          do
5:              bs_sub = selectNextSubsequence(bs)
6:              solutionFound = true
7:              for all gs ∈ GS do
8:                  if isAlignable(bs_sub, gs) then
9:                      solutionFound = false
10:                     break
11:         while !solutionFound
12:         FS = FS ∪ bs_sub
13:     return FS
```

For example, we can execute Algorithm 7 with the sequences shown in Fig. 6.3. We start by selecting the shortest bad sequence, which is the sequence identified by leaf 5. We proceed by selecting subsequences from that sequence and aligning them to all the good sequences in Fig. 6.3 identified by the leaves 8, 4, 9. Starting from all bad subsequences of size 1, we observe that such sequences are all contained in at least one good sequence and therefore they can be discarded. If we try to align subsequences of size 2 we observe that the only subsequence not alignable with any good sequence is the sequence $\langle F, K \rangle$, which is the result for this example. Note that the subsequence $\langle F, K \rangle$ is contained in good sequence 9 and therefore should be alignable performing the classical pairwise alignment. Since we are also considering disabled behaviors as additional constraints, this alignment is forbidden by the disabled behavior $F \leftarrow E$ (Fig. 6.4, leaf 9) which does not allow skipping the identifier E in the alignment, making the subsequence $\langle F, K \rangle$ not alignable with any good sequences. We complete the process by mapping back the sequence of identifiers $\langle F, K \rangle$ to the sequence of instructions $\langle Store, 0x4000, X\_i \rangle$ $\langle Store, 0x3000, X\_j \rangle$ with $X\_i \neq 1$ and $X\_j = 1$.

6.3.4 Simulation results

According to the threat model described in Section 6.3.1, we evaluated the effectiveness and efficiency of the methodology in two case studies concerning the validation of a memory protection mechanism and the detection of a logic bomb in a control firmware of a mobile robot. The experimental results have been carried out on a 2.9 GHz Intel Core i7 processor equipped with 16 GB of RAM and running Mac OS.

6.3.4.1 Memory protection mechanism

The analyzed firmware acts as an interface between a memory-mapped IP and an upper-level software. The firmware reads values from the IP interface, then it elaborates a memory address on which it stores the read values. In this scenario, the memory location storing the firmware code is not writable unless the flag BIOSwe

Code 1

```
1: command = 0
2: curr_address = 0
3: curr_data = 0
4: while true do
5:     command = externalInput( )
6:     if command == 1 then
7:         curr_address = externalInput( )
8:     else if command == 2 then
9:         curr_data = externalInput( )
10:        writeToAddress(curr_data, curr_address)
11:    else if command == 12345678 then
12:        InterruptEnabled = 0
13:        BIOSwe = 1
14:    else
15:        No op
16: function afterAnyInstruction( )
17:    if InterruptEnable == 1 then
18:        BIOSwe == 0
```

is set. Moreover, each attempt to change this flag causes an interrupt that resets the BIOSwe at its default value, i.e. zero. In a correct execution flow, each value coming from the IP is then properly stored in a memory location, and, more importantly, any attempt to manipulate the firmware code itself has no effect. However, a security vulnerability can be located in the interrupt controller register. In fact, if the interrupts can be disabled, then BIOSwe is exposed to be set, and the firmware code in memory can be successfully overwritten afterwards. This is a concrete vulnerability, exploited in [96]. We run COME to analyse the binary-code firmware in the above vulnerable context.

In this case, study the defined vulnerability assertion is $BIOSwe = 1$. Code 1 shows a high-level representation of the firmware under analysis. It takes as input a command, a data value and an address. Depending on the given command, the input data value is provided as either a new address or a new word to be written at the current address. If the input command is equal to 1 then an address is received as input and stored in the variable *curr_address* (lines 6-7). If the input command is equal to 2 then a new data value is received as input and stored in the variable *curr_data* (lines 8-9). After that, the received data is written to the previously stored address using the function $writeToAddress(curr\_data, curr\_address)$ (line 10). If the input received is not a feasible command, the firmware ignores it entering in the "no op" branch (line 15). The function $afterAnyInstruction$ implements the memory protection mechanism described above: it is executed after every firmware instruction to keep the BIOSwe register set to 0 (lines 17-18).

This case study covers both the scenarios presented in section 6.3.1.

In the first scenario, the vulnerability is triggered by exploiting the addresses and data received as input. A filtered sequence triggering the vulnerability this way is drafted as follows:

$(a)\langle Load, command, X_a\rangle$ *with* $X_a = 1$
$(b)\langle Store, curr\_address, X_b\rangle$ *with* $X_b = InterruptEnable$
$(c)\langle Load, command, X_c\rangle$ *with* $X_c = 2$
$(d)\langle Store, curr\_data, X_d\rangle$ *with* $X_d = 0$
$(e)\langle Load, command, X_e\rangle$ *with* $X_e = 1$
$(f)\langle Store, curr\_address, X_f\rangle$ *with* $X_f = BIOSwe$

$(g)\langle Load, command, X_g\rangle$ *with* $X_g = 2$
$(h)\langle Store, curr\_data, X_h\rangle$ *with* $X_h = 1$
This filtered sequence summarizes the minimum operations able to trigger the vulnerability without using the shortcut described by the first filtered sequence. The sequence simply generates addresses for both the InterruptEnable (b) and BIOSwe (f) registers and then writes the values 0 and 1 to those addresses, respectively (d and h). Each operation is preceded by a *command* load that decides which branch will be taken in the if-then-else (a, c, e, and g).

In the second scenario, we considered the case where a developer included a malicious branch able to immediately activate the vulnerability by giving as input a certain command value. If the command is equal to 12345678 (line 11) then the firmware directly actives the vulnerability by setting the InterruptEnable register to 0 followed by setting the BIOSwe register to 1 (lines 12-13). Once the sequence triggering the vulnerability this way is filtered, we should end up with a sequence of size 1 containing only the "read" operation on the *command* variable inside the if statement which characterizes the instruction that made the execution fall in the malicious branch. Note that $InterruptEnable = 0$ and $BIOSwe = 1$ are not part of the filtered sequence. Once *command* equals to 12345678, these instructions are forced to be executed (concrete operations) and therefore the instruction $\langle Load, command, X\_i\rangle$ with $X\_i = 12345678$ is enough to characterize the vulnerability.

Table 6.1 reports the execution time of COME by increasing number of simulated firmware instructions. We can notice that the time required by step 1 of the proposed approach, which contains the bottleneck of the whole methodology (symbolic simulation), is of the same order of magnitude as the sum of the times of the succeeding steps.

Table 6.2 reports in this order: the number of simulated instructions, the number of generated symbolic nodes in the labelled symbolic tree, the number of simulated firmware execution paths, the number of simulated firmware execution paths where the vulnerability assertion holds, their average length in terms of number of simulated instructions, the number of generated output assertions and their average length in terms of the number of involved firmware instructions.

Observing Table 6.2 we can see that starting from hundreds of bad sequences (column 4) with dozens of instructions (column 5) we ended up with only the two sequences described above of sizes 1 and 8. The tool is able to find only the first "bad sequence" for tests below 7 millions of simulated instructions (rows 1-3). For tests simulating over 28 million instructions, the tool correctly identifies the two expected sequences (rows 4-5). All other bad sequences describe the same identical ways of activating the vulnerability, they differ only in the number of unnecessary elements they contain.

Finally, Fig. 6.5 shows the memory usage of COME throughout each step of the methodology for each simulation run reported in Table 6.1. Thanks to the efficient C++ implementation of COME and Klee, the whole approach is remarkably contained in its memory consumption; in fact, even when simulating millions of instructions, the memory usage does not exceed 350 MB.

6.3.4.2 Safety control mechanism

The second use case is related to the firmware for a safety control mechanism of a mobile robot. The considered robot is equipped with sensors used to keep track of

TABLE 6.1: Execution time (memory protection mechanism).

Simulated instructions	Step 1 time	Step 2 time	Step 3 time	Step 4 time	Total time
411'940	2618 ms	5902 ms	78 ms	88 ms	8686 ms
1'632'838	9511 ms	23654 ms	900 ms	90 ms	34155 ms
7'241'517	46501 ms	2.7 min	2091 ms	91 ms	3.46 min
28'313'022	3.11 min	7.28 min	3091 ms	97 ms	10.5 min
104'998'737	16.5 min	52.1 min	8012 ms	99 ms	69 min

TABLE 6.2: Simulation and filtering details (memory protection mechanism).

Simulated instructions	Symb. states	Tot seq.	Bad seq.	AVG bad seq. length	Generated assertions	AVG assertion length
411'940	84	67	4	34	1	1
1'632'838	548	293	8	40.5	1	1
7'241'517	2578	1345	12	47	1	1
28'313'022	9378	4853	72	59.7	2	4.5
104'998'737	38311	20331	628	74.8	2	4.5

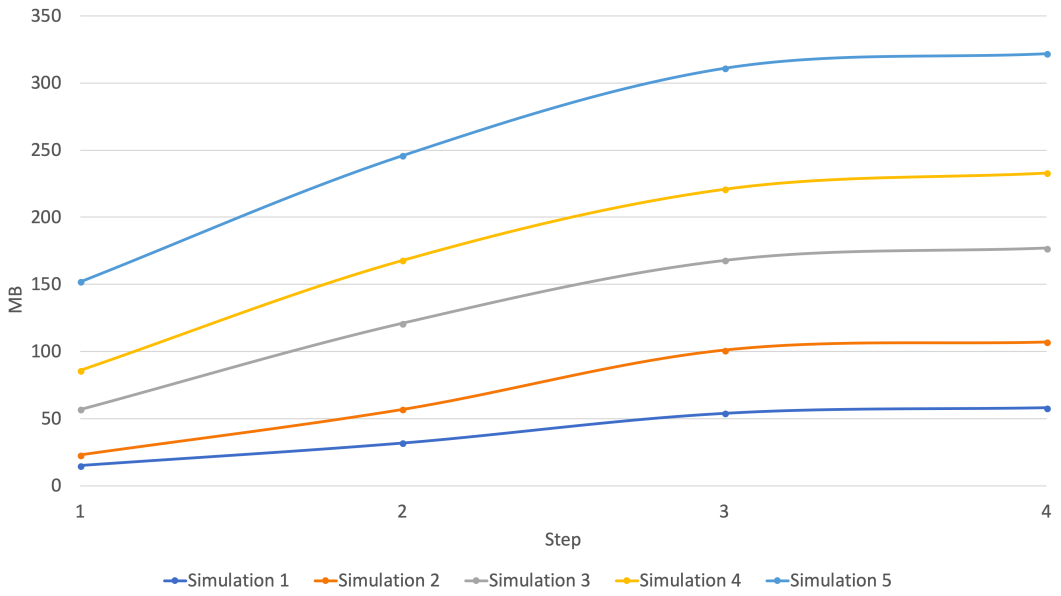

FIGURE 6.5: Memory usage for (memory protection mechanism).

nearby objects. Given a target position on the plane, the robot is capable of reaching such a location automatically. To implement this functionality, the underlying software exploits two main routines: a "local planner" and a "path planner". The path planner is used to generate a feasible path plan that the robot should follow to reach the target position. The local planner takes care of local trajectory adjustments required to avoid unexpected obstacles that might occur on the path. Moreover, the control firmware includes a safety control mechanism that halts the robot if it gets too close to an obstacle. We considered the scenario where a malicious developer included a logic bomb capable of disabling the safety control mechanism. A logic bomb is a piece of code intentionally inserted into a software that activates malicious functionalities when a certain condition is satisfied. In this case study, the vulnerability consists of the sequence of instructions capable of firing the logic

bomb.

Code 2

```
1: off = 4
2: while true do
3:     while !newTargetAvailable do
4:         NoOp
5:     off >> (target_x == target_y)
6:     while cur_x ≠ target_x || cur_y ≠ target_y do
7:         if dis(cur_x, obs_x, cur_y, obs_y) < 10 && off then
8:             haltRobot()
```

Code 2 shows a high-level representation of the firmware under analysis. The firmware receives its input from a set of memory-mapped ports. We assume that the values provided by these ports can potentially change after every instruction of the firmware. To simulate such a behavior, we instruct the symbolic simulation engine to inject a new symbolic value in all input ports after executing an instruction. This approach effectively reproduces the environment of the real firmware executed on the robot. The port *newTargetAvailable* becomes true when the robot receives a new target position. The firmware keeps waiting until a new target is provided (lines 3-4). The ports *target_x* and *target_y* provide the coordinates of the target position after *newTargetAvailable* becomes true. After that, the firmware enters a loop that keeps cycling until the robot reaches the target position, or in other words, until the condition $cur\_x \neq target\_x \,||\, cur\_y \neq target\_y$ becomes false (line 6). The ports *cur_x* and *cur_y* provide the current position of the robot. The safety mechanism is implemented by checking that the distance from the current position remains higher than an arbitrary threshold 10 (line 7). The distance is calculated with the function $dis()$, while the ports obs_x and obs_y provide the position of the detected obstacle. If the robot gets too close to the obstacle, the firmware calls the function $haltRobot()$ (line 8) which stops the motors of the robot as fast as possible. However, the malicious developer added a variable off to implement a vulnerability in the firmware. Each time the firmware receives a new target position (*newTargetAvailable* is true), if the given coordinates are equal ($target\_x == target\_y$), the variable off is shifted to the right by 1 (line 5). Since the variable off was initialized to 4, after three shifts the variable becomes equal to 0. Therefore, the condition $dis(cur\_x, obs\_x, cur\_y, obs\_y) < 10$ && off becomes stuck to $false$, effectively disabling the safety mechanism. To trigger this vulnerability, the attacker needs to provide as input the correct coordinates ($target\_x == target\_y$) three times. Table 6.3, Table 6.4 and Fig. 6.6 show the simulation results for the described case study. The analysis of those results follows the same line of reason applied to the previous case study.

TABLE 6.3: Execution time (safety control mechanism).

Simulated instructions	Step 1 time	Step 2 time	Step 3 time	Step 4 time	Total time
9'018'410	2.3 min	7 min	5091 ms	81 ms	9.4 min
12'930'221	2.9 min	10.2 min	5201 ms	88 ms	13.3 min
32'229'539	8.3 min	17.7 min	8981 ms	3803 ms	26.1 min
81'718'093	19.1 min	39.1 min	7202 ms	9101 ms	58.2 min

In this case study, COME was able to identify several bad sequences of instructions capable of triggering the vulnerability. However, after the filtering step, only

TABLE 6.4: Simulation and filtering details (safety control mechanism).

Simulated instructions	Symb. states	Tot seq.	Bad seq.	AVG bad seq. length	Generated assertions	AVG assertion length
9'018'410	1'623	1420	1	127.3	1	6
12'930'221	1'841	1589	8	129.2	1	6
32'229'539	3'409	1709	28	132.8	1	6
81'718'093	14'799	3812	251	141.9	2	8.5

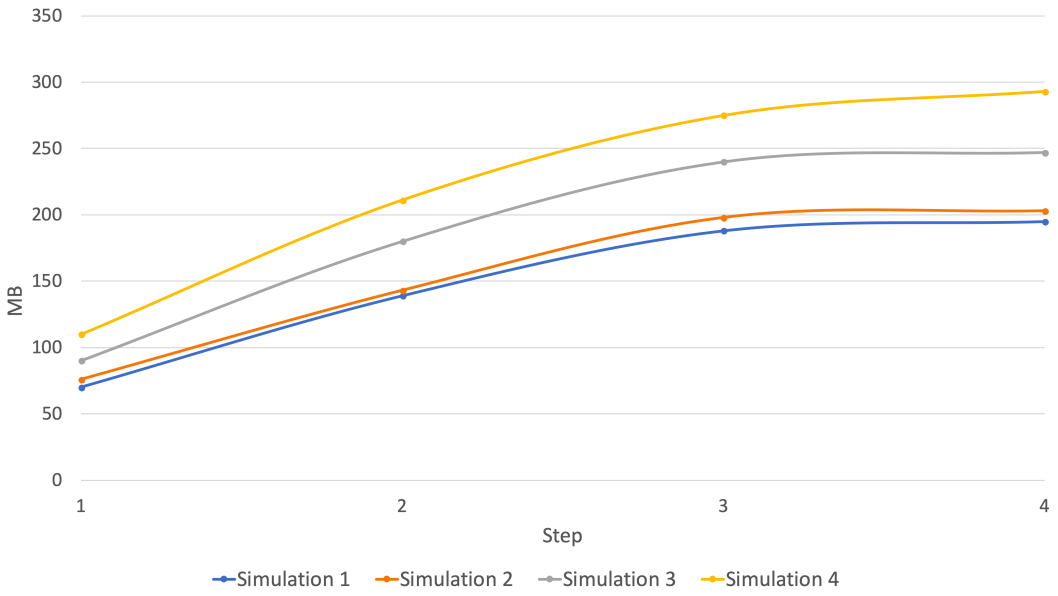

FIGURE 6.6: Memory usage (safety control mechanism).

two scenarios were identified. In the first scenario, the attacker simply gave as input three times in a row the coordinates (x, y) with $x == y$, correctly identifying the vulnerability. In the second scenario, the coordinates (x, y) also satisfied the condition $cur\_x == target\_x$ && $cur\_y == target\_y$, making the execution triggers the vulnerability with fewer instructions as the variable off was shifted without making the firmware entering in the third while loop (line 7).

We can conclude that COME is really effective in generating a low number of temporal assertions that define the causes of a vulnerability. This provides a great advantage for verification engineers that can definitely focus on a few assertions to decide the way of patching the firmware, thus solving the vulnerability. On the contrary, without COME, the verification engineer should analyze a huge number of bad execution sequences, which would be unfeasible even for a simple firmware.

6.4 BECAUSE

In this section, we present the second tool called BECAUSE. The tool works on any system-level implementation that can be compiled into LLVM bitcode. Given an unexpected behaviour formalised by means of a propositional assertion, the tool provides the user with a reduced execution trace that still triggers such behaviour, thus highlighting the essential instructions related to it.

The underpinning methodology applies a sequence of reductions to the execution trace through a program-slicing-based technique. After each reduction, we verify by simulation if the remaining trace is still an executable program capable of triggering the unexpected behaviour. This procedure works in two phases. Firstly, we remove all the instructions not belonging to the cone of influence of the unexpected behaviour by exploring the dynamic program dependency graph (DPDG). Secondly, we apply a heuristic based on an instruction-clustering procedure to further reduce the remaining trace. Furthermore, we extend the reported methodology to perform bug explanation of unexpected behaviours modelled as temporal assertions.

The rest of the section is organised as follows. In section 6.4.1, we provide a few preliminary definitions necessary to clearly understand the proposed approach. In section 6.4.2, we overview the methodology, and then we describe in detail each step. In section 6.4.3, we describe how to extend the methodology to perform bug explanation with temporal assertions. In section 6.4.4, we report the experimental results.

6.4.1 Preliminaries

Definition 26. *An **instruction** is a programming statement following the LLVM bitcode syntax [97].*

Definition 27. *An **execution trace** is a sequence of instructions representing an executable instance of a program.*

Definition 28. *Let i_1 and i_2 be two instructions, i_2 is **data dependent** on i_1 if i_2 accesses a portion of memory allocated or modified by i_1.*

Definition 29. *Let i_1 and i_2 two instructions, where i_1 is a branch with multiple branch targets, if changing the branch target of i_1 may cause i_2 to not be executed, then i_2 is **control dependent** on i_1.*

Definition 30. *A **dynamic program dependence graph** is a structure composed of nodes and edges where each node represents an instruction of an execution trace and each edge represents a data or control dependency between instructions. Let n_1,n_2 be two nodes of a DPDG, if n_2 has an incoming edge e_1 connecting n_2 with n_1, then the instruction represented by n_2 is either data-dependent or control dependent on the instruction represented by n_1.*

Fig. 6.8 shows an example of a DPDG where red edges are data dependencies and blue edges are control dependencies.

Definition 31. *Let as be an assertion and $A = \{a_0, a_1, ..., a_n\}$ the set of memory addresses of variables $v_0, v_1, ..., v_n$ on which as predicates, then the memory address a^f is a **fundamental address** of as if $a^f \in A$.*

6.4.2 Methodology

As shown in Figure 6.7, the proposed tool is composed of 3 main steps executed sequentially. The inputs of the tool are the LLVM code of the DUV and a set of propositional assertions capturing unexpected behaviours. Additionally, the user can provide the sequences of inputs that eventually falsify the assertions, thus highlighting the presence of a bug. For each failed assertion, the tool produces a sequence of minimal instructions explaining the cause of the failure, i.e., the reason for the bug. Hereafter, we provide an overview of the 3 main steps.

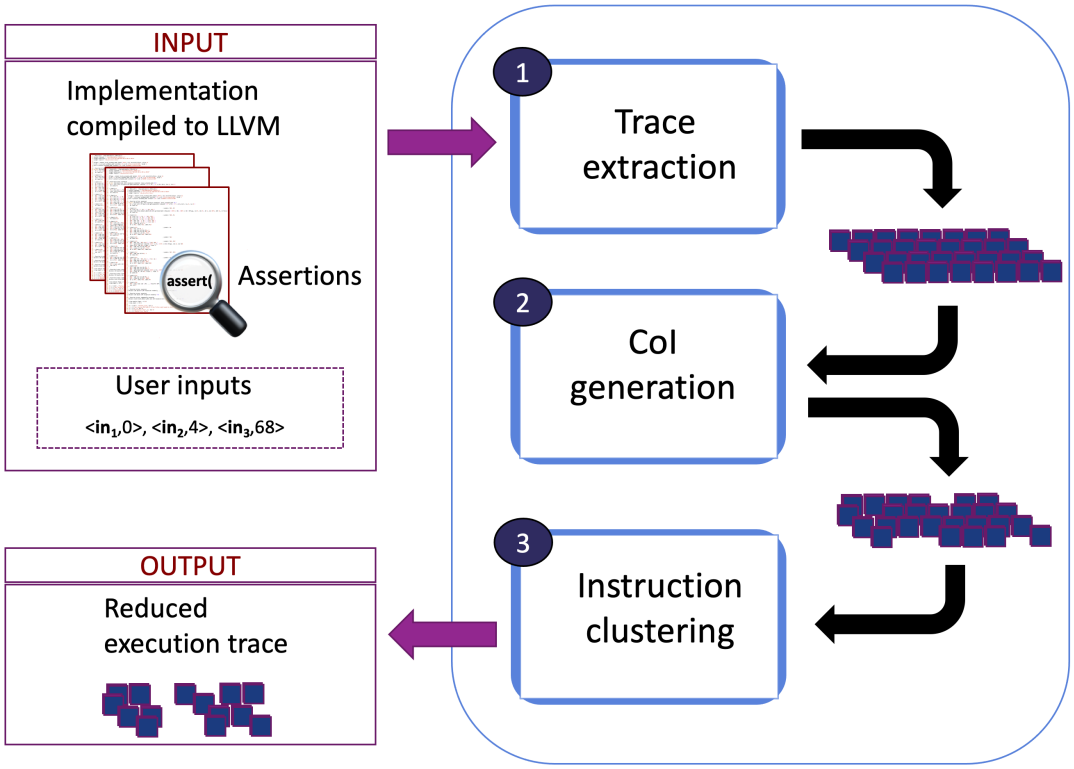

FIGURE 6.7: Methodology execution flow

1. **Trace extraction:** given the failure of an assertion, in the first step of the methodology, we extract the sequences of LLVM instructions that cause the execution to activate the unexpected behaviours. This procedure may occur in two ways, depending on whether the user provided the sequences of inputs or only the assertion. In the first case, the sequence of instructions firing the unexpected behaviour is extracted by executing the implementation with the given inputs until the related assertion fails. In the latter case, we use symbolic simulation to find a sequence of instructions capable of falsifying the assertion.

2. **Cone of influence generation:** each trace extracted in the previous step is reduced by applying a dynamic program slicing algorithm to eliminate all instructions not belonging to the cone of influence (CoI) of the assertion. For each trace, we generate a DPDG characterising control and data dependencies between instructions. After that, we apply a reachability algorithm to determine what instructions influence the value of the variables contained in the assertions. The instructions not selected by the above procedure are removed from the trace.

3. **Instruction clustering:** in the last step of the methodology, we apply a clustering procedure to further reduce the remaining instructions. Our approach consists of dividing the instructions into independent clusters such that applying any reduction procedures to one cluster would not prevent a satisfying minimisation in another. Once such clusters are identified, we apply a combinatorial-based reduction to obtain the minimal sequence in each cluster.

To simplify the exposition, we apply the proposed methodology to the example shown in listing 6.1. It consists of a design written in C implementing a simple arithmetic transformation. The code is decorated with an immediate assertion specifying

```
1 int in;
2 int main() {
3     int a=0;
4     int b=5;
5     while (1) {
6         in= getNextInput();
7         if (in == 0) {
8             a=4;
9             a++;
10        } else if (in < 5) {
11            a+=10;
12            a--;
13        } else if (in > 90) {
14            a-=2;
15            b+=3;
16            assert(a != 12);
17        }
18    }
19 }
```

LISTING 6.1: Running example

a property that must hold during execution (line 16).

6.4.2.1 Trace Extraction

In the first step of the methodology, we extract the sequences of LLVM instructions that expose the unexpected behaviour, namely, sequences starting with the first instruction of the program and ending with the assertion failure.

In Table 6.5 we report an execution trace falsifying the assertion contained in the running example. The instructions are labelled with two identifiers: the first uniquely identifies each LLVM instruction, and the second links each instruction to its corresponding high-level statement in listing 6.1. To extract such an execution trace, we symbolically simulate the DUV, until we find an execution path that falsifies the target assertion. To accomplish that, we exploit the symbolic simulation engine provided by KLEE [98]. To simulate the DUV with KLEE, the DUV inputs are marked as "symbolic" to declare where symbolic values should be injected. For example, to symbolically simulate the running example, line 6 must be replaced by $klee\_make\_symbolic(in)$, since variable *in* is the only input. Then the symbolic simulation explores the various paths of the running example until it finds a path that makes the assertion at line 16 fail. Such a path has the following symbolic constraints: $(in1 == 0, in2 < 5, in3 > 90)$, where the subscript i of ini refers to the value of the variable *in* at the symbolic iteration i.

Symbolic simulation is quite expensive in terms of computational resources. As a matter of fact, it is an exponential-time algorithm; however, if the user already has the required sequence of inputs to activate the bug, it can be run in a linear-time constrained mode, since only one path needs to be explored. In the running example, we assumed the following sequence of inputs: $\{\langle in1, 0\rangle, \langle in2, 4\rangle, \langle in3, 125\rangle\}$. Therefore, the symbolic simulation must explore only one path with the following constraints: $in1 == 0$, $in2 == 4$, $in3 == 125$ producing the sequence of instructions reported in Table 6.5.

6.4.2.2 Cone of influence generation

In the second step of the methodology, the execution trace extracted in the previous phase is reduced by applying a dynamic program slicing algorithm. The remaining elements of the execution trace correspond to instructions involved directly (or indirectly through association) in data or control dependencies with the variables contained in the failed assertion, that is, the cone of influence of the assertion. The procedure works in three main sub-steps.

In the first step, we generate the DPDG of the execution trace extracted in the first step of the methodology. In the last decades, many algorithms have been proposed to generate DPDGs efficiently, one of which can be found in [99]; therefore, we do not describe such an algorithm in this paper. Figure 6.8 shows the DPDG for the execution trace listed in Table 6.5.

In the second step, we identify all *store* instructions in the execution trace accessing fundamental addresses for the target assertion. These are the only instructions that can modify the variables on which the assertion predicates, and therefore, that can change its truth value. We call $fundInst$ the set of instructions collected with the above procedure. Since the algorithm to identify $fundInst$ is trivial, we do not give any further details on it. In the running example, there is only one fundamental address, namely, the memory address of variable a in assertion $a! = 12$. Such an address is allocated by instruction 3 of Table 6.5 and saved in the LLVM label %2. In this example, $fundInst$ is composed of the store instructions $\{5, 13, 16, 28, 31, 46\}$, which are accessing the address in label %2.

In the last step, we traverse the generated DPDG starting from each store instruction in $fundInst$ and going backward through the incoming edges until a node with no incoming edges is found. By construction, the generated DPDG is an acyclic direct graph, therefore the whole procedure has a worst-case time-complexity of $O(V)$, where V is the number of nodes in the DPDG. Each instruction represented by a node in the DPDG that is not visited in the aforementioned procedure will be removed from the execution trace. The whole procedure is formalised in function $extractCoI$ of algorithm 10.

Algorithm 10 Cone of influence extraction

```
1:  function extractCoI(fundInst, trace, dpdg)
2:      visited = ∅
3:      coi_Trace = ∅
4:      for all fi_id in fundInst do
5:          node = dpdg.getNodeFromId(fi_id)
6:          backwardDFS(node, visited)
7:      for id = 0, id < trace.size(), id++ do
8:          if !visited.contains(id) then
9:              reducedTrace.pushBack(trace[id])
10:     return reducedTrace
11:
12: function backwardDFS(node, &visited)
13:     visited.insert(node)
14:     for all edge in node.getInEdges() do
15:         sourceNode = edge.getSource()
16:         if !visited.contains(sourceNode.getId()) then
17:             backwardDFS(sourceNode, visited)
```

The inputs of this function are the identifiers corresponding to fundamental instructions $fundInst$, the execution trace $trace$ and the DPDG $dpdg$. First, $visited$ and

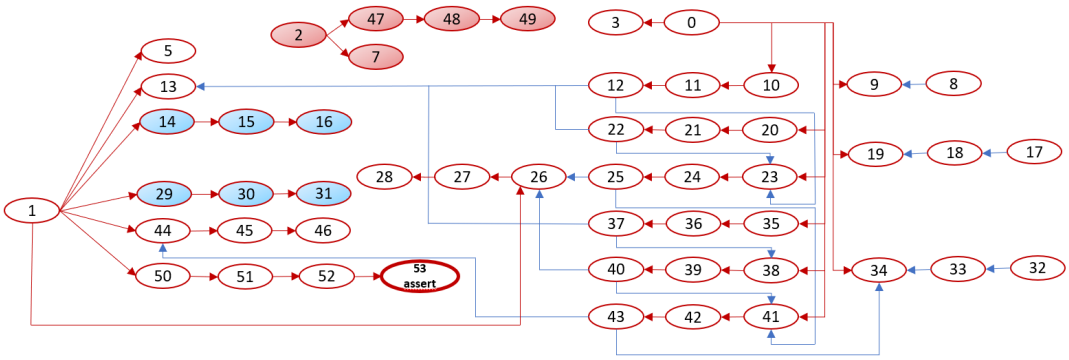

FIGURE 6.8: DPDG of the running example

reducedTrace are declared and initialised (lines 2,3); the first variable contains the visited nodes, while the latter contains the reduced execution trace. After that, we apply the function *backwardDFS* to all the nodes representing the fundamental instructions in $fundInst$ (lines 4-6). Each node is retrieved from the DPDG through the method *getNodeFromeId* (line 5) which returns a node data structure for a given instruction identifier. The function *backwardDFS* performs a depth-first search algorithm going backward from the incoming edges of each node. First, the function marks the current node as visited (line 13). After that, it iterates through all the incoming edges of the current node (line 14). Then, it retrieves the source node *sourceNode* connected to *node* through *edge* using the method *getSource* (line 15). If *sourceNode* is not already marked (line 16), then we apply *backwardDFS* recursively using *sourceNode* as input (line 17). When all the visits are concluded, we iterate on all the instructions in *trace* (line 7) and we add to *coi_Trace* the instructions that do not have a corresponding marked node (lines 8-9), that is, that do not have a corresponding node stored in *visited*. Finally, the reduced trace is returned (line 10).

If we apply the above procedure to the running example, the instructions corresponding to nodes 2, 7, 47, 48, and 49 are removed from the trace. These nodes are highlighted in red in Figure 6.8. Intuitively, these instructions refer to the declaration and utilisation of variable b, which does not have any control or data dependency with variable a in the assertion. From now on, we will use the term *CoI-Trace* to refer to the execution trace reduced with the above procedure.

6.4.2.3 Instruction clustering

In the last step of the methodology, we apply a heuristic procedure to further reduce the remaining instructions in the CoI-Trace. Further reductions are necessary because in most cases, step two of our methodology can not produce a minimal sequence of instructions falsifying an assertion. Consider, for example, the high-level instructions $a{+}{+}$ and $a{-}{-}$ contained, respectively, at lines 9 and 12 of the running example. Since the assertion predicates on variable a, which is data-dependent on these instructions, the previous step is not capable of removing them. In theory, any subsequence of instructions of the execution trace could be a minimum sequence of instructions explaining the unexpected behaviour. Therefore, any algorithm seeking to find the minimal sequence would suffer from exponential complexity, and hence, scalability issues. To tackle this problem, our approach splits the instructions of the CoI-Trace into independent clusters such that applying any reduction to one cluster would not prevent a satisfying reduction to another cluster. Since every cluster contains a small number of instructions, it is feasible to quickly find the

optimal reduction for each cluster. We generate such clusters by grouping store instructions accessing the same memory address. Note that this is just one method of clustering the instructions, the whole methodology can be still applied with different heuristics. Our clustering heuristic does not produce clusters completely data/control independent from one another; nonetheless, they provide a satisfying amount of independence to apply effective individual reductions. Since each store instruction can only access one memory address, the required clustering procedure is straightforward. In the running example, the clustering procedure produces two clusters for the execution trace of Table 6.5: $c_1 = \{13, 16, 28, 31, 46\}$ for the store instructions accessing the address of variable a, and $c_2 = \{9, 19, 34\}$ for the address of variable in.

Algorithm 11 Reduction through clustering and slicing

```
1:  function reduce(trace,dpdg)
2:      finalTrace = trace
3:      C = generateClusters(trace)
4:      for all c_i in C do
5:          for s = c_i.size(), s > 0, s-- do
6:              combs = getCombs(c_i.size(), s)
7:              for all comb_i in combs do
8:                  c_sel = select(c_i, comb_i)
9:                  trace^s = strip(c_sel, finalTrace)
10:                 if test(trace^s) then
11:                     removeLooseInst(c_sel, dpdg, trace^s)
12:                     finalTrace = trace^s
13:                     goto newCluster
14:         label newCluster
15:     return finalTrace
16:
17: function removeLooseInst(c_sel,dpdg,&trace^s)
18:     for all c_j in c_sel do
19:         visited = ∅
20:         node = dpdg.getNodeFromId(c_j)
21:         removeLooseNodes(node, visited)
22:         trace^s.erase(visited)
23:
24: function findLooseNodes(node,&visited)
25:     if node.getInEdges().size() > 1 then
26:         return
27:     visited.insert(node)
28:     for all edge in node.getInEdges() do
29:         sourceNode = edge.getSource()
30:         findLooseNodes(sourceNode, visited)
```

Let $a_i, a_2, ..., a_k$ be the addresses accessed in the store instructions of the CoI-Trace, $C = \{c_1, c_2, ..., c_k\}$ is the set of clusters generated with the above procedure, where c_i contains the store instructions for address a_i. We define the **optimal reduction** as the biggest set of instructions $optRed_i = \{i_1, i_2..., i_m\}$ in a cluster c_i such that if the execution trace is stripped of the instructions contained in $optRed_i$, the trace is still an executable program capable of falsifying the assertion. For each cluster, we find its optimal reduction and we remove the respective instructions from the execution trace. In the running example, instructions 16 and 31 correspond to the optimal reduction of cluster c_1. We identify a candidate optimal reduction $optRed_i$ of a cluster c_i by applying a "select and test" procedure. Firstly, we select a subset

$s_i \subseteq c_i$, then we remove the selected instructions from the trace. Secondly, we test if the execution trace is still an executable program capable of falsifying the assertion. To perform such a test, we exploit the KLEE LLVM interpreter to re-execute the reduced trace.This procedure can produce only three outcomes: (1) the assertion fails during execution; (2) the assertion does not fail; (3) a branch instruction jumps to a different target than the one in the original trace.

In the first scenario, removing the instructions does not affect the truth value of the assertion, hence, the removed instructions are considered a candidate optimal reduction. On the contrary, in the second and third scenarios, the removed instructions were necessary to, respectively, falsify or reach the assertion, therefore, they can not be removed from the trace. The biggest candidate optimal reduction identified with the above procedure is the optimal reduction for the given cluster. Step three of our methodology is completely formalised in the function *reduce* of Algorithm 11. First, the function generates the clusters of stores instructions (line 3) through the method *generateClusters*. Then, the selection and test procedure is performed for all clusters. The selection phase works by selecting progressively smaller combinations of cluster instructions (lines 4-8). For example, let $c_p = \{23, 45, 98\}$ be a cluster of instructions, the selection phase starts by selecting combinations of size 3, which is only $\langle 23, 45, 98 \rangle$. After that, it continues with combinations of size two, which are $\langle 23, 45 \rangle$, $\langle 23, 98 \rangle$, $\langle 45, 98 \rangle$ and finishes with combinations of size 1, which are $\langle 23 \rangle$, $\langle 45 \rangle$, $\langle 98 \rangle$. For each combination, a new reduced trace $trace^s$ is generated by removing the corresponding instructions using function $strip$ (line 9). $trace^s$ is re-executed through function *test* (line 10). If *test* returns true, then we are in scenario 1 of the aforementioned procedure and c_{sel} is an optimal reduction of c_i. In this case, the newly reduced trace is saved in $finalTrace$ (line 12) and the execution moves to the next cluster (line 13). Finally, when the trace is reduced using all clusters, we return the final trace (line 15). If we apply this procedure to cluster c_1 and c_2 of the running example, we discover that there is no candidate reduction for c_2 as all its store instructions are necessary to explain the unexpected behaviour; on the contrary, cluster c_1 admits an optimal reduction consisting of instructions 16 and 31.

In most cases, removing a store instruction i_s generates a chain of "loose instructions" i_1,i_2,...,i_{p-1},i_p where i_s is data dependent only to i_1, i_1 is data dependent only to i_2 ..., i_p is data dependent only on i_{p-1}. Since i_1 is the only data dependence of i_s, removing i_s causes i_1 to become independent from all the other instructions in the trace. Therefore, since i_1 is no longer part of the cone-of-the influence, we can safely remove it from the trace. In the same way, i_2...i_{p-1},i_p are removed in a chain-reaction fashion once their only dependence is removed. The above procedure is implemented by the function *removeLooseInst* of Algorithm 11. The inputs of *removeLooseInst* are the store instructions c_{sel} removed in the previous iteration of *reduce*, the DPDG $dpdg$ and the stripped trace $trace^s$. The procedure works in two phases executed for every instruction in c_{sel} (line 18). First, it finds the nodes *visited* corresponding to loose instructions in $dpdg$ using function $findLooseNodes$ (lines 19-21). Second, the found instructions are removed from $trace^s$ (line 22). Function $findLooseNodes$ performs the same task as $backwardDFS$, except that it returns when a node with more than one dependence is found. By removing instructions 16 and 31 in the running example, we generate the loose instructions 14, 15 and 29, 30, respectively. These instructions are removed automatically through the *removeLooseInst* function.

6.4.3 Bug explanation with temporal assertions

In the context of temporised DUVs, functional requirements involve the concept of time, where behaviours are allowed to span across multiple time units. These behaviours are usually verified using assertions formalised through temporal logic such as LTL. Due to its complex nature, understanding and fixing a bug involving temporal logic is way more demanding than finding the cause of an error observable through the failure of a simple propositional assertion.

In this section, we describe how to extend the methodology in section 6.4.2 to perform bug explanation where the unexpected behaviour is identified through a failing temporal assertion. First, we describe how to handle the advancement of time (sec. 6.4.3.1). After that, we report how to extract an execution trace that makes a temporal assertion fail (sec. 6.4.3.2). Finally, we show how to modify the extracted trace in order to apply the techniques explained in the second and third steps of the methodology (sec. 6.4.3.3).

6.4.3.1 Time flow

Temporal assertions are an invaluable tool to verify synchronous RTL designs where the advancement of time is usually defined through a clock signal. Each time a clock signal reaches a positive (or negative) edge, time advances by 1 unit inside the assertion. However, in the specific domain of application of this work, there is no signal that is responsible for articulating the advancement of time. To solve this issue, in this work time advances by one time unit each whenever a new input is provided to the design. The values of the variables inside an assertion at time t_i (corresponding to the i-th input) are equal to the values of the corresponding variables inside the design before executing the instructions necessary to read $input_{i+1}$. In the running example, the value of variable a is equal to 0 at time t_0, before reading the first input. a becomes equal to 5 after receiving the first input $\langle input_1, 0 \rangle$ at time t_1. Note that inside the assertion, the first evaluation unit is t_1 (first sample of the variables) and not t_0.

If the execution reads multiple consecutive inputs, they are all considered part of the same time unit. For example, if the execution is currently at time t_j and the simulation must execute the following instructions

```
in1 = getNextInput1();
in2 = getNextInput2();
in3 = getNextInput3();
```

then, time is equal to t_{j+1} after executing the third statement. This is necessary to allow the evaluation of multiple inputs on a single time unit.

In this work, we consider only safety assertions following the template $always(antecedent \rightarrow consequence)$ where both the antecedent and the consequent can be any LTL temporal formula.

6.4.3.2 Trace extraction

Evaluating temporal assertions while performing symbolic simulation presents several additional issues, we describe the main challenges below.

- The assertion is no longer part of the source code of the design; therefore, it must be handled by the simulator outside the simulation.

- The variables used inside an assertion might not be always available during simulation; this happens because the existence in memory of a variable depends on the scope in which it is declared.
- The symbolic simulation explores several computational paths; therefore, the simulator must keep track of the state of the temporal assertion for every path.

To solve the above issues, we have developed the procedure described in Fig. 6.9.

Before starting the simulation, the LTL assertion is translated to a checker in the form of a deterministic finite-state automaton. The automaton always contains a root node as the initial state of the checker and a rejecting node where the assertion fails. The state of a checker is completely identified with an unsigned integer. Each edge is labelled with a propositional formula. Given a checker ch in state s_i and a proposition p_k on the outer edge connecting s_i with s_j; if p_k is true for the current sample, then s_j is the next state of the checker. A sample is a set of couples $S_i = \{(var_1, val_1)_i, ..., (var_n, val_n)_i\}$ where each element $(var_j, val_j)_i$ corresponds to value val_j at time i of variable var_j; $var_1, ..., var_n$ are the variables contained in the LTL assertion. To determine value val_j, the simulator must know the scope in which to find the corresponding variable var_j; therefore, the user has to add such information in the assertion by appending the scope to the variable. In the assertion of Fig. 6.9, variable a is used as $main :: a$ since it is declared in the main function; likewise, variable in is used without any additional information to specify that it is declared in the global scope. If the simulator tries to make a sample of variable var_k that does not exist in memory at time i, then the sample will contain a val_k equal to 0.

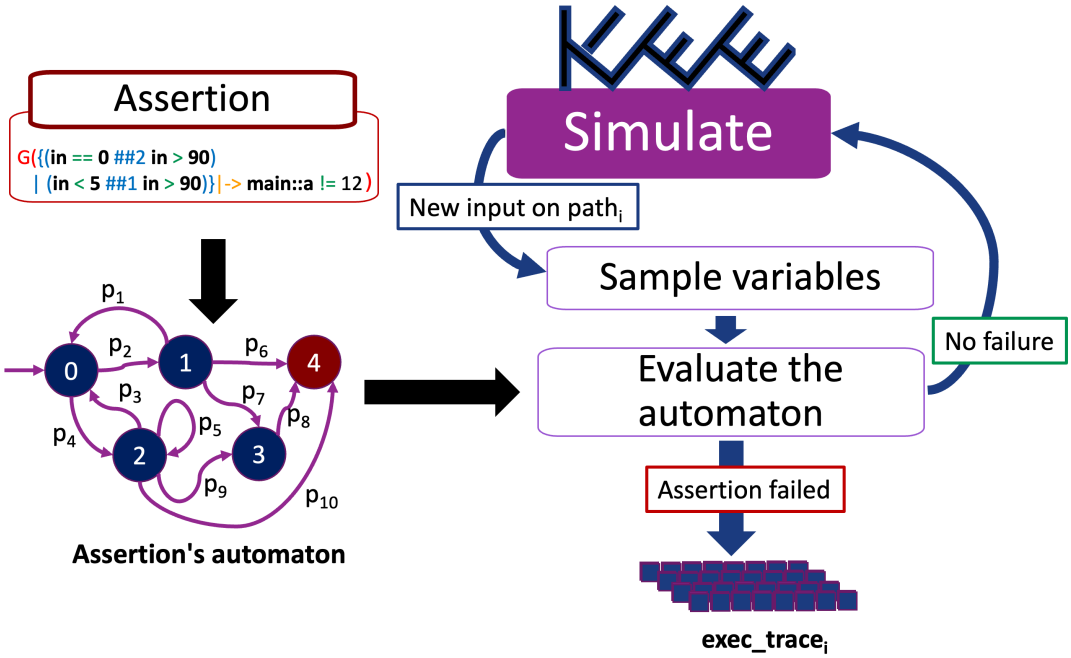

FIGURE 6.9: Trace extraction with temporal assertion

Function *evalAutomaton* of Algorithm 12 formalises how to perform an evaluation for an automaton *aut* and a sample *samp*. The function searches for an outer edge *outEdge* labelled with a proposition that is true for sample *samp* (lines 2-3). After that, the state of the automaton is updated (line 4). If the next state is rejecting (line 5), then the function returns false to notify that the assertion failed (line 6). If the next state is not rejecting, then the function returns true as the assertion did not fail on the current time unit (line 8).

Algorithm 12 Automaton's evaluation

```
1: function evalAutomaton(aut, samp)
2:     for all outEdge in aut.currState.outEdges do
3:         if outEdge.prop.evaluate(samp) then
4:             aut.state = edge.toState
5:             if outEdge.toState.type == Rejecting then
6:                 return false
7:             break
8:     return true
```

Once the checker and all the utilities to evaluate it on a trace are prepared, we perform symbolic simulation to identify a computational path on which the assertion fails. To do that, we have extended the KLEE framework [98]. In particular, each time a new input must be read in the execution (new symbolic value), the simulator creates a sample of the variables and evaluates the checker on the current time unit. Note that each computational path (called ExecutionState in KLEE) contains a unique instance of the checker stored as an unsigned integer (we only need to keep track of its current state). If the evaluation of $checker_i$ on $path_i$ returns false, then the assertion failed and a faulty execution trace $exec\_trace_i$ is found; otherwise, the simulation continues. As in section 6.4.2.1, if the user provided the inputs necessary to make the assertion fail, then only one path is explored by the symbolic simulation.

6.4.3.3 Trace decoration

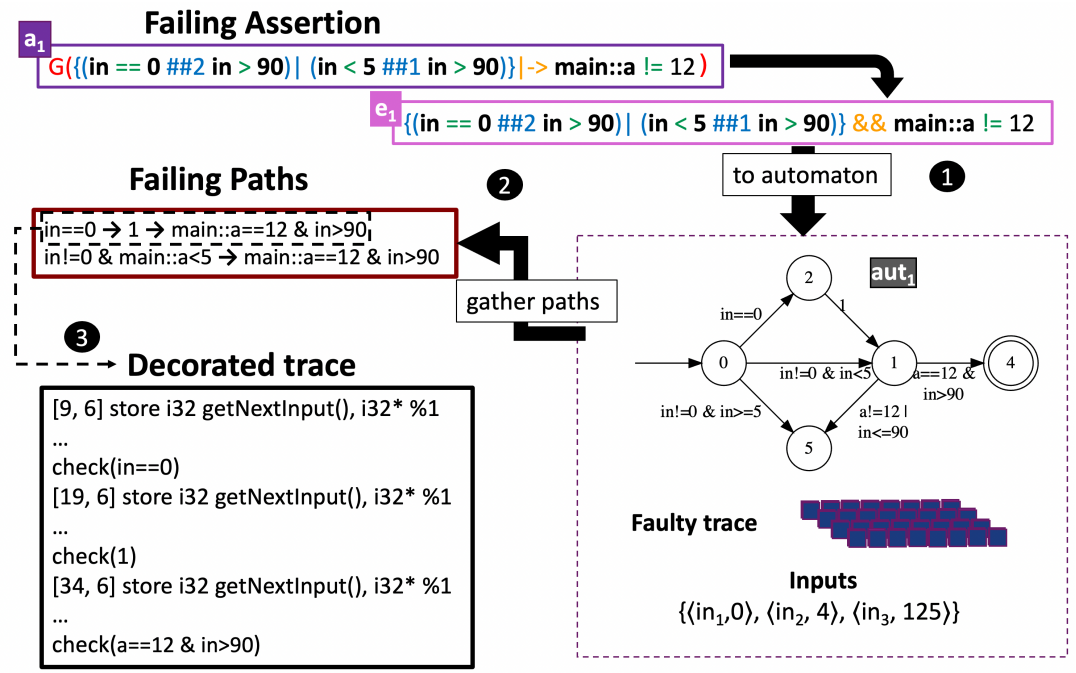

FIGURE 6.10: Trace decoration of the running example

In this section, we describe how to modify an extracted execution trace to include information on the failure of a temporal assertion. The result of this procedure is a set of decorated execution traces on which to apply steps 2 and 3 of the methodology described in section 6.4.2. To simplify the exposition, we will refer to the example in

fig. 6.10. The example involves the same implementation reported in listing 6.1 that generates the same execution trace reported in table 6.5 on which assertion a_1 fails.

The methodology is based on the assumption that the failure of a temporal assertion can be described as a sequence of propositions $\langle p_1, ..., p_n \rangle$ that are true on a sequence of time units $\langle 1, ..., n \rangle$, where p_i is true at time i. For example, assertion a_1 of Fig. 6.10 fails if the sequence of propositions $\langle in! = 0 \,\&\, a < 5, a == 12 \,\&\, in > 90 \rangle$ is true on two consecutive time units. This sequence of propositions corresponds to an accepting path of the automaton generated from the expression *ant* & !*con*, where *ant* and *con* are the antecedent and the consequent of the original assertion. The simulator deduces that the assertion fails on the execution trace by checking that all the propositions in the sequence are true on the corresponding time units.

The whole procedure consists of three main steps. First, the original assertion $G(antecedent \rightarrow consequent)$ is converted to the expression *antecedent* & !*consequent* and translated to an automaton. Note that this automaton contains both accepting and rejecting states. Fig. 6.10 contains the conversion of assertion a_1 to expression e_1 and its translation to automaton aut_1.

In the second step, the procedure retrieves the paths of the automaton justifying the failure of the assertion on the execution trace. This process is formalised in function *retrievePaths* of algorithm 13. The idea of the algorithm is to evaluate the edges of the automaton using the samples of the execution trace to build the sequences of propositions that make the assertion fail. The inputs of function *retrievePaths* are the automaton *aut* and the list of samples *samps*. Variables *paths* contains the list of retrieved paths and *currPath* is a utility variable used to build the paths (lines 2-3). The algorithm starts by evaluating the edges of the accepting state of the au-

Algorithm 13 Function to retrieve the paths triggering the failure

```
1:  function retrievePaths(aut, samps)
2:      paths = ∅
3:      currPath = ∅
4:      for all inEdge in aut.accState.inEdges do
5:          visitAut(inEdge, aut, samps, paths, currPath, samps.size() − 1)
6:      return paths
7:
8:  function visitAut(currEdge, aut, samps, paths, currPath, si)
9:      if currEdge.prop.evaluate(samps[si]) then
10:         currPath.push_front(currEdge.prop)
11:         si−−
12:         if currEdge.fromState == aut.rootNode then
13:             paths.push_back(currPath)
14:         else if si >= 0 then
15:             for all inEdge in currEdge.fromNode.inEdges do
16:                 visitAut(inEdge, aut, samps, paths, currPath, si)
17:         si++
18:         currPath.pop_front()
```

tomaton (where the assertion fails) with the last sample of the execution trace (lines 4-6). In the running example, the algorithm starts from state 4 of aut_1 with the sample obtained after the third input $\langle in3, 125 \rangle\}$. For each edge *aut.accState.inEdge*, the algorithm calls function *visitAut*. Among the inputs of *visitAut* we have the edge *currEdge* with which the function is trying to build a path and the index *si* to keep track of which sample must be used to evaluate the proposition on *currEdge*. At line 5, *visitAut* he is called with *si* equal to $sample.size() - 1$ to specify that the path is built from the last sample (last time unit). Function *visitAut* recursively visits the

inner edges of each state of *aut* in a DFS fashion (lines 8-18). Each time the function manages to build a path that connects the root state with the accepting state of *aut* (line 12), a new path is found and stored in *paths* (line 13). Fig. 6.10 reports the two failing paths retrieved from assertion a_1 in the running example.

In the final step of the procedure, each sequence of propositions is used to generate a decorated execution trace. Formally, a sequence of propositions $\langle p_1, ..., p_n \rangle$ is used to decorate an execution trace with a sequence of checkpoints $\langle c_1, ..., c_n \rangle$ where c_i is a function that returns true if p_i is true at time i, false otherwise. If all checkpoints return true, then the assertion must fail on the execution trace. Fig. 6.10 reports the execution trace decorated with one of the failing paths.

Once a decorated execution trace is generated, we can easily apply the techniques described in the second and third steps of the methodology by considering the differences highlighted below.

- The DPDG must consider the fundamental addresses of all the propositions in the checkpoints
- To determine if an assertion fails on a decorated execution trace, the simulator must verify that all the checkpoints return true.

6.4.4 Experimental results

The proposed methodology has been implemented in an automatic tool extending the KLEE symbolic engine. Its effectiveness and efficiency have been evaluated on four well-known C benchmarks compiled to LLVM:

- *xtea* implements the Extended Tiny Encryption Algorithm;
- *matrix mult* is a matrix multiplication algorithm;
- *graph DFS* is a depth-first search algorithm;
- *Newton-Raphson* is the famous root-finding algorithm.

The experimental results have been carried out on a 2.9 GHz Intel Core i7 processor equipped with 16 GB of RAM and running Ubuntu 20.04 LTS.

Table 6.6 reports the results in terms of execution time and reduction quality referred to an execution trace exposing a bug for each design. In particular, Table 6.6 compares the results of our tool with a *baseline* obtained by applying the best achievable reduction, that is, by manually inspecting the trace and removing the unnecessary instructions; indeed, this procedure can be performed only on short traces. The second column (*Original length*) reports the length of the original execution trace that makes the assertion fail, before applying any reduction. The third column (*Our approach*) reports the final length of the trace after applying our approach. The fourth column (*Manual Inspection*) reports the baseline. The fifth column reports the reduction quality as a ratio between "Manual inspection" and "Our approach". Here we can observe that our tool produces results very close to the baseline (reduction quality close to 1) for all the reported tests. The last column reports the execution time of our tool.

Table 6.7, instead, shows the scalability of our approach. It reports, for the *Netwon-Raphson* benchmark, the reduction percentage and the execution time at the increasing of the length of the target execution trace. These results show that our tool is capable, in a few seconds, of providing a reduction of over 60% of the original trace, even for traces hundreds of instructions long.

<**label**>**:0:**	<**label**>**:14:**
[0, 1] %1 = alloca i32	[26, 11] %15 = load i32, i32* %2
[1, 3] %2 = alloca i32	[27, 11] %16 = add add nsw i32 %15,10
[2, 5] %3 = alloca i32	[28, 11] store i32 %16, i32* %2
[3, 1] store i32 0, i32* %1	[29, 12] %17 = load i32, i32* %2
[5, 3] store i32 0, i32* %2	[30, 12] %18 = add nsw i32 %17, -1
[7, 4] store i32 5, i32* %3	[31, 12] store i32 %18, i32* %2
[8, 5] br label %4	[32, 18] br label %31
<**label**>**:4:** //in=0	<**label**>**:31:**
[9, 6] store i32 getNextInput(), i32* %1	[33, 5] br label %4
[10, 7] %6 = load i32, i32* %1	<**label**>**:4:** //in=125
[11, 7] %7 = icmp eq i32 %6, 0	[34, 6] store i32 getNextInput(), i32* %1
[12, 7] br i1 %7, label %8, label %11	[35, 7] %6 = load i32, i32* %1
<**label**>**:8:**	[36, 7] %7 = icmp eq i32 %6, 0
[13, 8] store i32 4, i32* %2	[37, 7] br i1 %7, label %8, label %11
[14, 9] %9 = load i32, i32* %2	<**label**>**:11:**
[15, 9] %10 = add nsw i32 %9, 1	[38, 10] %12 = load i32, i32* %1
[16, 9] store i32 %10, i32* %2	[39, 10] %13 = icmp slt i32 %12, 5
[17, 18] br label %31	[40, 10] br i1 %13, label %14, label %19
<**label**>**:31:**	<**label**>**:19:**
[18, 5] br label %4	[41, 13] %20 = load i32, i32* %1
<**label**>**:4:** //in=4	[42, 13] %21 = icmp sgt i32 %20, 90
[19, 6] store i32 getNextInput(), i32* %1	[43, 13] br i1 %21, label %22, label %31
[20, 7] %6 = load i32, i32* %1	<**label**>**:22:**
[21, 7] %7 = icmp eq i32 %6, 0	[44, 14] = load i32, i32* %2
[22, 7] br i1 %7, label %8, label %11	[45, 14] %24 = sub nsw i32 %23, 2
<**label**>**:11:**	[46, 14] store i32 %24, i32* %2
[23, 10] %12 = load i32, i32* %1	[47, 15] %25 = load i32, i32* %3
[24, 10] %13 = icmp slt i32 %12, 5	[48, 15] %26 = add nsw i32 %25, 3
[25, 10] br i1 %13, label %14, label %19	[49, 15] store i32 %26, i32* %3
	[50, 16] %27 = load i32, i32* %2
	[51, 16] %28 = icmp ne i32 %27, 12
	[52, 16] %29 = zext i1 %28 to i32
	[53, 16] %30 = call @assert

TABLE 6.5: LLVM execution trace of the running example

TABLE 6.6: Analysis of the reduction quality

Name	Original length	Reduced length		Reduction quality	Reduction time
		Our approach	Manual inspection		
xtea	190	155	155	1	1830 ms
matrix mult	150	127	122	0.96	2631 ms
Newton-Raphson	213	76	76	1	2056 ms
graph DFS	236	207	205	0.99	4623 ms

TABLE 6.7: Analysis of the approach's scalability

Original length	Reduced length	Reduction Time	Reduction
482	154	3s	68.05%
1379	389	36s	71.79%
10283	2888	437s	71.91%

Chapter 7

Runtime verification: CARMINE

7.1 Introduction

Thank to the recent advances in robotics and artificial intelligence, autonomous mobile robots are today adopted in a large spectrum of applications. These include smart manufacturing [100], surveillance [101], precision agriculture [102], warehouse [103], and delivery systems [104]. Nevertheless, their safety and reliability requirements as well as their robustness guarantees remain major barriers to their large-scale adoption in real-world systems. While different formal verification methods have been proposed to validate end-to-end their software correctness [105], [106], solutions to support *runtime system verification* are still understudied. Runtime verification is a mandatory component for the validation process of the robotic infrastructure. It is required to verify that the software implementing the robot's mission and behaviour is correct and also satisfies extra-functional constraints (e.g., real-time, energy efficiency, reliability) when run on a real robotic platform in the physical world.

Good design practice for runtime validation requires the use of ABV [107]. Assertions are first synthesised into *checkers* and then checked at runtime to report any violation and possibly enforce fail-safe behaviours. At the state of the art, checkers for robotic applications are generally applied to watch system resources and to detect local faults [108]. Nevertheless, with the increased complexity in perception and control, modern robots and autonomous systems need more advanced and complex monitoring tasks during their daily missions. Such monitoring tasks range from enforcing security and safety properties to pattern matching over sensor readings to help perception [109].

Runtime verification based on such a kind of checker introduces computational overhead, whose amount depends on the number of active assertions, the complexity of the assertions, and the observed signals. When adopted in resource-constrained architectures, such workload variability can lead to system overloads and failures even with very few checker instances. In similar contexts, migrating functional tasks from IoT edge devices to edge servers or to the cloud has shown to be a valid solution for freeing resources and avoiding system bottlenecks [110]. On the one hand, applying the same technique for migrating checkers may free resources for functional tasks at the edge. On the other hand, moving checkers far from the sensors or from the functional tasks that generate the observed events (i.e., signal values) may require continuous updating of these events through the communication network. As a consequence, the migration of checkers is a challenging task, as it could move the bottleneck problems from the edge to the network, with consequent inefficiency from the point of view of the overall system performance.

We address this challenge by proposing an ABV platform for the automatic generation, orchestration, and deployment of checkers across edge-cloud computing

architectures for robotic systems. Starting from LTL assertions expressing the system behaviours as required by its specification, the platform synthesises checkers compliant with the Robot Operating System (ROS) [111] standard. It then enables dynamic balancing of the SUV workload by considering a trade-off among verification accuracy, runtime constraints and resource requirements. This is achieved by a novel approach that allows migrating the runtime evaluation of the checkers across the different layers of the edge-cloud computing platform. Finally, the verification environment is containerised using Docker to enable portability.

The rest of this chapter is organised as follows. Section 7.2 summarises the related work. Section 7.3 defines the problem statement. Section 7.4 describes the overall verification architecture. Section 7.5 deals with checker synthesis. Section 7.6 is devoted to containerisation and deployment. Section 7.7 presents the dynamic migration of checkers. Section 7.8 discusses experimental results.

7.2 Related works

Several solutions have been proposed in the past to automatically synthesize checkers from their high-level specifications and integrate them into ROS-based designs, for both single robots [112], [113] and robot swarms [114]. In these solutions, temporal logic and regular expressions are the main adopted languages to describe system properties and temporal patterns. Such formal languages provide powerful environments to define temporal order and concurrency among states and events. Variants and extensions have been also proposed to deal with the complexity of such monitoring tasks [113], [115].

A common strategy is to design checkers to reproduce the original system specification according to a set of rules and the current inputs. Then, each checker alerts if, after such a transformation, a certain form has been obtained [116], [117]. As an alternative, the checkers are designed as automata, which are implemented through a big look-up table that maps all possible transformations for all possible inputs [118]. The static definition of the table provides better performance at runtime than rewriting-based checkers. Nevertheless, these approaches do not scale well in size as the automata could potentially become very large, non-compositional, and non-extensible.

A compositional and extensible approach relies on the design of checkers as a network of small computation nodes generated from temporal logic specifications [119]. The approach has been then extended for timed specifications [120], quantitative [121], and parametric [122]. Similar solutions have been applied for fault detection and condition verification of production facilities [123], [124].

The aforementioned solutions assume no limit from the point of view of the availability of computational resources for the execution of checkers. As a consequence, they are not effective in scenarios where strict limits must be respected concerning the overhead caused by the runtime verification of the SUV. Differently from all the other techniques presented in the literature, this work addresses the above limitations by taking advantage of the edge-to-cloud computing continuum, which is a key component of modern and complex robotic platforms.

7.3 Problem statement

As a starting point, we consider the SUV being implemented as a set of distributed software tasks, which are modelled through ROS nodes. The ROS nodes execute

on computing devices, which are distributed across an edge-cloud platform. This assumption is justified by the fact that ROS is the standard developing environment in the robotic community. It is based on a *publish-subscribe* communication paradigm, where a sender node publishes data on a *topic*, on which one or many nodes subscribe to receive the data. The adoption of ROS provides different advantages. First, it allows the modelling and simulation of complex systems through nodes running on different target devices. Second, it implements inter-node communication in a modular way to guarantee code portability. Finally, it adopts standards and widespread protocols requiring minimum intervention or modifications to the original code.

Given this premise, this work aims to automatically generate and deploy a monitoring platform for runtime verification of ROS-compliant robotic systems. The challenge consists in implementing a strategy for integrating checkers into the SUV implementation such that they ensure an accurate runtime verification of the SUV without negatively affecting its functionality (e.g., degrading the quality of the service or violating real-time constraints), which may happen if part of the computing resources has to be preempted to be periodically dedicated to the verification tasks.

Each ROS node is located at a computational layer between the edge and the cloud. Its position is initially chosen with the ideal goal of keeping the source of data as near as possible to the computing unit that will elaborate it. At the edge, since data is elaborated immediately, also monitoring and verification involve low response latency. On the other hand, edge devices are generally characterised by limited computational resources. As a consequence, the execution of the SUV functionality may be hampered by the addition of run-time verification as monitoring steals computational resources from other software tasks, possibly delaying their completion.

On the cloud, we assume no limitations in terms of computational capabilities. On the other hand, the end-to-end verification latency can sometimes be extremely high and unpredictable due to the variability of bandwidth and traffic on the shared communication network. As a consequence, failure notifications could be delayed. We propose a verification architecture that generates ROS-compliant checkers that can automatically migrate between computing layers in accordance with the available resources and requirements of the functional tasks to best take advantage of the strengths of each platform.

7.4 Verification architecture

Fig. 7.1 shows an overview of the proposed verification platform. It is intended to support assertion-based verification of robotic systems at runtime, by dynamically migrating the execution of checkers among the computational layers of the SUV implementation from edge to cloud. The platform automatically synthesises checkers from LTL assertions in the form $always(antecedent \rightarrow consequent)$. Checkers exploit the ROS publisher-subscriber paradigm to collect, at the passing of time, the values of the SUV variables involved in the target assertions, thus enabling their evaluation. In the following, we call *event* the value assumed by an SUV variable at a time instant t. Whenever an event occurs, the system publishes such information on a topic. Then, each checker subscribes to the topics associated with the variables included in the corresponding assertion, to receive all the events required for its evaluation. In this way, the platform permits the integration of checkers without modifying the source code of the SUV.

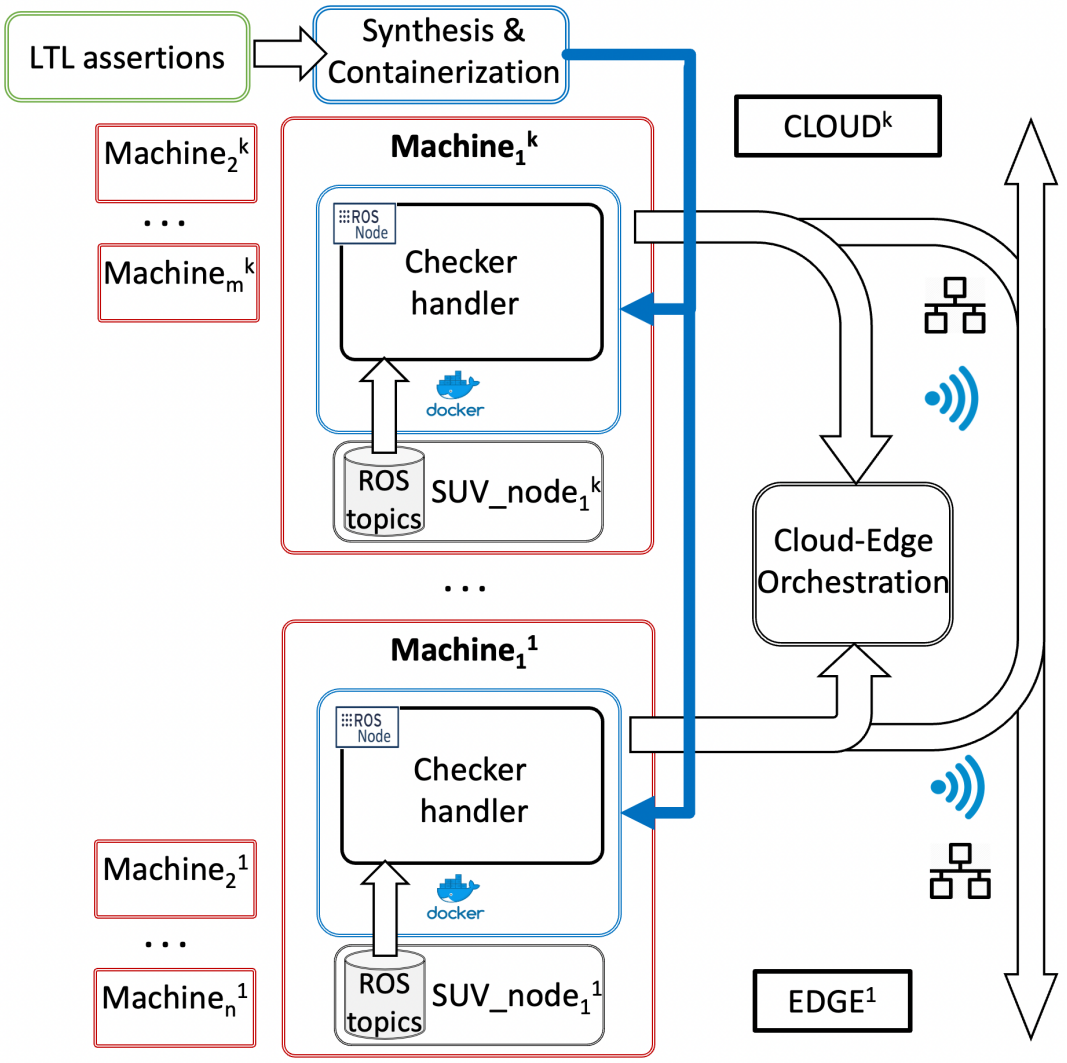

FIGURE 7.1: Verification architecture.

The execution of checkers is finally managed by a set of *checker handlers*. There is one checker handler per computing device in the system. The handler is a ROS-compliant node containing an orchestrator and an instance of every checker. At each execution instant, one (and only one) handler activates one instance per checker, according to the decision taken by the orchestrators. Such a decision relies on the analysis of the trade-off between verification accuracy, runtime constraints, and resource requirements.

In this way, we can set up an ABV environment that dynamically migrates the execution of checkers across the computing devices, taking into consideration both resource constraints and communication latency.

Through *containerisation* of the verification environment, the platform allows for the deployment of checkers across different hardware architectures and operating systems, as well as for handling the resources allocated for verification. The software application implementing the robot tasks is also containerised through Docker and orchestrated through Kubernetes/KubEdge.

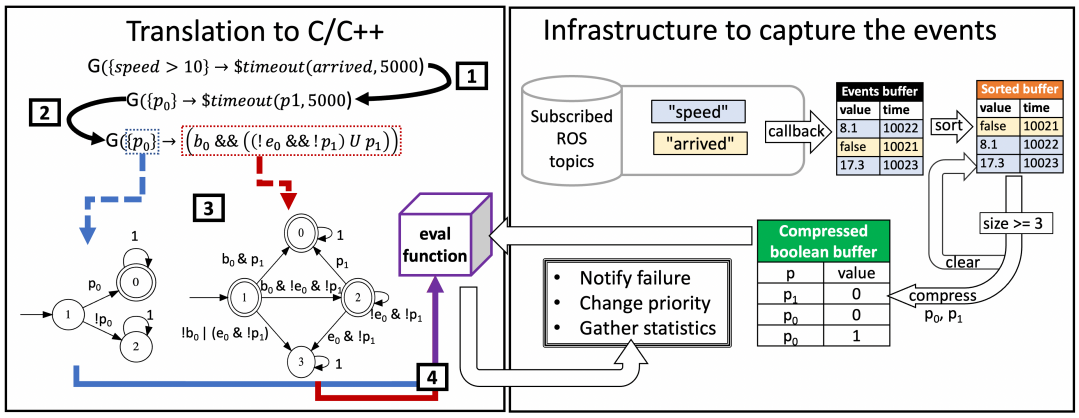

FIGURE 7.2: checker synthesis (left part) and event subscription (right part).

7.5 Checker synthesis

The input of our verification architecture is a set of assertions expressed by using an extended version of LTL, as detailed in Section 7.5.1. Assertions are then synthesised into checkers. They are composed of a C++ evaluation function to check the assertions dynamically, generated as described in Section 7.5.2, and a ROS-compliant handler to capture the events necessary for performing their evaluation, created as reported in Section 7.5.3.

7.5.1 Assertion grammar

Overall, our verification architecture allows the formalisation of assertions following the LTL logic. The employed grammar is similar to the one reported in 5.1.

The operators allowed in that grammar can be used to express the timing evolution of the system under verification; however, they are less suited to represent, in a compact way, deadlines, which are a fundamental ingredient to guarantee the predictability of tasks, at run-time, in robotic systems. Therefore, in this work, we introduce also a new operator to compactly handle deadlines, which we named *timeout*. The expression $\$timeout(proposition, N)$ evaluates to true if *proposition* becomes true within N milliseconds from the current instant of time; else, it evaluates to false.

7.5.2 Checker evaluation function

The process of translating an assertion to a C/C++ evaluation function consists of four main phases, as shown in the left part of Fig. 7.2:

1. substitution of each proposition in the assertion with a placeholder;
2. translation of the *timeout* operator to a standard LTL expression;
3. generation of equivalent Büchi automata for both the antecedent and consequent of the assertion;
4. translation of the Büchi automata to a C++ evaluation function.

Each proposition included in the target assertion is first substituted with a placeholder (i.e., a Boolean variable). This is motivated by the fact that Boolean information can be compressed through a single bit per value, which in turn allows a more efficient checker migration (see Sec. 7.7.

If the assertion contains the $\$timeout$ operator, in the second phase, it is translated into an LTL-compliant expression implementing the equivalent behaviour. This is necessary as the subsequent steps of the synthesis require an LTL expression as input, while the $\$timeout$ operator is not recognised by any standard. In general, the expression $\$timeout(p_i,\ N)$ is then translated to b_j & $((!e_j$ &$!p_i)\ until\ p_i))$, where b_j is a mock variable always equal to true while e_j evaluates to false if more than N milliseconds have passed since the "timer was fired", true otherwise. Note that b_j and e_j are internal variables of the checker, tailor-made to handle the $\$timeout$ operator; therefore, their values do not depend on the events received from the ROS topics.

To exemplify these two phases, let us refer to the running example shown on the left part of Fig. 7.2. The assertion:

$$always(\{speed > 0\} \rightarrow \$timeout(arrived,\ 5000)$$

is first substituted by:

$$always(\{p_0\} \rightarrow \$timeout(p_1,\ 5000)$$

where p_0 and p_1 are the placeholders for $speed > 0$ and $arrived$, respectively, and then, by replacing the *timeout* operator, it finally becomes:

$$always(\{p_0\} \rightarrow (b_0\ \&\&\ (!e_0\ \&\&\ !p_1)\ U\ p_1)$$

where e_0 becomes true after 5000 milliseconds since p_0 has fired.

Once the substitution of propositions and the translation of the *timeout* operators are completed, phases 3 and 4 take place for generating the evaluation function.

In phase 3, we exploit the spotLTL library [24] to automatically generate two Büchi automata, one for the antecedent and one for the consequent of the target assertion. After that, in phase 4, we exploit the automata to synthesise two corresponding evaluation functions, named *EVAL_ANT* and *EVAL_CON*, which are finally combined to build the overall evaluation function *EVALUATE*, as exemplified in Algorithm 14 (described later in this section) for the assertion shown in the left part of Fig. 7.2. *EVALUATE* exploits *EVAL_ANT* to determine the truth values of the antecedent each time a new event is received as input. Then, for each true instance of the antecedent, it calls *EVAL_CON* to evaluate the consequent, deriving the truth values for the whole assertion instance $antecedent \rightarrow consequent$. When *EVALUATE* detects a false instance for the assertion, a notification of failure is issued. We report below the details on the generation of the Buchi automata and the implementation of the corresponding *EVALUATE* function together with the related example.

7.5.2.1 Synthesis of the evaluation function

In our previous work [125], to implement the evaluation function, we generated a single deterministic Büchi automaton for the whole target assertion, without distinction between antecedent and consequent. In this case, the synthesis of the corresponding evaluation function consists of a procedure that, by visiting the Büchi automaton, generates a *case* of a *switch* statement for each state, and a next-state function inside each *case* for every edge of the state. By adopting this representation,

the global status of the checker is stored in memory by means of an unsigned integer indicating the "current state" of the automaton. This representation is quite convenient and straightforward to be implemented. On the other hand, it does not allow us to (i) keep track of how many times the antecedent and the consequence became true or false during the system execution, or (ii) to take note of pending instances (i.e., instances whose antecedent has been fired, but the evaluation of the consequent has not yet finished). Solving (i) and (ii) is paramount to ensure a fine-grained and high-quality verification. In particular, it allows us to distinguish between actual true assertion instances (i.e., when a fired antecedent implies the consequent holds) and vacuous passes (i.e. when the assertion is trivially true as the antecedent is not fired). In addition, we can also assess the severity of an assertion failure, by counting the number of instances where the antecedent is true and the consequent is false). At run-time, depending on the importance of the assertion, we might, indeed, decide to ignore an assertion failure if it is related to only a tolerable number of instances.

In the following, we then present a novel approach to generate an evaluation function that solves the two issues mentioned above. The main idea is to use a pair of counters ($curr_i$ and $next_i$) for each state s_i of each automaton. It applies to both the implementation of the antecedent (i.e., *EVAL_ANT*) and the consequent (i.e., *EVAL_CON*) functions. The counter $curr_i$ of the antecedent (consequent) automaton contains the number of antecedent (consequent) instances currently pending in a certain state, while $next_i$ retains the number of instances that will become active in the next time frame. Each time a new event must be evaluated (i.e., we have a new value for a variable in the assertion), the evaluation procedures for the antecedent and the consequent follow three steps:

1. Increment $curr_{root}$ by 1 for the root state of the automaton as a new pending instance enters the root state;

2. For each non-terminal state s_i with $curr_i > 0$, find an edge e connecting s_i with s_j that is true in the current evaluation (in a deterministic automaton, there must always be one edge satisfying this requirement), then increment $next_j$ (i.e., the *next* counter for s_j) by the value of $curr_i$. After that, set $curr_i$ to 0.

3. For each pair of counters related to a non-terminal state s_i with $next_i > 0$, swap the value of $next_i$ with $curr_i$. This is done to clear $next_i$ of the instances handled in the current time frame and to load $curr_i$ with the pending instances of the following time frame.

Instances of the antecedent (consequent), which increment the counter *next_i* related to any rejecting/accepting state s_i of the automaton, keep track of the number of times in which the antecedent (consequent) has failed/succeeded.

To exemplify the above procedure, let us consider the assertion reported in the left part of Fig. 7.2 and the corresponding automaton for the consequent. Fig. 7.3 shows how our procedure works for evaluating the assertion consequent in three consecutive instants. At time t_0, following step 1, the counter $curr_1$, corresponding to state 1 (i.e., the root) is set to 1. Then, at step 2, the edge connecting state 1 with state 2 contains the only true expression b_0 & $!e_0$ & $!p_1$; therefore, $next_2$ is incremented by 1 and $curr_1$ is set to 0. Finally, according to step 3, $next_2$ is swapped with $curr_2$. The same procedure is repeated at time t_1 and t_2. After the third evaluation instant (t_2), three instances fail as they reach the rejecting state 3.

To clarify the implementation details, Algorithm 14 reports the pseudo-code of the evaluation function automatically synthesised for the assertion of Fig. 7.2 according to the above idea. *EVALUATE* is responsible for separately evaluating the

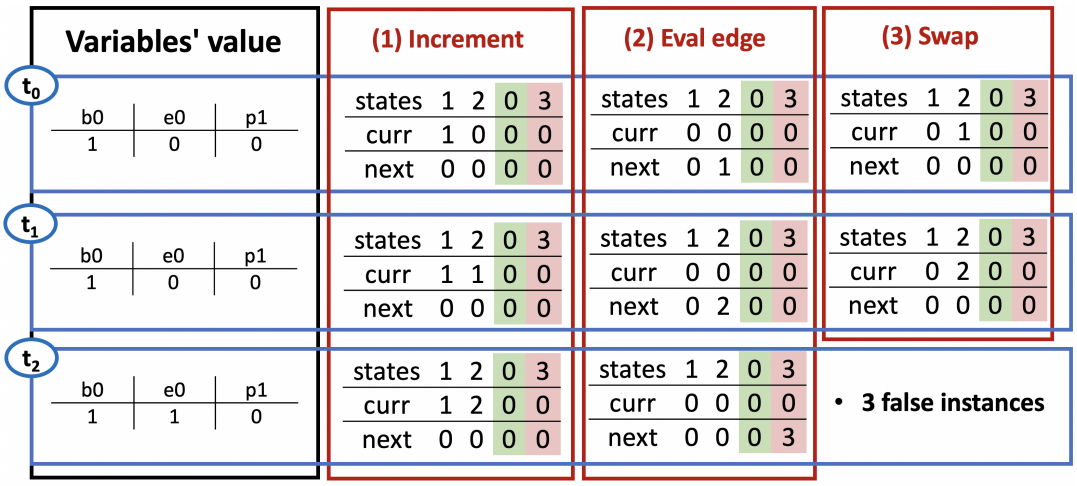

FIGURE 7.3: Evaluation of instances for consequent of the assertion reported in the left part of Fig. 7.2.

antecedent and the consequent of the assertion by calling functions *EVAL_ANT* and *EVAL_CON*, respectively. The inputs of these three functions are the values of placeholders p_0 and p_1. First, the function retrieves a new evaluation of the antecedent and stores it in *antResult* (line 2). *antResult* contains a pair of values, the first (second) value contains the number of instances that made the antecedent true (false) in the previous evaluation. If there are new true instances, the function evaluates the consequent as many times as the number of such instances (lines 3-5). If the evaluation of the consequent detects a false instance, then the function notifies the failure by calling $notifyFailure()$ (lines 6-7). Variables *_currAnt*, *_nextAnt*, *_currCon* and *_nextCon* correspond to the state of the automaton and are used to keep track of the active instances.

Function *EVAL_ANT* operates as follows. First, it initialises the result of the current evaluation (line 10), and then it increments by one the number of active instances in the root state (line 11). After that, if p_0 is true, then the function increments the number of successful instances (antecedent true, lines 12-13); otherwise, if p_0 is false, it increments the number of failed instances (antecedent false, lines 14-15). Note that the statements on lines 16-17 could be optimised away as the expression under evaluation is not complex enough to require the full use of *curr* and *next*.

Function *EVAL_CON* implements a variation of the above implementation, where the checker must also infer a proper value for the internal variables b_0 and e_0, which are used to handle the semantics of the corresponding *timeout* operator (see phase 2 at the beginning of Section 7.5.2). The idea is that the checker starts a new timer each time an active instance (or a set of active instances) traverses an edge containing b_i & $!p_j$; that would be b_0 & $!p_0$ in the running example. Conversely, the checker removes a timer (or a set of timers) each time an instance traverses an edge containing p_j and not containing b_i, or an edge containing e_i; those would be edges $2 \rightarrow 0$ and $1 \rightarrow 3$ in the running example. Note that b_i is used only to determine where to add new timers in the automaton, the variable itself is optimised away when generating the actual evaluation function. To add and remove timers, we use functions $addTimer(timerID, nInstances)$ and $removeTimer(timerID, nInstances)$ respectively, where *nInstances* is the number of instances connected to a certain timer identified with *timerID*. We use function $evalTimer(timerID, instanceID)$ to determine the value of e_i; the function returns true if a certain timer *timerID* has expired

for a certain instance $instanceID$, otherwise, it returns false. In the running example, we have only one timer with ID equal to 0. In function $EVAL\_CON$, Lines 23-34 cover state 1 of the automaton while Lines 35-46 cover state 2. The only difference from $EVAL\_ANT$ is that we use functions *removeTimer*, *addTimer* and *evalTimer* to determine the value of e_0 for each active instance (lines 27-34 and 39-46). Note that since the algorithm keeps track of timers as a sorted list of timestamps, where the most recent timestamps are stored at the end of the list, each time *evalTimers* returns false for a certain instance (timer has not expired), we do not need to evaluate e_0 for the rest of the instances, as we already know that *evalTimer* would return false (lines 32-34 and 45-46).

7.5.3 Checker handler

The checker evaluation function described in the previous section needs to receive as input the sequence of values assumed by the variables involved in the corresponding assertion, at the passing of time, during the SUV execution. This section describes how we generate a checker handler, compliant with ROS, to provide the evaluation function with such values.

The checker handler subscribes to all topics used to exchange information (i.e., values of variables) included in the corresponding assertion. For each variable, a callback is used to retrieve the interesting events (i.e., variable changes) from the corresponding ROS topic. The system attaches a callback procedure to an independent thread, which executes each time a message is processed from the subscriber queue. More formally, an *event* is a triple $\langle v, new\_value, timestamp \rangle$ to specify that the value new_value is assumed by the variable v at time $timestamp$, during the execution of the SUV. A captured event is added to an *event buffer* in the checker handler each time a callback is executed, delaying its processing. As stated in Section 7.7, the buffer enables the orchestrator to move the checker evaluation across the edge-to-cloud layers, without considering the location where the events were observed. Furthermore, the buffer can be sorted by making use of the timestamps, thus ensuring that the events are processed in chronological order. This minimizes the evaluation errors caused by synchronization problems and/or communication lags.

When the buffer's size reaches a certain threshold, it is sorted. A higher threshold improves verification accuracy since this increases the probability of evaluating the events in the correct order; however, this can severely reduce verification responsiveness. For this reason, every computing node in our architecture is synchronised with the precision time protocol (PTP) guaranteeing synchronisation errors of the order of nanoseconds while requiring minimal bandwidth and little processing overhead. Consequently, the sorting threshold can be set to a low value, allowing high verification responsiveness, while suffering from negligible accuracy errors.

After the buffer has been ordered, the Boolean placeholders of the target assertions are replaced by using the values corresponding to its events. These values are stored, in the same order, as Boolean constants, in a new *compressed buffer*, where the timestamps associated with the events are removed (as the events are already ordered). Each time the evaluation function is called, an event is consumed from the compressed buffer and used to advance the verification.

In the example reported in the right part of Fig. 7.2, the ROS handler subscribes to two topics: *speed* and *arrived*. They correspond to the variables involved in the assertion shown in the left part of Fig. 7.2. In this example, three events are captured and added to the *event buffer* by the callback functions: $\langle speed, 8.1, 10022 \rangle$, $\langle arrived, false, 10021 \rangle$, and $\langle speed, 17.3, 10023 \rangle$. If the sorting threshold, for example, was set to 3, after the

Algorithm 14 Evaluation function

```
1: function EVALUATE(p_0, p_1)
2:     antResult ← EVAL_ANT(p_0, p_1)
3:     if antResult.first > 0) then
4:         for i ← 0 to antResult.first do
5:             conResult ← EVAL_CON(p_0, p_1)
6:             if conResult.second > 0 then
7:                 notifyFailure()
8:
9: function EVAL_ANT(p_0, p_1)
10:    ret ← {0, 0}
11:    _currAnt[1] ++
12:    if p_0 then
13:        ret.first += _currAnt[1]
14:    else
15:        ret.second += _currAnt[1]
16:    _currAnt[1] ← _nextAnt[0]
17:    _nextAnt[0] ← 0
18:    return ret
19:
20: function EVAL_CON(p_0, p_1)
21:    ret ← {0, 0}
22:    _currCon[1] ++
23:    if p_1 then
24:        ret.first += _currCon[1]
25:    else
26:        for i ← 0 to _currCon[1] do
27:            e_0 ← evalTimer(0, i)
28:            if e_0 then
29:                ret.second ++
30:                removeTimer(0, 1);
31:            else
32:                addTimer(0, _currCon[1] − i)
33:                _nextCon[2] += _currCon[1] − i;
34:                break
35:    if p_1 then
36:        ret.first += _currCon[2]
37:        removeTimer(0, _currCon[2]);
38:    else
39:        for i ← 0 to _currCon[2] do
40:            e_0 ← evalTimer(0, i)
41:            if e_0 then
42:                ret.second ++
43:                removeTimer(0, 1);
44:            else
45:                _nextCon[2] += _currCon[2] − i;
46:                break
47:    return ret
```

arrival of the third event, the elements of the *event buffer* are ordered according to their timestamps and moved to the *sorted buffer*. Then, each event is compressed and added to the *compressed buffer*. Events $\langle speed, 17.3, 10023\rangle$ and $\langle speed, 8.1, 10022\rangle$ are compressed to 1 and 0, respectively, as the corresponding proposition $p_1 : speed > 10$ is true for *speed* equal to 17.3 and false for *speed* equal to 8.1; moreover, $\langle arrived, false, 10021\rangle$ is directly translated to 0.

7.6 Checker containerisation

The checkers are *containerized* after the synthesis process to make the verification environment portable across various HW/SW architectures. Different containerisation techniques are at the state of the art for cloud-native applications. Nevertheless, they provide each container with its own private (isolated) subnet IP addresses. As a result, they only allow ROS nodes to communicate if they are mapped to the same subnet IP. To solve the communication issues between distributed ROS nodes, we expanded the containerization process based on Docker for edge computing.

The proposed platform automatically maps each container IP address to the IP address of the host device (i.e., where the ROS node executes) while randomly allocating port numbers. This decreases communication latency by eliminating any network overhead caused by containers [126]. The network address translation (NAT) layer is also eliminated.

By using *multi-architecture containers* (e.g., Docker `buildx`), the platform supports the simple generation and integration of containers for different HW/SW target architectures, from cloud to off-the-shelf edge devices (e.g., NVIDIA Jetson). These multi-architecture containers are concealed behind a single container that has multiple integrated versions.

7.7 Checker run-time management

The orchestration aims at making a trade-off between verification responsiveness and resource consumption during the SUV execution. Low verification responsiveness may cause delays in the identification of assertion failures. Conversely, intensive use of computational resources for verification may cause the violation of run-time constraints related to functional tasks running on the same device, thus causing assertion failures.

In the optimal situation, a checker executes on the same edge node generating the values required by its evaluation. When computing resources are no longer enough to support both the verification effort and the execution of the functional tasks, our orchestration system migrates the execution of the checkers towards another computing unit, possibly belonging to a higher computational layer in the Edge-Cloud architecture. This way, additional resources should be available at the destination device to handle the checker, although at the cost of lower responsiveness. In fact, in this new configuration, the checker is evaluated farther from the source of the observed events.

7.7.1 Architecture and workflow of the orchestrator

The orchestrator consists of two main elements, a set of handlers, one per each computing unit in the SUV, and a global coordinator (see Fig. 7.4), which compose a fully connected verification network.

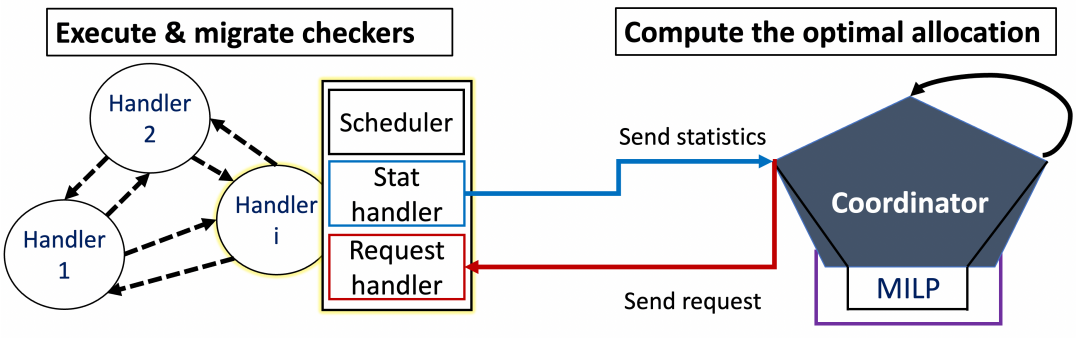

FIGURE 7.4: Orchestrator's architecture.

The coordinator orchestrates the allocation of checkers to the computing units in the SUV by continuously searching for the optimal allocation of resources. It then issues requests to the handlers to force such allocation. The coordinator requests are of the following types:

- EXEC request: the coordinator commands a handler to execute a certain set of checkers;
- MIGRATE request: the coordinator commands a handler to migrate a certain set of checkers to another handler;
- SHUTDOWN request: the coordinator commands a handler to stop executing.

Each handler is responsible for (i) gathering statistics about the state of the execution in the corresponding computing unit (through the *stat handler* module) and (ii) satisfying the requests from the coordinator (through the *request handler* module). It also controls the execution of checkers assigned by the coordinator through its *scheduler* module. Statistics are periodically sent by the handlers to the coordinator. They mainly include i) the current CPU usage dedicated to the verification process, and ii) the time elapsed from the publishing of a ROS topic (corresponding to the change of a variable) to the evaluation of the checkers that have subscribed for the topic.

The scheduler of each handler manages the dynamic allocation of computational resources to evaluate the elements stored in the event buffers (see Section 7.5.3) of allocated checkers. Each checker evaluation is seen as a request to be fulfilled by the scheduler. The scheduler satisfies these requests by spawning as many worker threads as the available cores of the device. A thread that receives a request keeps executing the evaluation function of the corresponding checker until a fixed time slice runs out or until the event buffer is empty. After that, the request is pushed back to the scheduler's queue. The scheduler handles the requests by using a priority delay queue. To avoid starvation, requests with low priority are promoted to higher priorities as their waiting time increases.

The orchestrator life cycle is then divided into three main parts: init, allocation, and termination.

At the *init* time, the verification network is initialised. Each handler discovers the existence of all the other computing units by sending a broadcasting message. Then the handlers run an election algorithm to determine which unit will execute the coordinator. The algorithm selects the handler executed on the unit with the highest available computational resources; such handler spawns an additional thread executing the coordinator. After that, each handler starts sending statistics to the coordinator and listening for requests. Note that the verification network is designed to

automatically include new nodes (handlers arrived late) and to elect a new coordinator if it becomes unresponsive for an extended period of time.

After that, the *allocation* phase takes place. The coordinator periodically analyses the statistics from the handlers of the SUV computing units and calculates the optimal allocation of checkers that maximises the verification responsiveness without saturating the CPU of any machine. Then, the coordinator sends EXEC or MIGRATE requests to handlers for enforcing the optimal allocation in the network. This is repeated throughout the SUV execution.

At *termination*, when the SUV execution stops, the coordinator issues a SHUTDOWN request to all handlers, and all nodes of the verification network terminate.

7.7.2 Computation of the optimal allocation

During the allocation phase, the coordinator periodically solves a mixed-integer linear programming (MILP) problem to determine where to allocate the checkers among the various computing units composing the SUV. The solution to the MILP problem corresponds to the allocation of checkers that maximises the responsiveness of the verification without saturating any CPUs. The coordinator constructs the MILP problem by using the statistics received from the handlers. The statistics are formalised as follows:

- $checkerCPU_{c_j}^{m_i}$ percentage of CPU usage spent to execute checker m_i on computing unit c_j;
- $topicCPU_{c_j}^{t_k}$ percentage of CPU usage spent to receive data from ROS topic t_k on computing unit c_j;
- $availableCPU_{c_j}$ percentage of unused CPU plus the CPU consumption of the verification process on computing unit c_j;
- $delay_{c_j}^{t_k}$ time (in milliseconds) to receive data from ROS topic t_k on computing unit c_j.

Formally, for M checkers, C computing units and T topics, we search for the allocation of any checker m_i to a computing unit c_j that minimises the objective function:

$$\sum_{k=1}^{T}(\sum_{j=1}^{C} getDelay\_t_k\_c_j(\bigvee_{i=1}^{M} m_i == c_j)). \tag{7.1}$$

The objective function is subjected to the following constraints:

$$\begin{aligned} &\bigwedge_{j=1}^{C}(availableCPU_{c_j} >= \\ &(\sum_{i=1}^{M} getCheckerCPU\_m_i\_c_j(m_i == c_j)+ \\ &\sum_{k=1}^{T} getTopicCPU\_t_k\_c_j(\bigvee_{i=1}^{M} m_i == c_j))) \end{aligned} \tag{7.2}$$

where, $m_1,\ m_2, ..., m_M$ are enumerated integer variables, associated with the checkers, that can assume one value among $c_1, c_2, ...c_C$, associated with the computing units. Intuitively, the above constraint ensures that the CPU usage of the resulting allocation of checkers does not exceed the available CPU on any computing unit.

The result of the MILP is an assignment of variables that minimises the objective function. The functions $getDelay\_t_k\_c_j$, $getCheckerCPU\_m_i\_c_j$ and $getTopicCPU\_t_k\_c_j$

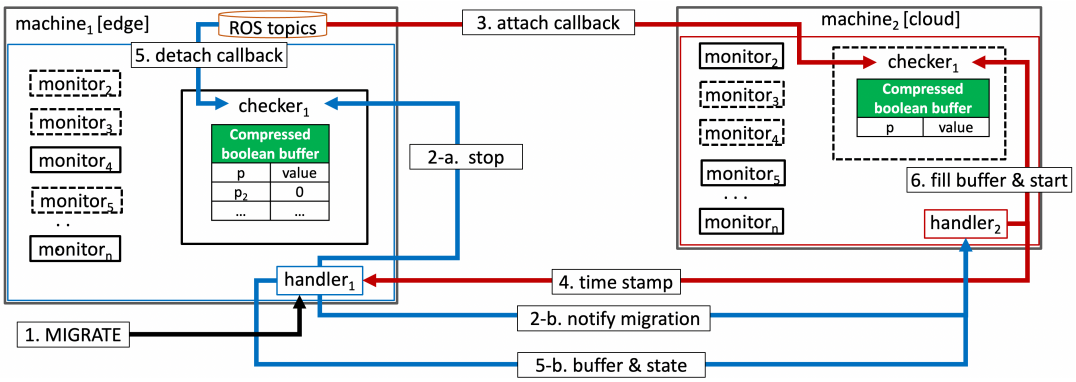

FIGURE 7.5: Example of buffer migration.

take as input a Boolean value, and they return zero if such a value is *false*; otherwise, they return, respectively, $delay_{c_j}^{t_k}$, $checkerCPU_{c_j}^{m_i}$, and $topicCPU_{c_j}^{t_k}$.

7.7.3 Checker migration

The migration of a checker takes place when the coordinator sends a MIGRATE request. Since each machine has a copy of all checkers, to move a checker m_i from the computing unit c_1 to the computing unit c_2 the migration procedure operates as follows: it first turns off m_i on c_1, then it sends the state and event buffer of m_i to $c2$, and, finally, it activates m_i on c_2. This procedure requires data to be moved through the network. However, the proposed migration strategy is incredibly effective because it is exceedingly lightweight, even in slower networks. This is due to the compression of the transferred data. Additionally, since no event is lost during migration, the suggested approach does not experience any false negatives (the checker fails when it should not) or false positives (the checker does not fail when it should).

To explain the migration protocol between the two machines, let us consider the example shown in Fig. 7.5. It shows $checker_1$ that executing at level l_i of the edge-to-cloud hierarchy, moves from $handler_1$ to $handler_2$ executing at level l_{i+1}. Before migration, $handler_1$ executes checkers 1 and 4, while $handler_2$ executes checkers 2 and 5. Considering that, at a given point, $checker_1$ receives a MIGRATE request from the coordinator (step 1), $handler_1$ removes $checker_1$ from the scheduler (step 2_a). The process of adding events to its buffer is not interrupted. Sequentially, $handler_1$ notifies the migration has started to $handler_2$ (step 2_b). As a consequence, $handler_2$ starts adding events to the buffer of $checker_2$ by attaching the callbacks (step 3). At this time, $checker_2$ is not yet on the scheduler. Once $checker_2$ receives enough events to make the first ordering, $handler_2$ returns to $handler_1$ the timestamp of the oldest event in $checker_1$ (step 4). This way, as soon as $checker_1$ gets the timestamp, it recognises which buffer events must be sent (i.e., the events evaluated with a timestamp lower than the received timestamp). When this happens, $handler_1$ detaches the callbacks from $checker_1$ to stop adding events to its buffer (step 5_a) and sends the correct events to $handler_2$ together with the state of $checker_1$ (step 5_b). At that point, $checker_1$ is inactive on $handler_1$ and $handler_2$ finishes the migration by filling the buffer of $checker_1$ with the received events and by putting the checker on the scheduler to start its evaluation (step 6).

7.7.4 Mending the worst-case scenario

The buffer migration procedure is intended to free computational resources on heavily loaded machines to balance the verification effort among the cloud-to-edge computation layers. This is of utter importance in systems where the lack of computational resources could severely affect the reactivity of the SUV, making its functional tasks miss deadlines or, in the worst scenario, causing the whole system to crash. However, it might happen that no allocation of checkers exists that does not saturate any CPU in the SUV. To handle this worst-case scenario, we implemented two additional strategies to reduce the computational load of verification: (i) by assigning the scheduling priorities according to the run-time criticality of the checkers, and (ii) by reducing the observation frequency of the events evaluated in the checkers. The first strategy impacts the verification responsiveness, while the second may affect the verification accuracy.

The first strategy consists of assigning a higher scheduling priority to checkers containing active instances (i.e., whose antecedent has been fired and the consequent is still pending), purposely increasing their responsiveness. In section 7.5.1, we showed that our evaluation function is capable of counting the number of active (antecedent/consequent) instances in a checker. After each evaluation, the priority of a checker is increased proportionally with the number of its active instances. The priority is the lowest when no instance is active.

The second strategy consists in discarding not-yet evaluated events from the event buffer and resetting the checker to its initial state. Thus, it performs an approximation of the event trace. When a checker is left behind in the evaluation, its event buffer keeps filling with past events. This can happen mainly for two reasons: either those events cannot be evaluated because the executing machine does not provide enough resources to the verification environment, or because of the low priority of the checker, which is considered non-critical by the scheduler, and thus, its evaluation function is executed with a lower frequency.

Approximating the event trace by removing "old" events from the buffer decreases the required resources, as the discarded events no longer have to be evaluated, and increases the responsiveness of the checker, as more up-to-date events are used in the evaluation. However, discarding events may affect verification accuracy. In fact, this approach can either cause a checker failure, when it should instead succeed (false negative), or can make the checker miss to detect a failure (false positive).

Fig. 7.6 shows an example to clarify this concept. Let us consider the simple assertion $always(a \rightarrow next(b))$. The variables a and b represent the input of the checker (reported in row I) in three different instants. The values in row O are the outputs of the checker. Evaluated events are shown in row E. In the upper part of Fig. 7.6, we show an example of a false negative. In this case, the values originally assumed by a and b, i.e., $(1,0), (0,1), (0,0)$, at event e_0, e_1, e_2 (table on the left) do not make the checker fail; therefore, the checker output is always 1. When event approximation is applied (table on the right), event e_1 is discarded. This causes the checker failure after receiving event e_2. Thus, the approximation produced a false negative. In the lower part of Fig. 7.6, we show an example of a false positive. In this scenario, the original event trace causes the checker failure after e_1. In the approximated trace, e_1 is instead discarded, thus making the checker misses the failure due to e_1. The approximation then produced a false positive.

Nevertheless, it is worth noting that, in our verification architecture, checkers synthesised from assertions following the template $always(antecedent \rightarrow consequent)$

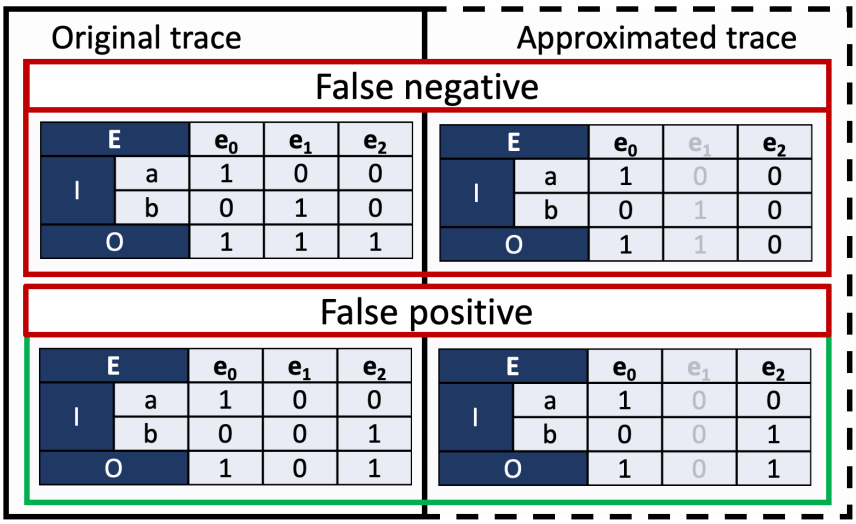

FIGURE 7.6: False negative/positive example.

are guaranteed to not produce false negatives because the checker is reset after discarding the events. Thus, only false positives are possible. Consequently, in scenarios where false positives are not the main concern, this approach is extremely useful to reduce resource consumption while maintaining good responsiveness.

7.8 Experimental Results

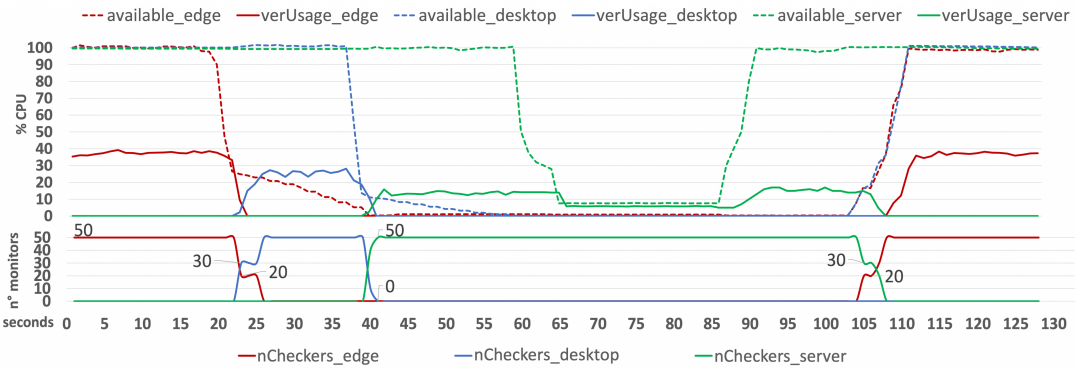

FIGURE 7.7: Verification statistics of the first case study.

We evaluated the effectiveness of the proposed approach in two case studies. In the first one, we set up a cluster of three nodes and implemented a synthetic benchmark to quantitatively evaluate the behaviour of the checker orchestration. In the second, we applied the platform in a real industrial case study, which implements the mission of a Robotnik RB-Kairos mobile robot in a smart manufacturing line.

7.8.1 Case Study 1: synthetic benchmark on a three-level cluster

We first tested the orchestration functionality described in Section 7.7 on a distributed Edge-Cloud system composed of three computing nodes. At the edge level, the cluster includes a 4-core Intel i7 (2.8 GHz) node with 4GB of RAM (i.e., *edge node*). At the cloud level, it includes a computing node equipped with a 16-core AMD Ryzen (3.9 GHz) CPU with 16GB of RAM (i.e., *server node*). It then includes a desktop machine

equipped with an 8-core Intel CPU (3.2 GHz, 8 GB of RAM), which is connected between the edge and the cloud nodes (i.e., *desktop node*). The server node lies at a 100ms (average) ping distance from the other nodes while the edge and desktop nodes lie at a 10ms (average) ping distance from each other. The verification environment consists of 50 checkers requesting input values from 5 different topics. All topics are published on the edge machine, which represents the most common configuration in most practical use cases. The orchestrator is set to recompute the optimal allocation of checkers every second. The SUV does not contain any tasks, while the CPU consumption of the computing nodes is artificially influenced by a custom testbench capable of generating a variable amount of computational workloads. The purpose of the testbench is to gradually saturate the CPU of the nodes, allowing us to study the behaviour of the orchestrator under such conditions.

Fig. 7.7 reports the quantitative history of the checker orchestration and the CPU utilisation of each node while executing both the testbench and the verification environment. It includes the available CPU of the different nodes (*available_edge*, *available_desktop*, *available_server*), which represents, in percentage, the total CPU minus the CPU used by the SUV tasks. The verification environment usage (*verUsage_edge*, *verUsage_desktop*, *verUsage_server*) corresponds to the percentage of CPU used to execute the verification environment on the corresponding machine. The values *nCheckers_edge*, *nChecker_desktop* and *nChecker_server* represent the number of checkers currently executing on the corresponding node. On the x-axis, the figure reports the time (in seconds) that elapsed from the beginning of the simulation. At time 0, all nodes had 100 percent of the available CPU and the edge node was executing all 50 checkers using, on average, 38 percent of the available CPU. After 20 seconds, the testbench started saturating the CPU of the edge node. As a consequence, the node was no longer able of handling the computational load of the verification environment. As a consequence, the orchestrator enabled the migration of checkers to the desktop node, which had available computing resources and guaranteed the least reduction in the verification responsiveness. The checker migration, as expected, was performed incrementally at the varying of the testbench workload. Since the edge CPU was not instantly saturated, the orchestrator started moving 20 checkers after 22 seconds, and then, at the next allocation period (second 23), as the edge CPU had been further loaded by the testbench, it migrated the remaining 30 checkers. From 23 to 40 seconds the desktop node executed all the checkers. After that, the testbench saturated the CPU of the desktop node to test the migration from the desktop to the server. Similarly to the previous iteration, the orchestrator migrated the checkers on the server node, which was the only remaining device with available CPU resources. Saturating such a node led the system to fall into the worst-case scenario. Note that this time, the checkers were transferred all at once towards the server, since the testbench reduced the available CPU on the desktop machine more quickly than in the previous transfer (from the edge machine), forcing the orchestrator to free computational resources on the desktop machine more quickly. To observe the behaviour of the verification environment in the worst-case scenario, the testbench started partially saturating the CPU of the server node (65 seconds), leaving only 8 percent of the available CPU. Since the verification environment required 18 percent of the CPU, it started dropping events from the buffers of the checkers, effectively reducing the computational cost of verification to 7.5 percent (on average). Then, the testbench removed the computational load on the server machine (87 seconds), restoring the execution of the checkers to their original accuracy and CPU consumption. Finally, the testbench removed the computational load on all nodes. As a consequence, the orchestrator moved back all the checkers to the edge

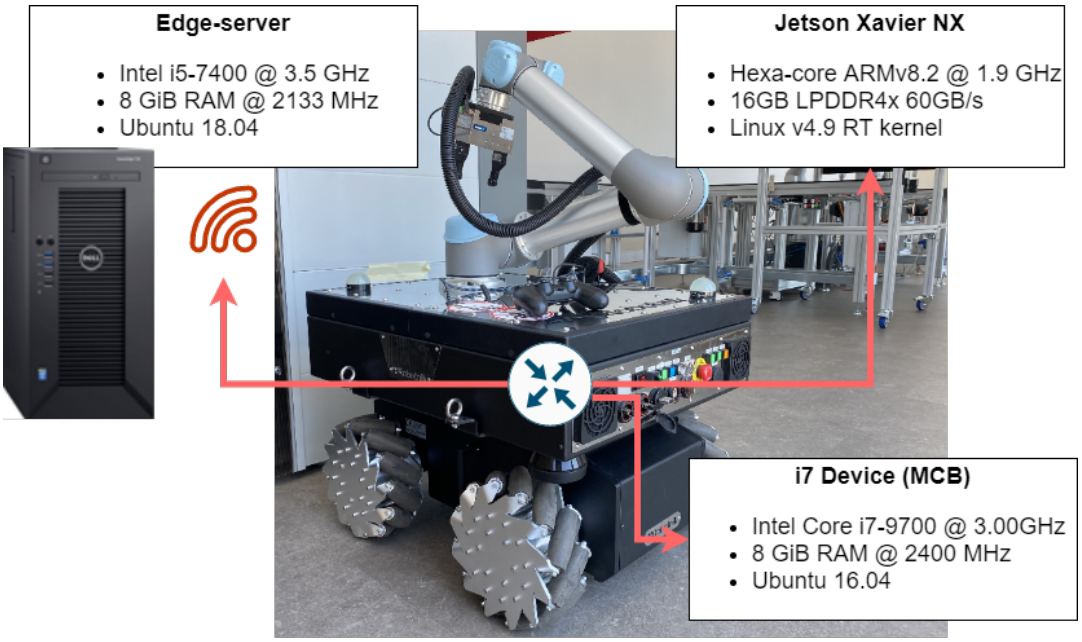

FIGURE 7.8: Overview of the programmable cluster nodes in the second case study.

node where verification could achieve the best responsiveness (105 seconds). During the entire simulation, the orchestrator spent on average 27ms (every second) to solve the MILP problem, proving the scalability of this approach.

7.8.2 Case Study 2: autonomous mobile robot for a smart manufacturing line

We evaluated the proposed methodology to asses the mission of a Robotnik RB-Kairos mobile robot in an industrial agile production chain. Such a mobile robot consists of a skid-steering platform equipped with a Universal Robots UR5 manipulator and a Schunk WSG50 end-effector for grasping (see Fig. 7.8). The robot is also equipped with different sensors including two Sick S300 laser scanners and an RGB-D Intel RealSense D415 camera for localization. The computing HW architecture consists of four edge devices installed on the board. One main control board (MCB) is equipped with an Intel i7 9700 3.0 GHz, 8GB of RAM, and Ubuntu 16.04 OS with ROS Kinetic. Two additional devices are installed for the real-time control SW of the UR5 manipulator and WSG50 gripper, respectively. They run a real-time Ubuntu OS and real-time kernels that communicate with the driver nodes on the MCB, they are not available for the MILP solver. The fourth edge device consists of an Nvidia Jetson Xavier NX with JetPack 4.5. The onboard devices are connected through a local gigabit Ethernet router (802.3ab). The cluster of nodes also includes an external server equipped with an Intel i5-7400 3.5 GHz, 8GB of RAM and Ubuntu 18.04. The server is connected to the onboard devices through an 866Mbps wireless network (802.11ac).

We configured a k3s cluster, version 1.20.4+k3s1, on the main control board, on the Jetson Xavier and on the external server. The server also runs the k3s master agent.

We measured the system performance while executing the ROS-compliant SW application implementing the robot mission. It consists of several tasks that implement the interaction of the mobile robot with an industrial agile production chain. Firstly, the robot is initialised to a starting position and aligned with the production chain. Then, a series of arm and gripper operations perform the grasping and the movement of production pieces from the conveyor belt to the cargo bay. The robot then moves the base towards the storage area, where it unloads the pieces from its cargo bay. Finally, the robot returns to the production line and re-aligns itself with the conveyor belt. It then moves the arm to the ready position. The total runtime is approximately two minutes.

The software for the Kairos is split between all three computing nodes inside the k3s cluster. The MCB handles most of the robotic functionality such as mission, arm and gripper planning, etc. The Jetson node is tasked with the navigation stack, while the server node handles HMI and monitoring. To collect the performance metrics we used Prometheus connected to Grafana. As the server node is tasked with scraping the data, its CPU usage fluctuates significantly when there is a data update.

We configured the MILP solver with threshold CPU utilisation for the MCB of 40%, 60% for the Jetson, and 90% for the external server. These thresholds allow the Kairos HW to continue working properly and not be overwhelmed by the computational load when the checkers are running together with the robotic software. The system includes 201 checkers running on the cluster, on all three computing nodes. We also used a moving window to average the last 10 values obtained from the ROS topics to create a low-pass filter that removes signal spikes that often happen when analysing real sensor data.

Functional analysis: With the use of RGB cameras and an inference-based image recognition system [127], the robotic software implements an ORB-SLAM [128] application for localization and mapping. It is then connected to a *move-base* system that relies on a global and local planner for obstacle avoidance.

Considering the temporal constraints of the software system, the aim was to check the functional correctness of the applications. In particular, we designated the global planner as a non-critical task on the server. We considered the ORB-SLAM and local planner running in real time on the Jetson (i.e., 125 ms application makespan). We enforced a corresponding minimum supported rate of 8 FPS for the RGB camera input stream.

To evaluate the effect of orchestration, we considered a scenario in which the robot has to reach a user-defined location $\langle x_2, y_2 \rangle$ from the starting point $\langle x_1, y_1 \rangle$. First, the global planner finds the path to reach the arrival point. Subsequently, while the robot moves, the local planner reschedules the path trajectory (waypoints) to take care of changes in the environment that may cause unwanted collisions with moving obstacles.

After the initial allocation of 201 checkers on the Jetson, we observed that the checker corresponding to the assertion $always((robot\_x_1 = x_1 \;\&\&\; robot\_y_1 = y_1 \;\&\&\; newGoal) \rightarrow (currentTime < timeOut \; U \; robot\_x_2 = x_2 \;\&\&\; robot\_y_2 = y_2))$ failed. This checker was set up to check if the robot can reach $\langle x_2, y_2 \rangle$ before a given timeout. The checker failed due to the overhead introduced by the verification environment, which causes an increment in the execution time. Consequently, the robot moved in the wrong direction because the robot's controller was unable to support the minimum updating frequency for the motor velocities. For proper operations, the controller for the motor velocities needs an update at least every 125 ms (8 Hz). The system guarantees a makespan of 115 ms without the verification environment (8.7Hz). When the verification overhead is introduced, this value increases to 140 ms (7.1 Hz). When, thanks

to the buffer migration approach, 150 checkers are automatically transferred from the Jetson to the server during execution; therefore, some of the verification load on the edge device is relieved. The robot's controller resumed functioning normally with a makespan of 125 ms (8 Hz), which prevented the aforementioned assertion from failing.

Component	Devices	No checkers	checkers
CPU	Jetson	38.64%	45.49%
	MCB	23.44%	28.50%
	Server	30.96%	50.98%
Network	Wi-Fi	0.31 MiB/s	0.39 MiB/s
	Ethernet	1.31 MiB/s	1.41 MiB/s

TABLE 7.1: Resource usage overhead with and without checkers.

Orchestration and overhead: Table 7.1 summarises the CPU and network usage with and without checkers. We can observe higher CPU usage overall, distributed across the computing cluster, and a slight increase in network usage of ≈ 0.1 MiB/s. The high bandwidth on the Ethernet interface is caused by the constant communication to handle the navigation SW components deployed on the Jetson. We also observed a negligible and static increase of $\approx$ 40MB of system memory usage on all nodes of the cluster due to the allocation of the resources needed by the checkers (not reported in table 7.1).

Figure 7.9 shows the overhead in more detail, as well as the number of running checkers on each node during the mission of the RBKairos. As reported in the summary table, CPU usage is higher overall for all devices due to the additional SW for the checkers running, but the highest average increase is found on the server and Jetson. On these two devices, the checkers spend most of their time as reported in the last graph (yellow and green for server and Jetson respectively). We also observed notable CPU usage spikes when the checkers are moved onto a particular device. An example is at the time instant 29, where some checkers are moved to the MCB and we can see the CPU usage reach the threshold, causing a migration towards the server. It is also evident at the time instant 85, where all checkers are migrated towards the MCB and its CPU usage exceeds the threshold, also causing a reduction in CPU usage on the other two devices. When the threshold value is exceeded, the **MILP** begins a migration towards the Jetson and also towards the server, causing spikes in usage on all the involved CPUs, but freeing the crucial resources needed by the MCB to correctly run the robot SW.

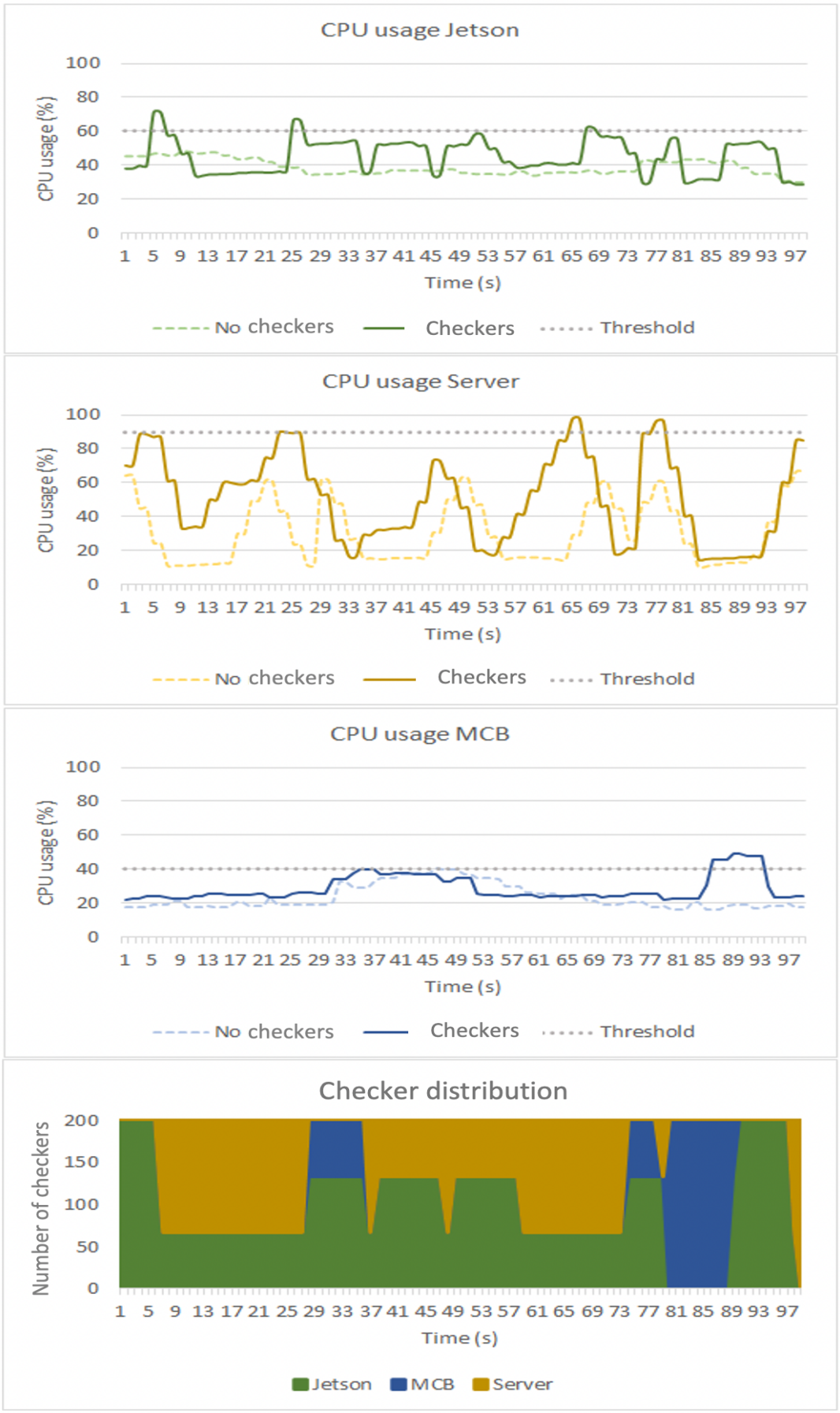

FIGURE 7.9: CPU overhead for all the nodes correlated to the number of checkers during the execution of the robot's mission.

Chapter 8

Design Exploration for Approximate Computing: DEA

8.1 Introduction

The Approximate Computing (AxC) paradigm was introduced to achieve higher power efficiency, lower area and better performances w.r.t. a "classical" computing system at the cost of a degraded, but still acceptable, output accuracy [129]. In recent years, AxC has been widely adopted since many of the most popular applications we use, such as digital signal processing of images or audio, data analytics, machine learning, web search and wireless communications have an inherent resiliency to errors due to imprecise computation [130]–[134].

AxC can be applied at several abstraction levels of a given computing system: from circuit to algorithm [129] leading to a wide design exploration space that quickly became the bottleneck to successfully deploy AxC. Indeed, the literature proposes many works to automatically trade-off between output accuracy and performances [135]. However, most of them lack the capability to identify resilient elements (e.g., HW component, HDL statements, etc.) of the design to be approximated. Consequently, exploring the design for AxC generally results in a long and tedious procedure.

Usually, existing approaches generate approximate variants of the Design Under Exploration (DUE). Every variant is then executed/simulated in order to determine the accuracy degradation [136]. The problem is that the accuracy depends on the application, and thus it requires a specific metric to be computed (e.g., similarity index, hamming distance, etc.).

Recently, the authors of [137] have presented a method to automatically identify parts of the DUE as candidates for approximation without significantly compromising the design correctness. Then, to evaluate the effect of approximations they use assertions, where an assertion is a logic formula that captures a specific functional behaviour implemented in the design [138]. In this way, there is no need to use a particular functional metric. In [139] assertions are evaluated on the approximated design for measuring how much the approximation alters the design functionality. However, assertions themselves are not exploited to identify the statements of the DUE more suited for the approximation.

Alternatively, this work proposes an effective way for guiding the approximation of the DUE, by leveraging the combination of fault injection and assertion-based verification. In particular, two approximation techniques are considered, bit-width and statement reduction, and fault injection is used to mimic their effect on the DUE. Assertions are then automatically mined from the original implementation of the DUE to capture its functionality. These assertions are then re-evaluated on the faulty designs to analyse the variations of their truth values with respect to the original implementation. These variations are then used to rank the different approximation

alternatives, according to their estimated impact on the functionality of the target design. The output of our methodology is a list of DUE elements (either bits of signals/registers or statements) ordered by increasing levels of severity: the higher the criticality, the more prominent the effect of approximating an element on the functional correctness of the original design. A clustering procedure is then exploited to suggest approximations to be applied simultaneously.

The rest of this chapter is organised as follows. Section 8.2 summarises the related work; Section 8.3 reports our methodology; Section 8.4 details the results obtained on the case study.

8.2 Related work

As stated before, one of the most challenging problems in AxC is selecting the "portion" of the application to be approximated. The portion depends on the abstraction level: it can be an instruction of the source code or a statement in the HDL representation. Existing methodologies allow the designer to explicitly select which lines or code blocks to approximate and how. For example, a programmer can annotate through pragmas that a given loop has to be approximated by applying the loop perforation technique [140]. Without annotations, each code line has to be considered as a potential approximation target, thus leading to virtually infinite possibilities of approximations.

At the hardware level, we have to cite ABACUS [141], which directly works with RTL implementations (i.e., HDL code). The proposed design-space exploration leverages a greedy algorithm to find a trade-off between accuracy and power consumption. In [136], [142], design space exploration exploits a genetic approach. Other approaches manually identify approximable sub-parts of circuits, mainly focusing on arithmetic components [143]–[145].

On the other hand, even if they aim at different goals, different approaches have been proposed for applying verification techniques to approximate computing. In [146], the authors present an AxC-based approach to achieve a fast and accurate enough repeated execution for security verification. In [147], the authors propose a dynamic verification methodology to assess the quality of the approximated circuit by exploiting mutations of test patterns and coverage information. However, none of the previous verification-oriented methodologies uses assertions to guide the design exploration. In [137], the authors introduce a verification-guided method to automatically identify program blocks suited for approximation, while avoiding significant compromises on program correctness. The method is based on identifying regions of code that are less influential for the computation of the program outputs and therefore, more approximable. In particular, they generate a statement ranking based on whether an instruction affects a particular primary output of the designs. Assertions are then used to evaluate the impact of the approximation on the functional behaviour of the final design. It is important to note that assertions are not directly used for statement identification (i.e., identify which portion of the code has to be approximated) as we do in this paper.

8.3 Methodology

Our methodology is intended to provide the designer with an automatic way to explore AxC alternatives on RTL descriptions. Two approximation strategies have been considered:

- **Bit-width reduction:** fixing to a constant value one or more bits on a subset of design signals/registers;
- **Statement reduction:** removing one or more statements.

The methodology consists of the three sequential steps shown in Fig. 8.1. The input is the RTL description of the DUE. The output is a ranked list of DUE elements that can be approximated. In the rest of the paper, we refer to such elements with the term *approximation tokens* (AT). In particular, according to the addressed AxC strategies, we consider the following kinds of ATs:

- **Statement token:** an instruction appearing in the RTL description of the DUE;
- **Bit token:** a bit of a signal/register occurrence appearing in a statement token.

Hereafter, we provide an overview of the three main steps of the methodology. A detailed description for each of them is then reported in the following sections.

1. **Trace generation:** in the first step of the methodology, we dynamically simulate the DUE and its approximations to generate a set of execution traces. First, we generate the golden trace by simulating the original implementation of the DUE. Then, for each AT, we generate a trace that reflects the effect of activating a corresponding approximation (i.e., bit-width reduction for bit tokens, and statement reduction for statement tokens). These traces are obtained

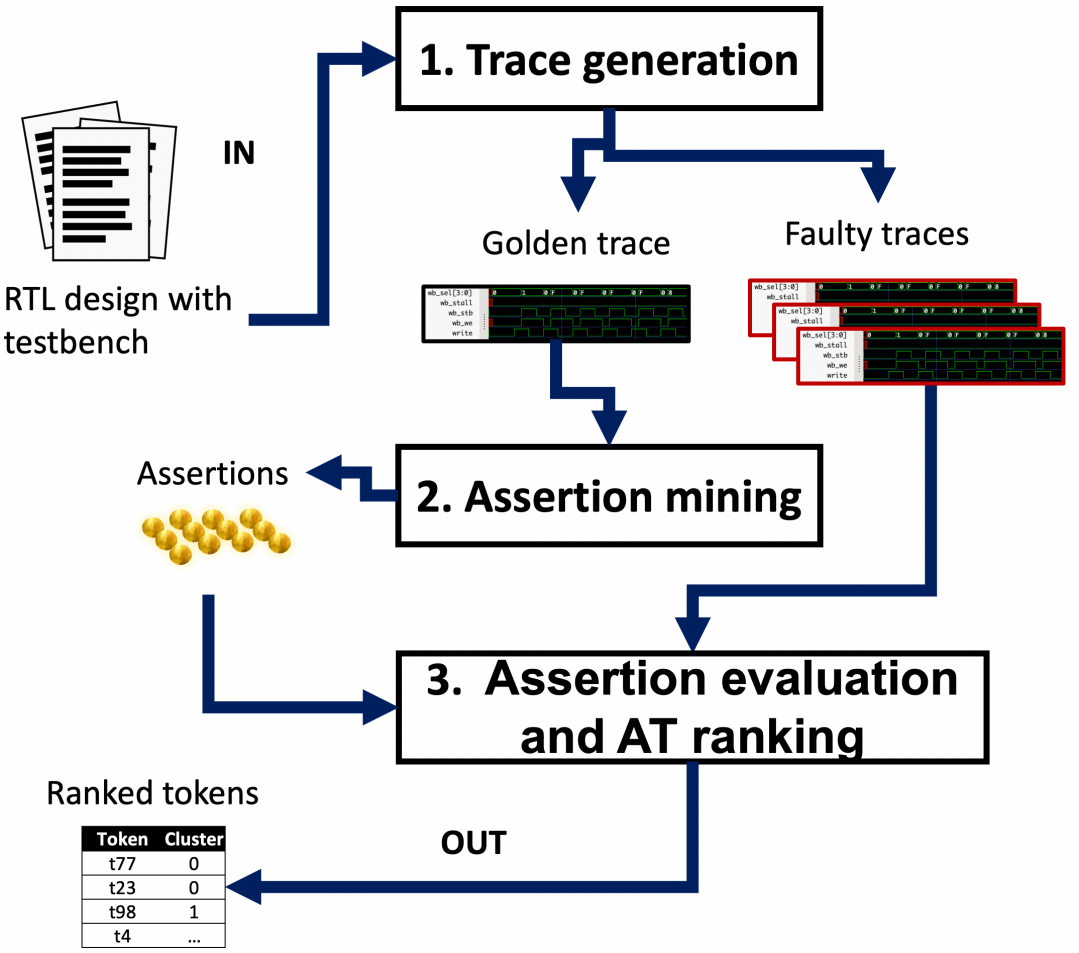

FIGURE 8.1: Overview of the methodology.

by simulating a faulty version of the DUE. In particular, stuck-at faults are injected in specific elements of the original implementation to exactly mimic the effect of approximating each AT.

2. **Assertion mining:** in the second step of the methodology, we employ an assertion miner to generate assertions holding on the golden trace. The assertions predicate on the inputs and outputs of the DUE and capture its golden behaviour.

3. **Assertion evaluation and AT ranking:** In the last step of the methodology, we re-evaluate the assertions mined in step 2 on the faulty traces obtained in step 1. By comparing the contingency tables of the assertions evaluated on the golden and faulty traces, we automatically estimate the approximability of each AT, in terms of how its approximation affects the functionality of the original design. Finally, we rank the ATs in order of decreasing approximability. ATs with similar approximability are clustered. The designer is then suggested to approximate the design by applying the approximations corresponding to the cluster that includes the top-ranked ATs.

To simplify the detailed exposition of each step, in the following sections, we refer to the running example reported in Algorithm 15. It takes as input two 8-bit unsigned integers (signals a and b) and returns their sum as output (out port). We assume that the adder module is stimulated by an external test tench, such that the two least significant bits of signal a and b (i.e., a[7:6] and b[7:6]) remain unused.

Algorithm 15 Running example

```
1: module adder(a, b, clk, out)
2: input [7:0] a, b
3: input clk
4: output [8:0] out
5: reg [8:0] sum
6: always @(posedge clk)
7: begin
8:       sum = a + b
9: end
10: endmodule
```

8.3.1 Trace generation

In the first step of the methodology, we simulate the DUE to generate a set of execution traces. We assume that the test bench thoroughly stimulates the design, purposely covering all its functional behaviours. This is a desirable condition for any dynamic approach.

First, we generate the golden trace by simulating the original implementation. Then, for each target AT, we generate a trace reflecting the effect of its approximation according to either the bit-width reduction or the statement reduction strategy.

For each class of AT, we identify a fault model to mimic the effect of its approximation in the functional behaviour of the DUE as follows:

- A bit token is approximated by injecting a stuck-at $0/1$ on the target bit. Stuck-at X has been not considered as the propagation of X values along the execution traces would prevent the evaluation of mined assertions in step 3.

- A statement token is approximated differently depending on the type of statement as follows:
  - assignment: the statement is removed; in case its left-hand side remains undefined, its value is assigned to 0 for bit-vectors and numeric types, stuck at 0/1 for a single bit/Boolean;
  - module instantiation: the statement is removed; in case the signals connected to the outputs of the module remain undefined, they are treated as in the case of assignments;
  - conditional statement: either the true or the false block is removed; in case any signal/register remains unsigned, it is treated as in the case of assignments.

8.3.2 Assertion mining

In the second step of the methodology, we automatically mine assertions holding on the golden trace of the original DUE implementation. To extract them, we exploit the state-of-the-art assertion miner available at [3].

The miner has been configured to automatically generate assertions on primary inputs and outputs of the DUE in the form $always(antecedent \rightarrow next[N](consequent))$, where N is the design depth, that is, the number of clock cycles necessary to propagate the effects of primary inputs toward primary outputs. This is done to ensure that the mined assertions represent meaningful I/O relations.

Both the *antecedent* and the *consequent* of each assertion are instantiated by the tool following the form $prop_1$ && $prop_2$ && ... && $prop_k$. Each proposition $prop_i$ is in the form $c_l <= var_j <= c_r$, or $var_j == c$, or $var_j <= c$, or $var_j >= c$, where var_j is an input or an output of the DUE located in the *antecedent* or the *consequent*, while c_l, c_r and c are numeric constants.

Finally, we rank the assertion set according to a score S, which is obtained, for each assertion a, by combining the following metrics:

- Metric 1: $Support(a) = ATCT/traceLength$;
- Metric 2: $Causality(a) = 1 - AFCT/traceLength$;

where, $ATCT$ and $AFCT$ are derived from the contingency table of a (see Def. 10). The score S is calculated as already described in section 5.8.

After the ranking procedure is completed, we keep only the assertions whose score S is greater than 0.

Figure 8.2 shows some of the assertions mined for the running example and the corresponding values for $Support$, $Causality$ and S.

8.3.3 Assertion evaluation and AT ranking

In the last step of the methodology, the generated assertions are re-evaluated on the faulty traces obtained by perturbing the original RTL description of the DUE by adopting the bit-width and the statement reduction strategies, as reported in Section 8.3.1. In particular, for each AT, the corresponding fault is injected and the assertions are re-evaluated generating a new contingency table. It is worth noting that the value ATCF is 0 in the contingency table of any assertion for the original implementation (as all assertions are true on the golden trace), while it is likely greater

N	Assertions	S	Causality	Support
1	G((a >= 58 && a <= 60) && (b >= 23 && b <= 63) |-> out >= 81 && out <= 126)	0.98	0.91	0.68
2	G((a >= 4 && a <= 63) && (b >= 61 && b <= 63) |-> out >= 65 && out <= 126)	0.96	0.64	1.00
...	...	...	...	...
38	G((a >= 28 && a <= 29) && (b >= 34 && b <= 35) |-> out >= 64 && out <= 73)	0.1	0.67	0.18

FIGURE 8.2: Assertions mined for the running example.

than 0 for assertions affected by the presence of a fault. Consequently, we can observe variations in the value of ATCT as well. The purpose of this procedure is then to analyse the effect of the different approximation alternatives (represented by ATs) on the functional behaviours (captured by the assertions) of the design.

The whole procedure is implemented in the *evaluate* function reported in Algorithm 16, whose behaviour is detailed hereafter.

Algorithm 16

```
1:  function EVALUATE(A, gt, FT)
2:      GCT ← ∅
3:      diff ← ∅
4:      for all a ∈ A do
5:          GCT[a] ← genContingency(gt, a)
6:      for all ft ∈ FT do
7:          for all a ∈ A do
8:              fct ← genContingency(ft, a)
9:              diff[ft] += abs(GCT[a] − fct)
10:     return diff
```

The function takes as inputs A, gt and FT, where A is the set of assertions mined in the previous step, gt is the golden trace and FT is the set of all faulty traces generated in the first step of the methodology. Note that for each $ft \in FT$ there exists a corresponding unique approximation token at and a unique fault f. The function returns a dictionary $diff$, which stores, for each ft, a matrix representing the sum of the differences between the contingency tables achieved on ft and gt for all assertions belonging to A.

First, we initialize variables GCT and $diff$, where GCT is intended to store the contingency tables of the golden trace (lines 2-3). Then, we iterate over the assertions in A such that the function genContingency(gt, a) evaluates the assertion a on the trace gt to retrieve the corresponding contingency table (line 5). After that, for each couple $\langle ft, a \rangle \in FT \times A$, a "faulty" contingency table is generated and stored in fct (line 8). ftc is then compared with the golden contingency table $GTC[a]$; this is done by returning the absolute difference between the two matrices (line 19). The result of this operation is stored in $diff[ft]$, which contains the impact of fault f on all the considered assertions. Finally, the sum of differences $diff$ is returned (line 10).

At this point, we employ the dictionary of differences $diff$ to sort the ATs in order of decreasing approximability. Since each matrix belonging to $diff$ contains 9 fields (3x3 matrix), there are several ways to estimate the impact of a fault f on the

Bit token	a[6]	a[7]	b[6]	b[7]	b[0]	b[1]	a[0]	a[1]	a[2]	b[2]	a[3]	...
ATCF	0	0	0	0	3	3	4	4	22	23	49	...
Rank	0	1	2	3	4	5	6	7	8	9	10	...
Cluster	0	0	0	0	1	1	2	2	3	3	4	...

TABLE 8.1: Final ranking of ATs for the running example.

functional behaviour of the design. In this work, we decided to rank each approximation token *at* by counting the number of times in which the corresponding fault f made the assertions in A fail. That is, each *at* is ranked by using the ATCF field (first row, second column) of the corresponding matrix $diff[ft]$. This choice is plausible because the higher the increment of ATCF, the worse the impact on the functional behaviour, and as a consequence, the lower the approximability of the corresponding AT. In particular, all the ATs are sorted by increasing order of ATCF.

Finally, we apply a clustering algorithm to group ATs exposing a similar approximability; this is done to help the designer in the process of simultaneously approximating multiple ATs. We apply a similar clustering procedure to the one described in section 5.9 involving the k-kmeans algorithm and the elbow method to determine a good candidate value of k.

Table 8.1 reports the result obtained by applying the ranking methodology to some ATs of the running example. In this case, we considered as ATs the bits of signals a and b of the statement at line 8 of Algorithm 15. As expected, the table shows that the two least significant bits of both a and b are ranked first. This is not surprising since these bits remain unused during simulation; therefore, the corresponding faults do not produce any effect on the functional behaviour of the design (ATCF is zero). Then, after the clustering, our methodology suggests that the designer should simultaneously apply the bit-width reduction strategy to a[7:6] and b[7:6].

8.4 Case Study

We evaluated the proposed approach on the RTL description of a common Sobel edge-detection filter. We applied both the bit-width and the statement reduction approximation strategies on a set of ATs (i.e., bit tokens and statement tokens) by injecting faults as reported in Section 8.3.1. We then ranked and clustered the ATs, as detailed in Section 8.3.3, by exploiting a set of assertions automatically mined as described in Section 8.3.2. We finally measured the approximation effect on the designs synthesized by approximating the different ATs in terms of functional accuracy and power/area reduction. The goal is to show the effectiveness of our approach in prompting the designer to simultaneously approximate a cluster of ATs that has a low impact on functional accuracy while guaranteeing a relevant saving in terms of area and power. We considered a total number of 236 bit tokens and 38 statement tokens.

8.4.1 Functional accuracy

For evaluating the effectiveness of the proposed approach from the point of view of the functional accuracy of the approximated designs, we adopted the Structural SIMilarity (SSIM) index [148]. Eight images have been used as the workload for the Sobel. For the bit-with reduction, we make two different experiments: the first by fixing the target bit token to 0, and the second to 1. Since the outcome is similar for both scenarios, we report the results only for the stuck-at 0 case.

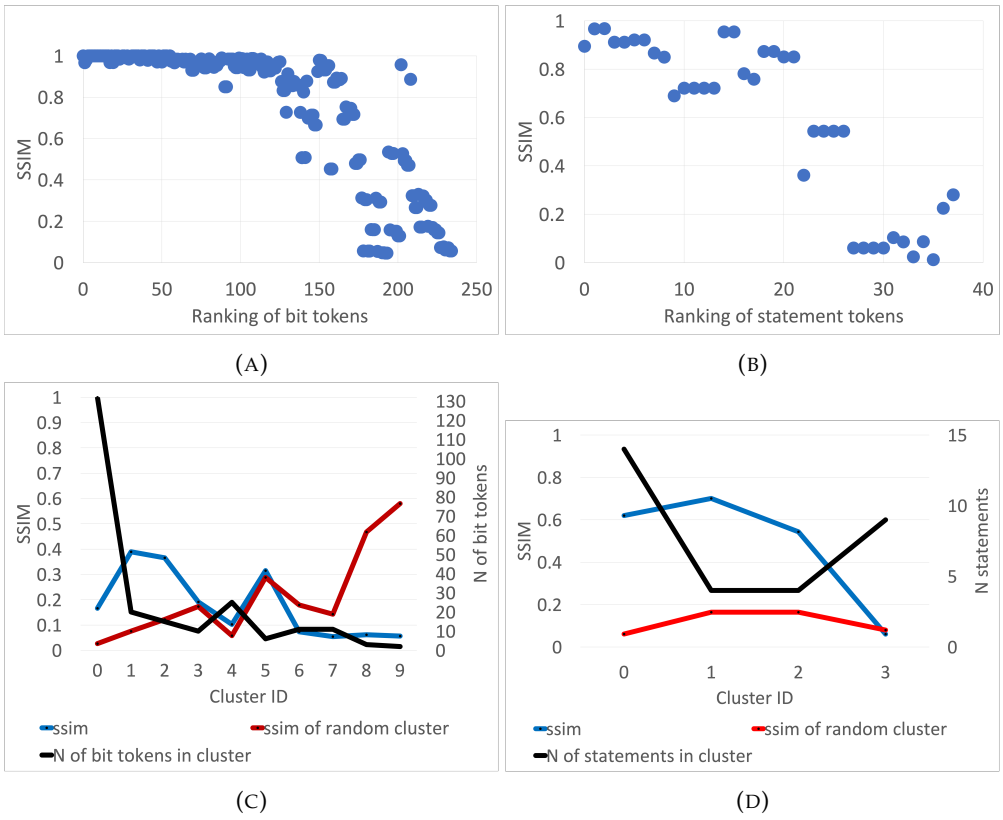

FIGURE 8.3: (A and B) Impact of AT approximation alternatives on the functionality of the Sobel. Bit tokens (A) and statement tokens (B) are ordered on the x axis according to the ranking metrics defined in Section 8.3.3.
(C and D) Impact on the functionality of the Sobel by simultaneously applying the approximations belonging to each AT cluster returned by the methodology proposed in Section 8.3.3. Clusters are ordered on the x axis from the top-ranked (Cluster 0) to the worst-ranked.

The diagrams in Figure 8.3(A)(B) show the SSIM (y axis) achieved by the designs obtained by applying the bit-width reduction (A) and the statement reduction (B) to each single AT, i.e, each bit token in (A), and each statement token in (B). The x axis refers to the ATs ranked in decreasing order according to the methodology proposed in Section 8.3.3. There is a clear trend showing that the ATs that guarantee the highest SSIM, when approximated, are those ranked first by our approach, for both bit tokens and statement tokens.

However, since approximating a single AT is not effective from the point of view of area and power saving, we propose to cluster the ATs for applying multiple approximations simultaneously. Thus, the diagrams in Figure 8.3(C)(D) present the effect of simultaneously applying the approximations belonging to the AT clusters returned by the k-means approach described at the end of Section 8.3.3. The points in the dark line refer to the size of the clusters, the points in the blue line indicate the SSIM of the clusters returned by our methodology, and the points in the red line highlight the SSIM achieved by a set of ATs randomly selected, whose size is the same of the corresponding cluster in the blue line. A first observation highlights that the random clusters achieve the worst accuracy. The second observation deals with the quality of the ranked clusters returned by our methodology. While the cluster containing the top-ranked ATs for the statement reduction strategy (Figure 8.3 (D))

shows that simultaneously applying the approximations of the top-ranked cluster (ID 0) guarantees almost the best SSIM, this is not the same for the bit-width reduction strategy (Figure 8.3 (C)), where larger clusters (e.g., ID 0) achieve an unsatisfactory SSIM, while smaller ones (e.g., IDs 1 and 2) generally perform better. This demonstrates that the SSIM for the bit-with reduction strategy is influenced by the size of the cluster, while this is not relevant for the statement reduction strategy. In addition, by analysing the composition of the clusters containing the statement tokens versus those related to the bit tokens, we observed that the former generally collect ATs belonging to the same cone of logic, while this is not true for the latter. As a consequence, the impact of the approximation on the functional accuracy is amplified when a large set of unrelated bit tokens are clustered, while this effect is mitigated while grouping statement tokens belonging to the same cone of logic.

Therefore, we conclude that the ranking and clustering procedure proposed in this paper for statement tokens is an effective methodology for guiding the designer in the exploration of the statement reduction approximation strategy in terms of functional accuracy. The next section will show that it performs very well also in terms of area/power savings.

8.4.2 Area and power saving

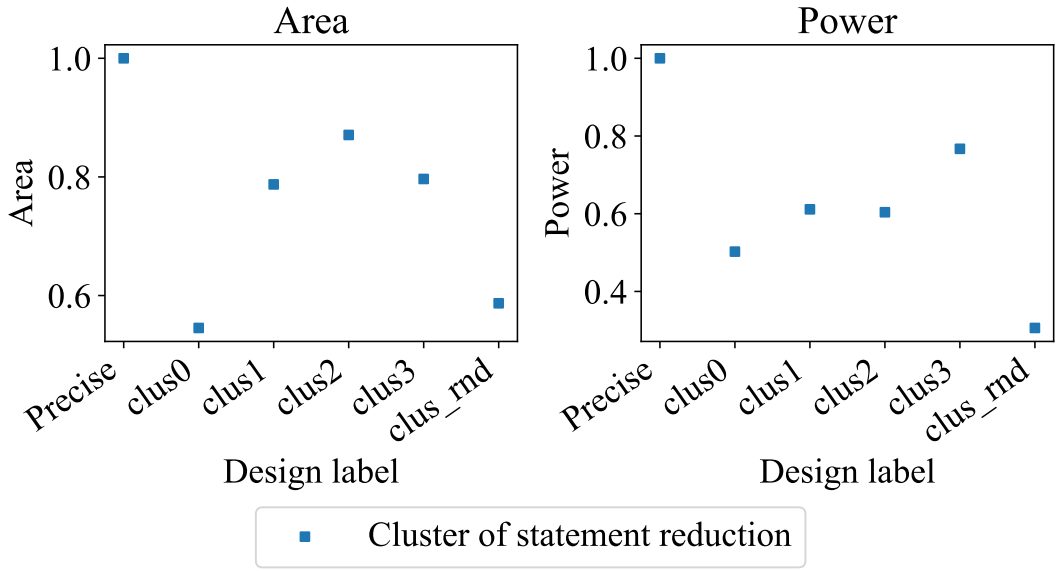

FIGURE 8.4: Saving in terms of area and power by considering the statement token clusters. *Precise* refers to the original design, *clus0*, *clus1*, *clus2* and *clus3* are related to the designs approximated according to the four clusters returned by our methodology in decreasing order of functional accuracy, *clus_rnd* indicates the average result for the design approximated by using a set of randomly chosen ATs.

We performed the synthesis of the designs obtained by approximating the statement tokens as indicated in the clusters obtained with the proposed methodology. We targeted the FreePDK45 45-nm standard cell technology library. Figure 8.4 reports the results, in terms of relative area and power consumption, calculated as $1 - \frac{Precise - clus_i}{Precise}$ (*Precise* variant has value 1). The chart highlights that the cluster with ID 0 (the best ranked by our methodology) shows the best area and power reduction w.r.t. the *Precise* design, i.e., 54.53% and 50.24% respectively, while still having a high

SSIM metric value, i.e. 0.62, as reported in Figure 8.3(D). Conversely, approximating a random set of statements with the same size as cluster 0 (i.e., 14 statements) provides a design having, on average, 58.69% area reduction and 30.60% power reduction, however having also a very lower SSIM value, i.e. 0.061. The results for the *clus_rnd* design were computed by synthesizing multiple designs obtained by approximating randomly the same number of statements as in cluster 0; finally, the average value was reported.

Chapter 9

Conclusions

In this paper, I propose a complete framework to verify complex distributed systems, from the formalisation of specifications to runtime execution. In particular, I propose 5 novel methodologies and the corresponding tools to cover several holes in the verification process of systems executing in an edge-to-cloud computing environment. Additionally, I propose a new tool called DEA that leverages assertions to help the exploration of the design.

9.1 MIST

MIST is an all-in-one tool capable of generating a complete environment to verify C/C++ firmware 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 from experienced developers allowed us to thoroughly assess the potential and limitations of MIST. The majority of limitations were overcome during the tuning process; however, there are still a 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 affects 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 test benches 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.

9.2 HARM

HARM is an efficient and flexible hint-based assertion miner. Its main characteristics include a customizable template-based procedure to mine assertions, efficient algorithms for instantiating assertion templates and evaluating if they hold on to the input trace, a 3-level parallelisation methodology that further speeds up the mining by fully exploiting the computing cores, and a context-based approach to rank the mined assertions. The experimental results show the efficiency of the tool and the quality of the generated assertions in comparison with two state-of-the-art miners. The scalability and effectiveness of HARM have been thoroughly analysed too. Finally, a concrete use case has been presented to highlight HARM's capability of ranking interesting assertions. Future works will be devoted to extending the mining capabilities of HARM. In particular, we plan to allow the generation of assertions compliant with the signal temporal logic (STL), which is a logic formalism for specifying properties of continuous signals and time. This formalism would simplify the generation of assertions for hybrid and real-time systems.

9.3 COME and BECAUSE

COME and BECAUSE are two tools to automatically remove irrelevant instructions from execution traces identifying unexpected behaviours. COME exploits semantic equivalence and alignment of sequences in order to extract from computational paths the essential instructions characterizing a vulnerability. I evaluated the efficiency and effectiveness of COME in two case studies regarding a memory protection mechanism and a safety control system, where COME was able to correctly filter out the sequences of instructions giving only the essential elements required to characterize the vulnerability. Future works will be devoted to optimizing the internal symbolic engine of COME by introducing the possibility of recognizing previously visited paths, to further increase the scalability of the approach. Moreover, we will perform the symbolic simulation directly on the firmware's code by removing the ISS. The methodology relies only on Read/Write operations, thus we realized that the ISS provides an excessive level of detail, slowing down the symbolic simulation.

BECAUSE works by performing a preliminary reduction involving a DPDG and dynamic program slicing. After that, the remaining instructions are further reduced through an instruction clusterization procedure. Experimental results show the effectiveness and scalability of the tool.

9.4 CARMINE

CARMINE is a tool to create, orchestrate, and deploy a ROS-compliant verification environment for robotic systems. The platform relies on different key contributions, from checker synthesis to containerisation and orchestration. The methodology presented an analysis of such a checker migration by means of two case studies, which confirm the platform as a comprehensive solution for applying runtime assertion-based verification to robotic systems.

9.5 DEA

DEA implements a novel solution leveraging assertion mining and fault injection to identify the elements (signal/register bits or statements) to be approximated into RTL descriptions through the bit-width and the statement reduction strategies. Approximation alternatives are implemented by fault injection. Their impact on the functional accuracy of the DUE is then estimated accordingly to a metric that evaluates the effect of the approximation on a set of assertions automatically mined from the original implementation to capture its functionality. A ranking and clustering procedure is then proposed to guide the designer in the identification of the best cluster of elements to be approximated. The experimental analysis shows the following major achievements: (1) The procedure, for both bid-with reduction and statement reduction, is able to rank the approximation alternatives such that they are decreasingly ordered with respect to their effect on functional accuracy. (2) The clustering performs very well by considering the statement reduction approximation, i.e., the functional accuracy is generally higher for the designs obtained by simultaneously applying the approximations corresponding to the top-ranked AT clusters than those synthesized by considering the lower-ranked clusters or by randomly selecting a set of statement tokens. (3) The clustering of bits is affected by the size of the cluster, which is more determinant than the ranking of its elements for mitigating the amplification of the errors caused by simultaneous bit-width reductions. (4) The saving in terms of area and power achieved by approximating the clusters of statements identified by the proposed approach is generally proportional to the functional accuracy: the higher the saving, the higher the accuracy.

Bibliography

[1] "Standard for property specification language (PSL)," *IEC 62531:2012(E) (IEEE Std 1850-2010)*, pp. 1–184, 2012.

[2] S. Germiniani, M. Bragaglio, and G. Pravadelli, "Mist: Monitor generation from informal specifications for firmware verification," in *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, 2020, pp. 111–116. DOI: 10.1109/VLSI-SOC46417.2020.9344072.

[3] S. Germiniani and G. Pravadelli, "Harm: A hint-based assertion miner," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4277–4288, 2022. DOI: 10.1109/TCAD.2022.3197525.

[4] S. Germiniani and G. Pravadelli, "Exploiting clustering and decision-tree algorithms to mine ltl assertions containing non-boolean expressions," in *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–6. DOI: 10.1109/VLSI-SoC54400.2022.9939640.

[5] S. Aldegheri, N. Bombieri, S. Germiniani, F. Moschin, and G. Pravadelli, "A containerized ros-compliant verification environment for robotic systems," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 222–225. DOI: 10.23919/DATE51398.2021.9474167.

[6] S. Germiniani, A. Danese, and G. Pravadelli, "Automatic generation of assertions for detection of firmware vulnerabilities through alignment of symbolic sequences," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 2, pp. 728–739, 2022. DOI: 10.1109/TETC.2020.3035187.

[7] M. Bragaglio, N. Donatelli, S. Germiniani, and G. Pravadelli, "System-level bug explanation through program slicing and instruction clusterization," in *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2021, pp. 1–6. DOI: 10.1109/VLSI-SoC53125.2021.9607008.

[8] A. Bosio, S. Germiniani, G. Pravadelli, and M. Traiola, "Exploiting assertions mining and fault analysis to guide rtl-level approximation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023.

[9] 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, Feb. 2004. DOI: 10.1016/S0164-1212(02)00160-7.

[10] 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," in May 2018, pp. 86–103, ISBN: 978-3-319-91601-9. DOI: 10.1007/978-3-319-91602-6_6.

[11] 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.

[12] [Online]. Available: https://www.iar.com/iar-embedded-workbench.

[13] 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.

[14] 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, 3555–3558 Vol. 4.

[15] 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.

[16] 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.

[17] "Ieee standard for property specification language (psl)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010. DOI: 10.1109/IEEESTD.2010.5446004.

[18] "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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] [Online]. Available: https://www.esa.int.

[24] 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," Oct. 2016, pp. 122–129. DOI: 10.1007/978-3-319-46520-3_8.

[25] C. Wang, F. He, X. Song, Y. Jiang, M. Gu, and J. Sun, "Assertion recommendation for formal program verification," in *Proc. of IEEE COMPSAC*, 2017, pp. 154–159.

[26] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," in *Proc. of ACM POPL*, 2002, pp. 4–16.

[27] A. Danese, V. Bertacco, and G. Pravadelli, "Symbolic assertion mining for security validation," in *Proc. of ACM/IEEE DATE*, 2018.

[28] M. D. Ernst., "Static and dynamic analysis: Synergy and duality," in *Proc. of WODA*, 2003, pp. 24–27.

[29] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertion with guidance from static analysis," *IEEE Trans. on CAD*, vol. 32, no. 6, pp. 952–965, 2013.

[30] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proc. of ICSE*, 1999, pp. 411–420.

[31] "Harm repository." (Dec. 7, 2022), [Online]. Available: `https://github.com/SamueleGerminiani/harm`.

[32] D. Lo and S. Maoz, "Specification mining of symbolic scenario-based models," in *Proc. of ACM PASTE*, 2008, pp. 29–35.

[33] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *Proc. of ACM KDD*, 2007, pp. 460–469.

[34] J. Henkel and A. Diwan, "Discovering algebraic specifications from java classes," in *Proc. of ECOOP*, 2003, pp. 431–456.

[35] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[36] M. D. Ernst, J. H. Perkins, P. J. Guo, *et al.*, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[37] A. Hekmatpour and A. Salehi, "Block-based schema-driven assertion generation for functional verification," in *Proc. of IEEE ATS*, 2005, pp. 34–39.

[38] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Proc. of ACM/IEEE DAC*, 2005, pp. 775–778.

[39] G. Chen, M. Liu, and Z. Kong, "Semantic inference for cyber-physical systems with signal temporal logic," in *Proc. of IEEE CDC*, 2019, pp. 6269–6274.

[40] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proc. of ACM/IEEE DAC*, 2010, pp. 755–760.

[41] L. Liu, C.-H. Lin, and S. Vasudevan, "Word level feature discovery to enhance quality of assertion mining," in *Proc. of IEEE ICCAD*, 2012, pp. 210–217.

[42] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *Proc. of ACM/IEEE DATE*, 2010, pp. 626–629.

[43] S. Vasudevan, D. Sheridan, and V. Athavale, "Automatic generation of assertions from system level design using data mining," in *Proc. of IEEE MEMOCODE*, 2011, pp. 191–200.

[44] K. L. McMillan, "The smv system," in *Symbolic Model Checking*. Springer, 1993, pp. 61–85.

[45] D. Pal, S. Offenberger, and S. Vasudevan, "Assertion ranking using RTL source code analysis," *IEEE Trans. on CAD*, vol. 39, no. 8, pp. 1711–1724, 2020.

[46] T. Ghasempouri and G. Pravadelli, "On the estimation of assertion interestingness," in *Proc. of IFIP/IEEE VLSI-SoC*, 2015, pp. 325–330.

[47] A. Danese, F. Filini, T. Ghasempouri, and G. Pravadelli, "Automatic generation and qualification of assertions on control signals: A time window-based approach," in *VLSI-SoC: Design for Reliability, Security, and Low Power*, Y. Shin, C. Y. Tsui, J.-J. Kim, K. Choi, and R. Reis, Eds., Springer, 2016, pp. 193–221.

[48] A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *Proc. of ACM/IEEE DATE*, 2015, pp. 67–72.

[49] M. Bonato, G. Di Guglielmo, M. Fujita, F. Fummi, and G. Pravadelli, "Dynamic property mining for embedded software," in *Proc. of ACM/IEEE CODES+ISSS*, 2012, pp. 187–196.

[50] M. Bertasi, G. Di Guglielmo, and G. Pravadelli, "Automatic generation of compact formal properties for effective error detection," in *Proc. of ACM/IEEE CODES+ISSS*, 2013, pp. 1–10.

[51] A. Danese, N. D. Riva, and G. Pravadelli, "A-team: Automatic template-based assertion miner," in *Proc. of ACM/IEEE DAC*, 2017.

[52] R. Srikant and J. F. Naughton, "Fast algorithms for mining association rules and sequential patterns," Ph.D. dissertation, 1996, ISBN: 0591211750.

[53] [Online]. Available: `http://www.atrenta.com/about-bugscope.htm5`.

[54] *Jasper Activeprop*. [Online]. Available: `http://www.jasper-da.com`.

[55] 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," in *Proc. of ATVA*, ser. Lecture Notes in Computer Science, vol. 9938, Springer, 2016, pp. 122–129.

[56] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982. DOI: `10.1109/TIT.1982.1056489`.

[57] "Goldmine designs." (Dec. 7, 2022), [Online]. Available: `https://bitbucket.org/debjitp/goldminer/src/master/example/`.

[58] "Trust hub." (Dec. 7, 2022), [Online]. Available: `https://trust-hub.org/%5C#/benchmarks/chip-level-trojan`.

[59] [Online]. Available: `https://opencores.org/`.

[60] [Online]. Available: `https://github.com/SamueleGerminiani/subplatform`.

[61] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984. DOI: `10.1109/TSE.1984.5010248`.

[62] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ser. SDE 1, New York, NY, USA: Association for Computing Machinery, 1984, pp. 177–184, ISBN: 0897911318. DOI: `10.1145/800020.808263`. [Online]. Available: `https://doi.org/10.1145/800020.808263`.

[63] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[64] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 37–61, Jan. 1985, ISSN: 0164-0925. DOI: `10.1145/2363.2366`. [Online]. Available: `https://doi.org/10.1145/2363.2366`.

[65] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '81, New York, NY, USA: Association for Computing Machinery, 1981, pp. 207–218, ISBN: 089791029X. DOI: 10.1145/567532.567555. [Online]. Available: https://doi.org/10.1145/567532.567555.

[66] T. Reps and T. Bricker, "Illustrating interference in interfering versions of programs," in *Proceedings of the 2nd International Workshop on Software Configuration Management*, ser. SCM '89, New York, NY, USA: Association for Computing Machinery, 1989, pp. 46–55, ISBN: 0897913345. DOI: 10.1145/72910.73347. [Online]. Available: https://doi.org/10.1145/72910.73347.

[67] B. Korel and J. Laski, "Dynamic slicing of computer programs," *Journal of Systems and Software*, vol. 13, no. 3, pp. 187–195, 1990, ISSN: 0164-1212.

[68] T. Chen and Y. Cheung, "Dynamic program dicing," in *Proc. of IEEE CSM*, 1993, pp. 378–385.

[69] M. Renieres and S. Reiss, "Fault localization with nearest neighbor queries," in *Proc. of IEEE ASE*, 2003, pp. 30–39. DOI: 10.1109/ASE.2003.1240292.

[70] W. E. Wong and Y. Qi, "Effective program debugging based on execution slices and inter-block data dependency," *Journal of Systems and Software*, vol. 79, no. 7, pp. 891–903, 2006.

[71] A. Groce, "Error explanation with distance metrics," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 108–122.

[72] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Model Checking Software*, T. Ball and S. K. Rajamani, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 121–136.

[73] C.Duanzhi, "A collection of program slicing," in *Proc. of ICCASM*, 2010.

[74] J. Field, G. Ramalingam, and F. Tip, "Parametric program slicing," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, New York, NY, USA: Association for Computing Machinery, 1995, pp. 379–392, ISBN: 0897916921. DOI: 10.1145/199448.199534. [Online]. Available: https://doi.org/10.1145/199448.199534.

[75] J. Field and F. Tip, "Dynamic dependence in term rewriting systems and its application to program slicing," in *Programming Language Implementation and Logic Programming*, M. Hermenegildo and J. Penjam, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 415–431, ISBN: 978-3-540-48695-4.

[76] J. Q. Ning, A. Engberts, and W. V. Kozaczynski, "Automated support for legacy code understanding," *Commun. ACM*, vol. 37, no. 5, pp. 50–57, May 1994, ISSN: 0001-0782. DOI: 10.1145/175290.175295. [Online]. Available: https://doi.org/10.1145/175290.175295.

[77] G. A. Venkatesh, "The semantic approach to program slicing," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI '91, New York, NY, USA: Association for Computing Machinery, 1991, pp. 107–119, ISBN: 0897914287. DOI: 10.1145/113445.113455. [Online]. Available: https://doi.org/10.1145/113445.113455.

[78] M. Kramkar, P. Fritzson, and N. Shahmehri, "Interprocedural dynamic slicing applied to interprocedural data flow testing," in *1993 Conference on Software Maintenance*, 1993, pp. 386–395. DOI: 10.1109/ICSM.1993.366924.

[79] J.-D. Choi, B. P. Miller, and R. H. B. Netzer, "Techniques for debugging parallel programs with flowback analysis," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 491–530, Oct. 1991, ISSN: 0164-0925. DOI: 10.1145/115372.115324. [Online]. Available: https://doi.org/10.1145/115372.115324.

[80] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006. DOI: 10.1109/TSE.2006.105.

[81] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. of ACM SIGPLAN PLDI*, 2005, pp. 15–26.

[82] W. Wen, "Software fault localization based on program slicing spectrum," in *Proc. of IEEE ICSE*, 2012, pp. 1511–1514.

[83] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of CGO*, 2004.

[84] *Gnu coreutils*. [Online]. Available: https://www.gnu.org.

[85] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution.," in *USENIX Security Symposium*, 2013, pp. 463–478.

[86] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proc. of ACM/IEEE IPSN*, Jan. 2010, pp. 186–196.

[87] P. Li and J. Regehr, "T-check: Bug finding for sensor networks," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, ACM, 2010, pp. 174–185.

[88] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[89] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.

[90] D. Song, D. Brumley, H. Yin, *et al.*, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, Springer, 2008, pp. 1–25.

[91] Y. Shoshitaishvili, R. Wang, C. Salls, *et al.*, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[92] P. Godefroid, "Random testing for security: Blackbox vs. whitebox fuzzing," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, ACM, 2007, pp. 1–1.

[93] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "Crete: A versatile binary-level concolic testing framework," in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2018, pp. 281–298.

[94] A. Danese, V. Bertacco, and G. Pravadelli, "Symbolic assertion mining for security validation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, IEEE, 2018, pp. 1550–1555.

[95] R. Baldoni, E. Coppa, D. C. Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 50:1–50:39, May 2018, ISSN: 0360-0300. DOI: 10.1145/3182657. [Online]. Available: http://doi.acm.org/10.1145/3182657.

[96] C. Kallenberg, S. Cornwell, X. Kovah, and J. Butterworth, "Setup for failure: Defeating secure boot," in *The Symposium on Security for Asia Network (SyScan)(April 2014)*, 2014.

[97] "Llvm syntax." (Dec. 7, 2022), [Online]. Available: https://llvm.org/docs/LangRef.html.

[98] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of USENIX OSDI*, 2008.

[99] M. J. Harrold, B. Malloy, and G. Rothermel, "Efficient construction of program dependence graphs," in *Proc. of ACM ISSTA*, 1993, pp. 160–170.

[100] D. Baumann, F. Mager, U. Wetzker, L. Thiele, M. Zimmerling, and S. Trimpe, "Wireless control for smart manufacturing: Recent approaches and open challenges," *Proceedings of the IEEE*, vol. 109, no. 4, pp. 441–467, 2021. DOI: 10.1109/JPROC.2020.3032633.

[101] A. Sathyamoorthy, U. Patel, M. Paul, Y. Savle, and D. Manocha, "Covid surveillance robot: Monitoring social distancing constraints in indoor scenarios," *PLoS ONE*, vol. 16, no. 12 December, 2021. DOI: 10.1371/journal.pone.0259713.

[102] X. Liu, S. Chen, G. Nardari, *et al.*, "Challenges and opportunities for autonomous micro-uavs in precision agriculture," *IEEE Micro*, vol. 42, no. 1, pp. 61–68, 2022. DOI: 10.1109/MM.2021.3134744.

[103] D. Maria, E. Sibi, S. Jerome, *et al.*, "Environment model generation and localisation of mobile indoor autonomous robots," in *ACCESS 2021 - Proceedings of 2021 2nd International Conference on Advances in Computing, Communication, Embedded and Secure Systems*, 2021, pp. 257–264. DOI: 10.1109/ACCESS51619.2021.9563306.

[104] Z. Ullah, F. Al-Turjman, U. Moatasim, L. Mostarda, and R. Gagliardi, "Uavs joint optimization problems and machine learning to improve the 5g and beyond communication," *Computer Networks*, vol. 182, 2020. DOI: 10.1016/j.comnet.2020.107478.

[105] A. T. Praveen, A. Gupta, S. Bhattacharyya, and R. Muthalagu, "Assuring behavior of multirobot autonomous systems with translation from formal verification to ros simulation," *IEEE Systems Journal*, 2022.

[106] R. Halder, J. Proença, N. Macedo, and A. Santos, "Formal verification of ros-based robotic applications using timed-automata," in *IEEE/ACM International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2017, pp. 44–50.

[107] H. Foster. Now Foundations and Trends, 2009.

[108] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez, "Braceassertion: Runtime verification of cyber-physical systems," in *Proceedings - 2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2015*, 2015, pp. 298–306. DOI: 10.1109/MASS.2015.15.

[109] H. Abbas, I. Saha, Y. Shoukry, *et al.*, "Special session: Embedded software for robotics: Challenges and future directions," 2018.

[110] H. Ko, M. Jo, and V. Leung, "Application-aware migration algorithm with prefetching in heterogeneous cloud environments," *IEEE Transactions on Cloud Computing*, 2021. DOI: 10.1109/TCC.2021.3064292.

[111] Open Source Robotics Foundation, "Robot Operating System," http://www.ros.org/.

[112] P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the mop runtime verification framework," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 249–289, 2012.

[113] E. Bartocci, J. Deshmukh, A. Donzé, *et al.*, "Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications," *Lecture Notes in Computer Science*, vol. 10457, pp. 135–175, 2018.

[114] C. Hu, W. Dong, Y. Yang, H. Shi, and G. Zhou, "Runtime verification on hierarchical properties of ros-based robot swarms," *IEEE Transactions on Reliability*, vol. 69, no. 2, pp. 674–689, 2020.

[115] K. Havelund, G. Reger, and G. Roşu, "Runtime verification past experiences and future projections," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10000, pp. 532–562, 2019.

[116] G. Rosu and K. Havelund, "Rewriting-based techniques for runtime verification," *Automated Software Engineering*, vol. 12, no. 2, pp. 151–197, 2005.

[117] D. Ulus, T. Ferrère, E. Asarin, and O. Maler, "Online timed pattern matching using derivatives," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9636, pp. 736–751, 2016.

[118] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard, "Quantified event automata: Towards expressive and efficient runtime monitors," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7436 LNCS, pp. 68–84, 2012.

[119] A. Pnueli and A. Zaks, "On the merits of temporal testers," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5000 LNCS, pp. 172–195, 2008.

[120] D. Basin, F. Klaedtke, and E. Zălinescu, "Algorithms for monitoring real-time properties," *Acta Informatica*, vol. 55, no. 4, pp. 309–338, 2018.

[121] A. Dokhanchi, B. Hoxha, and G. Fainekos, "On-line monitoring for temporal logic robustness," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8734, pp. 231–246, 2014.

[122] K. Havelund, D. Peled, and D. Ulus, "First-order temporal logic monitoring with bdds," *Formal Methods in System Design*, 2019.

[123] T. Zabinski, T. Maoczka, J. Kluska, M. Madera, and J. Sep, "Condition monitoring in industry 4.0 production systems - the idea of computational intelligence methods application," vol. 79, 2019, pp. 63–67.

[124] W. Zhang, M.-P. Jia, L. Zhu, and X.-A. Yan, "Comprehensive overview on computational intelligence techniques for machinery condition monitoring and fault diagnosis," *Chinese Journal of Mechanical Engineering (English Edition)*, vol. 30, no. 4, pp. 782–795, 2017.

[125] S. Aldegheri, N. Bombieri, S. Germiniani, F. Moschin, and G. Pravadelli, "A containerized ros-compliant verification environment for robotic systems," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 222–225. DOI: 10.23919/DATE51398.2021.9474167.

[126] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[127] NVIDIA, "Deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA Jetson," https://github.com/dusty-nv/jetson-inference.

[128] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: An open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017. DOI: 10.1109/TRO.2017.2705103.

[129] A. Bosio, D. Menard, and O. Sentieys, Eds., *Approximate computing techniques*, 1st ed. Cham, Switzerland: Springer Nature, Jun. 2022.

[130] A. Sampson, A. Baixo, B. Ransford, *et al.*, "Accept: A programmer-guided compiler framework for practical approximate computing," *University of Washington Technical Report UW-CSE-15-01*, vol. 1, no. 2, 2015.

[131] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Proc. of IEEE ETS*, 2013.

[132] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proc. of ACM/IEE DAC*, 2013.

[133] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proc. of ACM/IEE DAC*, 2015.

[134] W. Liu, F. Lombardi, and M. Shulte, "A retrospective and prospective view of approximate computing," *Proceedings of the IEEE*, vol. 108, no. 3, pp. 394–399, 2020.

[135] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, 62:1–62:33, Mar. 2016, ISSN: 0360-0300.

[136] S. Barone, M. Traiola, M. Barbareschi, and A. Bosio, "Multi-objective application-driven approximate design method," *IEEE Access*, vol. 9, pp. 86 975–86 993, 2021. DOI: 10.1109/ACCESS.2021.3087858.

[137] S. Mitra, M. Das, A. Banerjee, K. Datta, and T.-Y. Ho, "A verification guided approach for selective program transformations for approximate computing," in *Proc. of IEEE ATS*, 2016.

[138] H. Foster, D. Lacey, and A. Krolnik, *Assertion-Based Design*, 2nd ed. USA: Kluwer Academic Publishers, 2003.

[139] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi, "Phase-aware optimization in approximate computing," *Proc. of ACM/IEEE CGO*, pp. 185–196, 2017.

[140] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proc. of ACM ESEC/FSE*, 2011.

[141] K. Nepal, Y. Li, R. Bahar, and S. Reda, "Abacus: A technique for automated behavioral synthesis of approximate computing circuits," in *Proc. of ACM/IEEE DATE*, 2014.

[142] M. Barbareschi, S. Barone, A. Bosio, J. Han, and M. Traiola, "A genetic-algorithm-based approach to the design of DCT hardware accelerators," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 18, no. 3, pp. 1–25, Jul. 2022. DOI: 10.1145/3501772. [Online]. Available: https://doi.org/10.1145/3501772.

[143] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate arithmetic circuits: A survey, characterization, and recent applications," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, 2020.

[144] W. Liu, T. Cao, P. Yin, *et al.*, "Design and analysis of approximate redundant binary multipliers," *IEEE Transactions on Computers*, vol. 68, no. 6, pp. 804–819, 2018.

[145] V. Mrazek, L. Sekanina, and Z. Vasicek, "Libraries of approximate circuits: Automated design and application in cnn accelerators," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 4, pp. 406–418, 2020. DOI: 10.1109/JETCAS.2020.3032495.

[146] X. F. M. Ye and S. Wei, "Runtime hardware security verification using approximate computing: A case study on video motion detection," in *Proc. of IEEE AsianHOST*, 2019.

[147] Y. M. K. Yoshisue and T. Ishihara, "Dynamic verification of approximate computing circuits using coverage-based grey-box fuzzing," in *Proc. of IEEE IOLTS*, 2021.

[148] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.