Alessandro Danese

System-level functional and extra-functional characterization of SoCs through assertion mining

Ph.D. Thesis

January 14, 2018

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
Prof. Graziano Pravadelli

Università degli Studi di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

Summary. Virtual prototyping is today an essential technology for modeling, verification, and re-design of full HW/SW platforms. This allows a fast prototyping of platforms with a higher and higher complexity, which precludes traditional verification approaches based on the static analysis of the source code. Consequently, several technologies based on the analysis of simulation traces have proposed to efficiently validate the entire system from both the functional and extra-functional point of view.

From the functional point of view, different approaches based on invariant and assertion mining have been proposed in literature to validate the functionality of a system under verification (SUV). Dynamic mining of invariants is a class of approaches to extract logic formulas with the purpose of expressing stable conditions in the behavior of the SUV. The mined formulas represent likely invariants for the SUV, which certainly hold on the considered traces. A large set of representative execution traces must be analyzed to increase the probability that mined invariants are generally true. However, this is extremely time-consuming for current sequential approaches when long execution traces and large set of SUV's variables are considered. Dynamic mining of assertions is instead a class of approaches to extract temporal logic formulas with the purpose of expressing temporal relations among the variables of a SUV. However, in most cases, existing tools can only mine assertions compliant with a limited set of pre-defined templates. Furthermore, they tend to generate a huge amount of assertions, while they still lack an effective way to measure their coverage in terms of design behaviors. Moreover, the security vulnerability of a firmware running on a HW/SW platforms is becoming ever more critical in the functional verification of a SUV. Current approaches in literature focus only on raising an error as soon as an assertion monitoring the SUV fails. No approach was proposed to investigate the issue that this set of assertions could be incomplete and that different, unusual behaviors could remain not investigated.

From the extra-functional point of view of a SUV, several approaches based on power state machines (PSMs) have been proposed for modeling and simulating the power consumption of an IP at system-level. However, while they focus on the use of PSMs as the underlying formalism for implementing dynamic power management techniques of a SoC, they generally do not deal with the basic problem of how to generate a PSM.

In this context, the thesis aims at exploiting dynamic assertion mining to improve the current approaches for the characterization of functional and extra-functional properties of a SoC with the final goal of providing an efficient and effective system-level virtual prototyping environment. In detail, the presented methodologies focus on: efficient extraction of invariants from execution traces by exploiting GP-GPU architectures; extraction of human-readable temporal assertions by combining user-defined assertion templates, data mining and coverage analysis; generation of assertions pinpointing the unlike execution paths of a firmware to guide the analysis of the security vulnerabilities of a SoC; and last but not least, automatic generation of PSMs for the extra-functional characterization of the SoC.

Contents

Part I Preliminary

1 Introduction 9
1.1 Objectives of the thesis 11

2 Background 13

Part II Functional Verification

3 Invariant Mining 17
3.1 Introduction 17
3.2 State of the Art 18
3.3 Objectives 19
3.4 Background 19
3.5 Methodology 20
3.5.1 Turbo 21
3.5.2 Mangrove 31
3.6 Conclusions 36

4 Assertion Mining 37
4.1 Introduction 37
4.2 State of the Art 38
4.3 Objectives 39
4.4 Methodology 39
4.4.1 Oden 40
4.4.2 A-TEAM 52

5 Vulnerability Detection 65
5.1 Introduction 65
5.2 State of the Art 66
5.3 Objectives 66
5.4 Background 66
5.5 Methodology 67

5.5.1 DOVE 67

Part III Extra-Functional Verification

6 Power State Machine 83
6.1 Introduction 83
6.2 State of the Art 83
6.3 Objectives 84
6.4 Methodology 85
6.4.1 PsmGen 85

Part IV Conclusions

7 Conclusions and Future works 99

8 Published contributions 101

References 103

Part I

Preliminary

1

Introduction

Virtual prototyping is today an essential technology for modeling, verification and re-design of full HW/SW platforms. With respect to the serialized approach, where the majority of SW is developed and verified after the completion of the silicon design, with the risk of failing aggressive time-to-market requests, virtual prototyping guarantees a faster development process by implementing the software part almost in parallel with the hardware design (Fig 1.1). This enables software engineers to start implementation months before the hardware platform is complete, and HW designers to explore different solutions concerning functional and extra-functional (e.g., power behavior) aspects. The core of virtual prototyping is represented by the virtual system prototype, i.e., an electronic system level (ESL) software simulator of the entire system, used first at the architectural level and then as a executable golden reference model throughout the design cycle. Virtual prototyping brings several benefits like, for example, efficient management of design complexity, decoupling of SW development from the availability of the actual HW implementation, and control of prototyping costs. More pragmatically, it enables developers to accurately and efficiently explore different solutions with the aim of balancing design functionality, flexibility, performance, power consumption, quality, ergonomics, schedule and cost.

A fundamental aspect of the virtual prototyping is its functional [12] and extra-functional [9] verification. In particular, functional verification aims to answer questions such as: “is my system correctly implemented? Have I satisfied all the initial

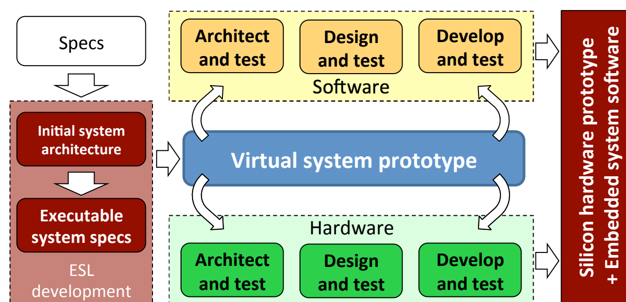


Fig. 1.1. The virtual prototyping approach.

requirements?” A way to answer these questions is through assertions [19, 52, 79] and invariant mining [32, 39]. These approaches try to infer the behavior of the system under verification (SUV) through the analysis of simulation traces generated with test cases. However, the current approaches for assertions mining based on simulation traces still have several limitations:

- they tend to generate too complex assertions (i.e., composed by large expressions involving several variables together in the same formula);
- they capture too specific situations related to the peculiarity of the actual values stressed by the test cases rather than to the real functionality implemented in the design;
- they are a very time consuming approach when hundreds of variables have to be considered in execution traces with millions of simulation instants;
- an analysis of the mined assertions is impractical without a ranking procedure.

Moreover, the security vulnerability [66, 47] of the software running on the platform is becoming ever more critical in the context of virtual prototyping. In this case, verification engineers have to verify the integration between hardware components of the system, and the software controlling the entire platform and providing the final user with an interface for the entire system. The typical way to verify such of intergeneration is to manually write down a set of assertions pinpointing the behaviors that must not occur and simulate the entire system. However, the current approaches focus only on raising an error as soon as one of these manual assertions is violated. No methodology was proposed in the literature to investigate the issue that this set of assertions could be incomplete and that different, unusual behaviors could remain not investigated.

Last but not least, we have to deal with aspects related to the extra-functional verification of the virtual prototyping, and in particular, with the power consumption. In this case, extra-functional verification wants instead to give an answer to the question: “What will the power consumption of the system be like?” The past has shown that modeling and estimating the energy consumption through state machines with adaption to the environment is a promising concept. However, they focus on the use of Power State Machines (PSM) as the underlying formalism for implementing dynamic power management techniques and they generally do not deal with the basic problem of generating PSMs. Despite of the wide adoption of PSMs, in the most of the works either the presence of PSMs is assumed or they are manually defined starting from a more or less precise knowledge of the functional blocks composing the target design [38]. Only in a few cases, automatic approaches are proposed to create the association between PSM states and their power consumptions [9, 57], but the identification of such states remains a manual effort.

In the context of functional and extra-functional verification of embedded systems, my research activity aims at making automatic the extraction of functional and extra-functional properties that characterize the behaviors of a design. In particular, the proposed methodologies in this thesis rely on the concept of assertion mining to automatically extract these aspects from the simulation traces of the design implementation.

1.1 Objectives of the thesis

This thesis proposes the methodologies reported in Figure 1.2 to improve the characterization of functional and extra-functional properties of a SoC. In detail, the proposed methodology focuses on:

1. Efficient extraction of invariants, namely arithmetic/logic relations characterizing stable conditions among the variables of a design. In particular, this research activity concerns the use of a GP-GPU based parallelizable approach to make more efficient the extraction of invariants through the analysis of execution traces and that greatly reduces the execution time with respect to existing techniques without affecting the accuracy of the analysis.
2. Extraction of temporal assertions, namely temporal-logic formulas describing how the behavior implemented by the design evolves during the execution time. In particular, with respect to the current approaches in literature, this second activity aims at extracting a more compact set of human readable temporal assertions from execution traces.
3. Automatic detection of firmware security vulnerabilities. This activity aims to propose an innovative methodology for the automatic detection of security vulnerabilities of firmwares by performing assertion mining on symbolic traces generated through concolic testing of a firmware under verification.
4. The fourth activity concerns the fully-automatic generation of Power State Machines by adopting an approach that maps, through a calibration process, the functionalities implemented by the design with its corresponding power consumption.

The rest of this thesis is organized as follows:
Part I introduces some preliminary definitions used in this thesis (Chapter 2). Part II exposes the approaches developed to improve the functional verification of a SoC and, in detail, it is divided in three different topics: Chapter 3 and Chapter 4 are respectively focus on invariant mining and assertion mining through the analysis of simulation traces. The automatic detection of firmware security vulnerability is shown in Chapter 5. Part III introduces an approach to improve the extra-functional verification of a SoC and, in particular, it focuses on the automatic generation of power state machines describing the power consumption of an IP (Chapter 6). Finally, Part IV draws some conclusions and introduces future extensions of the presented approaches while reporting the publications developed during this thesis in Chapter 7.

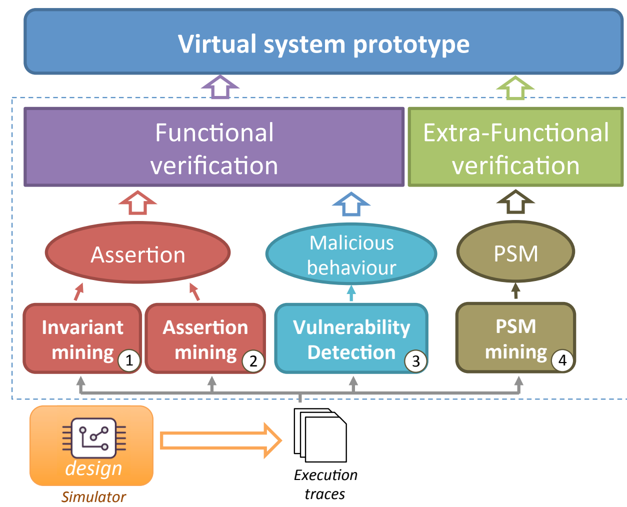


Fig. 1.2. Overview of the research activities.

2

Background

The following definitions are necessary to formalize the methodologies proposed in the rest of the thesis.

Definition 2.1. *Given a finite sequence of simulation instants* $\langle t_1, ..., t_n \rangle$ *and a model* M *working on a set of variables* V*, an* ***execution trace*** *of* M *is a finite sequence* $T = \langle V_1, ..., V_n \rangle$ *generated by simulating* M*, where* $V_i = eval(V, t_i)$ *is the evaluation of variables in* V *at simulation instant* t_i*.*

Definition 2.2. *An* ***atomic proposition*** *is a formula that does not contain logical connectives.*

In the rest of the thesis, we will consider atomic propositions on Boolean variables, like $b = True$ and $b = False$, and between numeric or bit vector data types, like v op u, where op is one of the followings: $=, <, \leq, >, \geq, \neq$.

Definition 2.3. *A* ***proposition*** *is a composition of atomic propositions through logic connectives. An atomic proposition itself is a proposition.*

In this thesis, I consider the connectives $\vee$ and $\wedge$ to compose propositions.

Definition 2.4. *Given a finite set of atomic propositions* AP*, the set of* ***Linear Time Logic*** *(LTL) formulas over* AP *can be defined, in negation normal form, as follows:*

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

The semantics of temporal operators X (*next*), U (*until*) and R (*release*) 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.

In the rest of the thesis, a composition of n X operators $XX \ldots X(a)$ is abbreviated in $X[n](a)$; a formula of the kind $\neg a \vee b$ is represented by using the logical implication $a \rightarrow b$; and, finally, $G(a)$ is used as a shortcoming for $false\ release\ a$.

Definition 2.5. *A* ***temporal assertion*** *is a composition of propositions through temporal operators*

Part II

Functional Verification

3

Invariant Mining

3.1 Introduction

Automatic invariant detection is a widely adopted strategy to analyze several aspects in verification of both SW programs and HW designs. Without forgetting the importance of invariants for documentation purposes, invariant inference has been used, for example, for test generation [22], analysis of dynamic memory consumption [16], static checking [63], detection of race conditions [69], identification of memory access violations [41], generic bug catching [74].

Independently from its use, an invariant is a logic formula that holds between a couple (or several couples) of points, A and B, of an implementation, thus expressing a stable condition in the behavior of the system under verification (SUV) for all its executions. Possibly, A and B may correspond, respectively, to the beginning and the end of the SUV execution. Different kinds of invariants, like, for example $(x \leq y)$, $(ax + b = y)$, $(x \neq NULL)$, can be inferred by either static or dynamic analysis of the SUV.

Static approaches, like [37, 76], are exhaustive and work well for relatively small/medium-size implementations. In this category we can find tools such as Axiom Meister [77]. The strongest point of all these approaches is the exhaustiveness and correctness of the result. Thanks to the formal analysis, we know that the extracted invariants will be never falsified by the implemented program. Owing to the complexity of this analysis, these approaches cannot be applied on very big and complex programs since they will take too much time to exhaustively analyze their behaviors for every input.

An alternative to static approaches is represented by dynamic invariant mining [33] (Fig 3.1). In this case, invariants are extracted by analyzing a finite set of execution traces obtained from the simulation of the SUV. Dynamic inference works even if the source code is not available and it scales better for large SUVs. Indeed, the efficiency of dynamic miners is more related to the length of analyzed execution traces and the number of observed variables than the complexity of the SUV.

As a drawback, these techniques, being not exhaustive, can extract only *likely invariants*, i.e., properties that are only statistically true during the simulation of the SUV. Then, to increase the degree of confidence on invariants mined by dynamic approaches, a large (and representative) set of execution traces must be analyzed

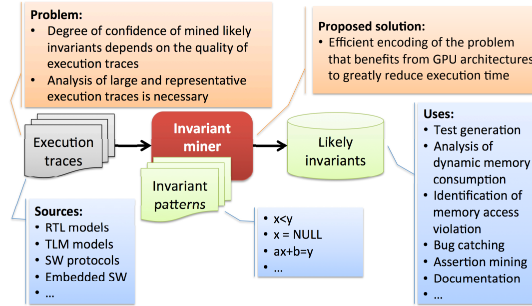


Fig. 3.1. Dynamic mining of likely invariants.

However, for complex HW designs this could require to analyze thousands of execution traces, including millions of clock cycles, and predicating over hundreds of variables, which becomes an unmanageable time-consuming activity for existing approaches. Similar considerations apply also for execution traces derived from embedded SW applications.

3.2 State of the Art

To the best of my knowledge the most effective and flexible miner of likely invariants is Daikon [33]. It analyses execution traces through an inference engine that incrementally detects invariants according to templates specified in a grammar configuration file. To extract invariants on specific points of a program, code instrumentation is required. Daikon has been mainly used for documentation, debugging, testing and maintainability of SW programs. A brief overview of Daikon's functionalities is reported at the end of the current section.

Several mining tools alternative to Daikon, and the relative uses, are referenced in [24]. In this section, I refer to some of them in representation of different categories (commercial vs. academic, hardware vs. software-oriented). For example, from a commercial point of view, Daikon inspired the creation of Agitator [60] that dynamically extracts invariants to check their compliance with respect to manually defined conditions. An alternative academic approach is implemented in DIDUCE [74]. It aids programmers to identify root causes of errors on Java programs. DIDUCE's engine dynamically formulates strict invariant hypotheses obeyed by the program at the beginning, then it gradually relaxes such hypotheses when violations are detected to include new behaviours. Finally, in the HW domain, IODINE [40] infers likely invariants for HW design descriptions. Inferred invariants refer to state-machine protocols, request-acknowledge pairs, and mutual exclusion between signals.

Contrary to the approach proposed in the current thesis, previous approaches require the instrumentation of program points which can be done only when the source code of the SUV is available. Moreover, they cannot take advantage of massive parallel execution on GPUs, thus they scale badly for large sets of long execution traces.

Daikon

Daikon analyses the execution traces through an inference engine that incrementally detects likely invariants according to a list of templates specified in a configuration file. The execution traces are generally obtained by running an instrumented target program that reports the values of several program points. Usually, the most used program points on which Daikon infers invariants are global variables and input/output arguments of methods. The internal engine of Daikon can be represented as a hierarchy of classes. Each of them implements a checker for a specific arithmetic/logic pattern between variables. Several variables' domains are currently supported, e.g., Daikon can extract likely invariants for Boolean, numeric, string and vector variables. The main idea behind the incremental invariant-inference engine of Daikon can be summarized in three steps: 1) instantiate a candidate invariant (i.e., a class) for each selected template given a combination of variables; 2) remove the candidate invariants contradicted by a sample of the trace; and 3) report the invariants that remain after processing all the samples, and after applying post-processing filtering. In order to efficiently extract invariants many optimizations have been implemented in Daikon. The most relevant of them are:

- If two or more variables are always equal, then any invariant that can be verified for one of those variables is also verified for each of the other variables.
- A dynamically constant variable is one that has the same value at each observed sample. The invariant $x = a$ (for constant a) makes any other invariant over (only) x redundant.
- Suppression of invariants logically implied by some set of other invariants is adopted. For example, $x > y$ implies $x \geq y$, and $0 < x < y$ and $z = 0$ imply $xdivy = z$.

With respect to the approach proposed in Daikon, in this thesis I do not need code instrumentation, and I encode information on candidate invariants by means of a vector-based data structure, which is more efficient and particularly suited for parallel computing, as proved by experimental results. On the contrary, the inference engine of Daikon cannot be easily ported on a GPU. To the best of my knowledge this is the first implementation of an invariant miner that runs on a GPU.

3.3 Objectives

To overcome the scalability issue affecting the current state-of-the art approaches for mining likely invariants, I present the invariant miners $Turbo$ and $Mangrove$ in this thesis. By exploiting GPU architectures, both the proposed approaches can greatly reduce the execution time for invariant mining with respect to existing techniques, without affecting the accuracy of the analysis.

3.4 Background

This section reports preliminary definitions that are necessary to understand the proposed approaches. Then, it presents a brief overview of the Graphic Process Unit (GPU) architecture to create the necessary background for describing the parallel algorithm of the proposed mining methodologies.

Preliminary definitions

Definition 3.1. *Given an execution trace T (def. 2.1), and two simulation instants t_i and t_j such that $1 \leq t_i \leq t_j \leq n$, a* ***time window*** *$TW_{i,j} = \langle (V_i, t_i), \ldots, (V_j, t_j) \rangle$ is a subsequence of contiguous elements of T.*

Definition 3.2. *Given a set of variables V of a model $\mathcal{M}$ and an execution trace T (def. 2.1), a* ***trace invariant*** *(T-invariant) is a logic formula over V that is true for each simulation instant in T.*

Definition 3.3. *Given a set of variables V of a model $\mathcal{M}$, an execution trace T (def. 2.1), and a time window $TW_{i,j} \subseteq T$, a* ***time window invariant*** *(TW-invariant) is a logic formula over V that is true for each simulation instant in $TW_{i,j}$.*

GPU architecture

GPUs are multi-core coprocessors originally intended to speed-up computer graphics. However, their highly-parallel structure makes GPUs powerful devices also for the elaboration of general-purpose computing-intensive processes that work in parallel on large blocks of data. This approach is commonly known as general-purpose computing on graphics processing units (GPGPU). The affirmation of GPGPU was further supported by the definition of ad hoc parallel computing platforms and programming models, like CUDA [23] and OpenCL [64].

Figure 3.2 shows the internal architecture of common GPUs. A GPU is composed of various (streaming) multiprocessors, each one consisting of several processing cores that execute in parallel a sequence of instructions, commonly known as *kernel-function*. Multiple program threads organized in *blocks* are distributed and concurrently executed by the cores of each multiprocessor. Inside a multiprocessor, data are elaborated in SIMD (single instruction, multiple data) mode. As a consequence, threads running on the same core that need to perform instructions on different branches of a conditional statement are executed sequentially. This issue is known as "divergence" and it is one of the most important cause of performance degradation in GPGPU. In this computational platform, there are four types of memories, namely *shared*, *constant*, *texture* and *global* memory. All of them, but shared memory, are freely accessible by an external CPU, which is used to submit kernels to the GPU. The shared memory is very fast and it is available only for threads belonging to the same block for data sharing.

3.5 Methodology

In the next section, the methodologies *Turno* and *Mangrove* are exposed. In the first approach (*Turno*), a sequential algorithm, and a parallel algorithm for GPUs are presented to manage the mining of invariants on execution traces composed of millions of simulation instants and tens of variables in a few seconds. The two algorithms can work on execution traces from both HW and SW domains. Moreover, when an exhaustive mining of invariants on different time windows, belonging to the same execution trace, is required (for example, for extracting invariants to be used in mining of temporal assertions [25]), the parallel version can analyze hundreds of thousands of sub-traces on the order of minutes.

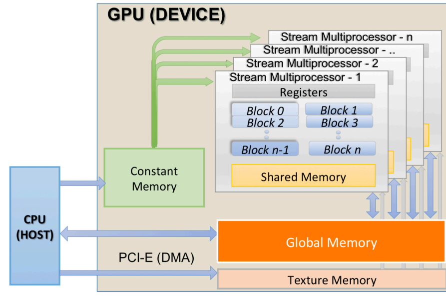


Fig. 3.2. GPU architecture.

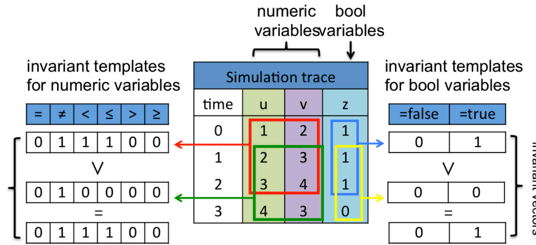


Fig. 3.3. Use of the invariant vector to check invariants on different time windows.

The second approach, namely *Mangrove*, is an alternative to the parallel version of *Turbo*. *Mangrove* is particularly addressed in mining trace invariants, namely invariants always holding on the entire analyzed simulation trace. As reported in the experimental results, the overall performance of the mining algorithm are increased up to three orders of magnitude with respect to *Turbo*.

3.5.1 Turbo

Methodology

This section presents a dynamic parallelizable approach for mining both trace invariants and time window invariants. Indeed, a T-invariant is a TW-invariant for a time window that extends from the first to the last simulation instant of the corresponding trace. Thus, to avoid burdening the discussion, in the following, I use the term invariant when concepts apply indistinctly to T-invariants and TW-invariants. I first propose a sequential algorithm (Section *Sequential algorithm*) to dynamically infer invariants. Afterwards, I will illustrate the changes that I made to implement an its parallel version running on a GPU (Section *Parallel algorithm*).

Given a set of variables V of a model $\mathcal{M}$, both the sequential and parallel algorithms rely on a bit vector-based data structure, called *invariant vector*, to efficiently represent logic relations among the variables in V. Without lack of generality, let TW be a time window, and $I = \{inv_1, inv_2, \ldots, inv_n\}$ be a list of n invariant templates representing logic relations among the set of variables V. We can track if a m-ary logic relation corresponding to the invariant $inv_i \in I$, instantiated with

a tuple[1] of variables $(v_1, \ldots, v_m) \in V^m$, holds in TW by using an invariant vector, *inv_result*, composed of n elements. Element i of *inv_result* corresponds to the instance $inv_i(v_1, \ldots, v_m)$ of the invariant template inv_i referred to the tuple $(v_1, \ldots, v_m)$. Thus, $inv_result[i]$ is 0 if $inv_i(v_1, \ldots, v_m)$ is false at least once in TW; it is 1 otherwise.

Given a set of execution traces and a set of different time windows, this invariant vector allows us to rapidly analyze the following conditions, for all instances of the invariant templates:

C1: $inv_i(v_1, \ldots, v_m)$ is true for at least one time window of one execution trace. This is necessary, for example, to prove that there exists at least one execution run that brings the model to a stable condition where $inv_i(v_1, \ldots, v_m)$ remains true for a given time interval.

C2: $inv_i(v_1, \ldots, v_m)$ is true for at least one time window of all the considered execution traces. This shows a stable condition occurs, where $inv_i(v_1, \ldots, v_m)$ is true, for a given time interval at least once per each execution run of the model.

C3: $inv_i(v_1, \ldots, v_m)$ is true for at least one execution trace. This can prove that there exist at least one run of the model where the condition $inv_i(v_1, \ldots, v_m)$ remains always stable for the entire duration of the execution run.

C4: $inv_i(v_1, \ldots, v_m)$ is true for all the analysed execution traces. This statistically proves $inv_i(v_1, \ldots, v_m)$ holds always each time the model is executed, assuming that the analysed traces are statistically representative of all the model's behaviors.

For example, in Figure 3.3, the use of the invariant vector is reported for a simple execution trace involving two numeric variables (u and v) and one Boolean variable (z). Two time windows of length 3 are highlighted, related, respectively, to the time intervals [0,2] and [1,3]. The six logic relations on the left and the two on the right are used as invariant templates, respectively, for the numeric and the Boolean variables. By considering only numeric variables (same considerations apply for the Boolean variable), in the first time window, the invariant templates $u \neq v$ and $u < v$ are true (red box), thus the corresponding invariant vector is $\{0, 1, 1, 1, 0, 0\}$. Meanwhile, in the second time window only the invariant template $u \neq v$ is true (green box), thus the corresponding invariant vector is $\{0, 1, 0, 0, 0, 0\}$. As a consequence, a global invariant vector for the numeric variables, for example to check condition C1, is obtained by applying a bitwise OR among the invariant vectors of each time window. Condition C2 is checked by a bitwise AND among the global invariant vectors of different execution traces. Finally, C3 and C4 are similarly obtained by analyzing the whole execution traces without time-window partitioning.

Sequential algorithm

In the current implementation, my algorithm can infer binary logic relations represented by the following invariant templates

- $\{(u = v), (u \neq v), (u < v), (u \leq v), (u > v), (u \geq v)\}$ for a pair of numeric variables (u, v);
- $\{(v = true), (v = false)\}$ for a Boolean variable v.

[1] The arity of the tuple depends on the arity of the invariant.

However, the approach is independent from the specific template, thus it can be easily extended to consider further kinds of arithmetic logic relations between two or more variables and constants, like, for example, being in a range ($a \leq v \leq b$) and linear relationships ($v = au + b$).

The sequential approach follows the strategy implemented in Algorithm 1. Given a set V of variables, an execution trace T and an integer $l > 0$, it extracts all formulas that hold on at least one time window of length l included in T. This thesis is intended to present the mining algorithm and its optimization for parallel computing, while no consideration is reported on the choice of the length of the time windows. Indeed, the selection of the value for the parameter l depends on the desired kind of verification. For example, by varying the parameter l, different time window intervals can be analyzed to check conditions of kind C1. On the contrary, if l is set to the size of T, the algorithm computes invariants holding on the whole execution trace, thus providing results for analyzing conditions of kind C3. Finally, calling the algorithm on several execution traces, the existence of invariants satisfying conditions C2 and C4 can be analyzed too.

Assuming the presence of two sets of invariant templates: I_{Bool} for Boolean variables and I_{Num} for numeric variables, the algorithm starts by invoking the function $invariantChecker$, which calls $getBoolInv$ and $getNumInv$, respectively, on each Boolean variable $u \in V$ and on each pair of numeric variables $(u, v) \in V \times V$.

The execution flow of $getBoolInv$ and $getNumInv$ is practically the same. They first initialize elements of the invariant vector inv_result to 0 (lines 17 and 37). In the current implementation, I have 6 invariant templates for numeric variables and 2 for Boolean variables, as described at the beginning of this section. At the end of the algorithm execution, $inv_result[i]$ is 1 if the corresponding invariant inv_i holds at least on one time window of T. Then, $getBoolInv$ and $getNumInv$ iterate the following steps for each time window of length l belonging to the execution trace T (lines 18-32 and 38-54):

1. Before starting the analysis of a new time window, another invariant vector ($local_res$) of the same length of inv_result is initialized to 1 (lines 19 and 39). At the end of the time window analysis, $local_res[i]$ is 1 if no counterexample has been found for the corresponding invariant inv_i.
2. During the analysis of a time window, $local_res[i]$ is set to 0 as soon as a counter example is found within the time window for invariant inv_i (lines 22-23 and 43-45).
3. At the end of the time window analysis, inv_result is updated according to the value of $local_result$ (lines 28 and 50). If $local_result$ is 1 then also inv_result becomes 1 to store that the algorithm found a time windows where inv_i holds.

The number of checks performed by the algorithm (i.e., lines 22 and 23 for the Boolean variables, and line 44 for the numeric variables), in the worst case, depends on:

- the number of variables' pairs to be checked (i.e., $|V|^2$, for the considered 2-ary invariants);
- the length of the time-window (i.e., l), and consequently the number of time windows in the execution trace (i.e., $(length(T) - l + 1)$); and
- the total length of the execution trace (e.g., $length(T)$).

Thus, according with the previous considerations, the algorithm scales best in $|T|$ for very small time windows or those close to $|T|$. To reduce the overall execution

Algorithm 1 Sequential Algorithm

```
1: function INVARIANT_CHECKER(T, l, V)
2:     for all u ∈ V do
3:         if getType(u) == BOOL then
4:             print(GETBOOLINV(T, l, u));
5:         end if
6:         if getType(u) == NUMERIC then
7:             for all v ∈ V ∧ u ≠ v do
8:                 if getType(v) == NUMERIC then
9:                     print(GETNUMINV(T, l, u, v));
10:                end if
11:            end for
12:        end if
13:    end for
14: end function
15:
16: function GETBOOLINV(T, l, u)
17:    inv_result[2] = {0} ;
18:    for t = 0; t <getSize(T)−l + 1; t = t + 1 do
19:        local_res[2] = {1};
20:        for s = 0; s < l; s = s + 1 do
21:            u_val=getValue(T, t + s, u);
22:            local_res[0]=local_res[0]∧(u_val==false);
23:            local_res[1]=local_res[1]∧(u_val==true);
24:            if allZero(local_res) then //optimization 1
25:                break;
26:            end if
27:        end for
28:        inv_result = inv_result ∨ local_res;
29:        if allOne(inv_result) then //optimization 2
30:            break;
31:        end if
32:    end for
33:    return inv_result;
34: end function
35:
36: function GETNUMINV(T, l, u, v)
37:    inv_result [6] = {0} ;
38:    for t = 0; t <getSize(T)−l + 1; t = t + 1 do
39:        local_res [6] = {1};
40:        for s = 0; s < l; s = s + 1 do
41:            u_val=getValue(T, t + s, u);
42:            v_val=getValue(T, t + s, v);
43:            for i = 0; i <getSize(I_Num); i = i + 1 do
44:                local_res[i]=local_res[i]∧check(inv_i, u, v);
45:            end for
46:            if allZero(local_res) then //optimization 1
47:                break;
48:            end if
49:        end for
50:        inv_result = inv_result ∨ local_res
51:        if allOne(inv_result) then //optimization 2
52:            break;
53:        end if
54:    end for
55:    return inv_result;
56: end function
```

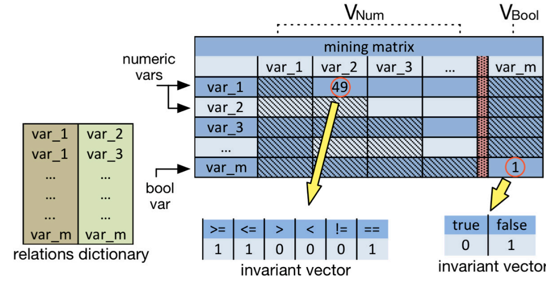


Fig. 3.4. A relation dictionary and the corresponding mining matrix. Shaded elements are not used. The diagonal is used for Boolean variables.

time, two optimizations were introduced. The first concerns the checking of invariants within a single time window TW. The iteration on all the simulation instants of TW exits as soon as a counter example for each invariant has been found (lines 24-25 and 46-47). This optimization generally reduces execution time in case of very long time windows that expose very few invariants. The second optimization concerns the checking of invariants by considering all time windows of the same execution trace T. The iteration on all the time windows of T exits as soon as all invariants have been verified on at least one time window (lines 29-30 and 51-52). This optimization reduces execution time in case of very long execution traces with several time windows and high probability of finding time windows where the candidate invariants hold.

Parallel algorithm

According to the considerations reported at the end of the previous section, the sequential algorithm is efficient when the length of the execution trace T is low and the corresponding time windows are either short or almost as long as T. On the contrary, the worst cases occur for a very long execution trace T with time windows whose length is near to the half of the length of T. To preserve efficiency even in cases where invariant mining becomes unmanageable by using the sequential algorithm, I defined also a parallel version that can be executed by a GPU.

The parallel algorithm works on a *mining matrix* M of $|V| * |V|$ unsigned integers. The set of variables V of a model is partitioned in two subsets V_{Bool} and V_{Num} that contain, respectively, Boolean and numeric variables. The binary representation of the unsigned integer stored in each element $M[i][j]$, with $i \neq j$, corresponds to the *invariant vector* of the pair $(v_i, v_j) \in V_{Num} \times V_{Num}$. Each element on the diagonal $M[k][k]$ corresponds to the invariant vector of a Boolean variable $v_k \in V_{Bool}$. Elements $M[i][i]$ with $v_i \in V_{Num}$ and $M[i][j]$ with either $v_i \in V_{Bool}$ or $v_j \in V_{Bool}$ are not used. Elements $M[j][i]$ below the diagonal are not used too, since they would contain the dual of the invariant vector stored in $M[i][j]$. In summary, $\big(|V_{Bool}| + (|V_{Num}| * (|V_{Num}| - 1))\,/2\big)$ elements of M are active during the execution of the algorithm. A list of the variable pairs corresponding to the active elements of the mining matrix is stored in a *relation dictionary*. Figure 3.4 shows an example of a relation dictionary and the corresponding mining matrix.

The mining matrix can be generalized to an n-dimensional array to mine logic relations with arity till n. For example, to mine unary, binary and ternary templates,

a three-dimensional array (a cube) should be used to represent all the possible combination of three variables. In this case, unary relations will be stored in the element $M[i][i][i]$ (diagonal of the cube), binary relations will use only the faces of the cube, and ternary relations also internal elements. For simplicity and without loss of generality, in the following I consider only unary relations on Boolean variables and binary relations on numeric variables.

The execution flow of the parallel approach can be summarized in three steps:

1. create and copy the mining matrix and the relation dictionary into the global memory of the GPU;
2. run a parallel kernel in the GPU to extract invariant;
3. read the mining matrix from the global memory and print the results.

In order to achieve a better performance, two different kernel implementations were defined for the step 2: one for mining T-invariant (*check_T-invariants*) according to conditions C3 and C4 defined at the beginning of this section, and one for mining TW-invariants (*check_TW-invariants*) according to conditions C1 and C2.

Mining of T-invariants: The *check_T-invariants* kernel searches for invariants that are true in every simulation instant of an execution trace. The kernel takes advantage of the efficient allocation of threads in the GPU. The idea behind the approach is:

- to instantiate in the GPU as many *thread blocks* as the number of entries of the relation dictionary (i.e., each block works on a different entry of the relation dictionary), and
- to instantiate for each block the maximum number of available threads (e.g., 1024 threads in case of the GPU I used for experimental results).

Every thread of a block checks in parallel to the other threads of the same block if each of the considered invariant templates is true for the target entry of the relation dictionary in a precise simulation instant t of the execution trace (i.e., each thread works in parallel on different simulation instants). The approach to verify if an invariant template holds on a pair of variables is exactly the same implemented in functions $getBoolInv$ and $getNumInv$ of the sequential algorithm previously introduced. After checking, each thread updates the corresponding invariant vector into the $mining_matrix$ (the elements of the matrix are at the beginning initialized with 1). In particular, the thread that works on pair (v_i, v_j) for a simulation instant t stores the result in element $M[i][j]$ by means of an *AtomicAnd* operation, which is executed sequentially with respect to other *AtomicAnd* performed by different threads that work on the same pair (v_i, v_j) but on different simulation instants. In this way, when all threads complete the kernel execution, the number stored in $M[i][j]$ represents the final invariant vector of (v_i, v_j) over the execution trace. The same considerations apply for elements of kind $M[k][k]$ related to each Boolean variable v_k.

Moreover, to increase the efficiency of the parallel approach, the following optimizations have been implemented:

- The execution trace is partitioned in *slices* which are asynchronously loaded into the GPU global memory. To achieve better performance I used different streams (i.e., *cudaStream*) to asynchronously load and elaborate different slices of the execution trace.

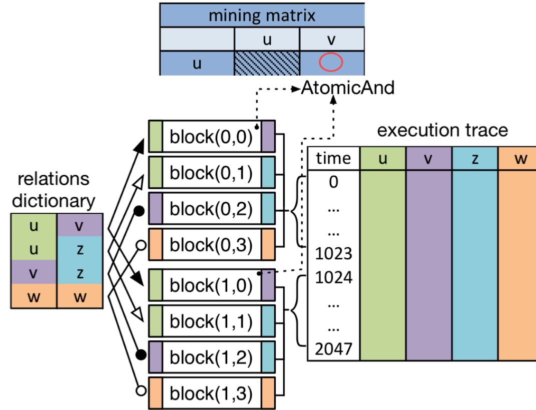


Fig. 3.5. Allocation of thread blocks. Block(i, j) works on dictionary entry j by analyzing slice i of the execution trace.

- If the threads of a block falsify all invariant templates for an entry of the relation dictionary in one slice of the execution trace, they do not check the same invariant templates in the subsequent slices of the same execution trace. This does not create divergence on threads because all the threads of a block deal with the same entry of the relation dictionary.

Figure 3.5 graphically shows how the threads of different blocks can work in parallel, on different entries of the relation dictionary and different time intervals, to speed-up the invariant checking. For example, block(0,0) works on simulation instants belonging to the interval [0, 1023] for the entry (u, v), while block (0,1) works on the same interval but for the entry (u, z), and block(1,0) works on the same entry (u, v) of block (0,1) but on the interval [1024, 2047].

Mining of TW-invariants: The *check_TW-invariants* kernel searches for invariants that are true in at least one time window of an execution trace. The idea behind the approach is basically the same as for the $check_T - invaraint$ kernel, i.e., to assign an entry of the relation dictionary to every block of threads. However, two aspects differentiate *check_T-invariants* from *check_TW-invariants*:

- each thread of the same block checks if invariant templates are true on a *different time window* of the same execution trace (not on a different time instant);
- the thread that works on the entry (v_i, v_j) of the relation dictionary for a given time window stores the result in element $M[i][j]$ of the mining matrix (the elements of the matrix are at the beginning initialized with 0) by means of an *AtomicOr* operation. This guarantees that at the end of the procedure, each element of the invariant vector stored in $M[i][j]$ is set to 1 if the corresponding invariant template has been satisfied by at least one time window.

Furthermore, to increase the efficiency of the parallel approach, the following optimizations have been implemented:

- Since all threads of the same block analyze the same entry of the relation dictionary on overlapping time windows, the currently-analyzed slice of the execution

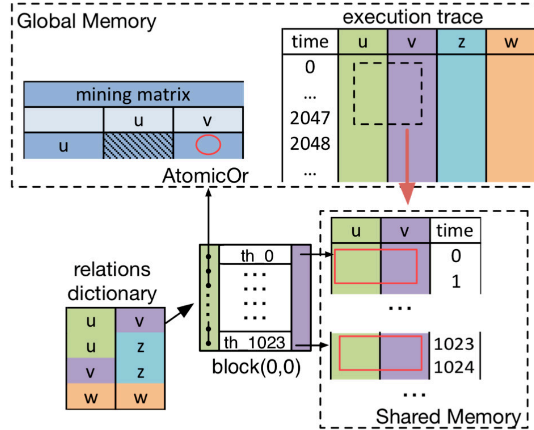


Fig. 3.6. Use of the shared memory to speed up mining of TW-invariants.

trace is copied in the GPU shared memory. This greatly reduces the time required for retrieving the value of analyzed variables, since the latency of the shared memory is really lower than the latency of the GPU global memory. For example, Fig. 3.6 shows how a block of 1024 threads works to check invariant templates for the dictionary entry (u, v). First, values assumed by u and v on a slice of the execution trace (e.g., simulation instants in the interval [0, 2047]) are copied into the shared memory. Then, all threads of the block check the invariant templates on different time windows with the same length. Each time window starts one simulation instant later than the precedent time window. If a time window exceeds the slice, new data are shifted from the execution trace into the shared memory and the verification process is resumed. When all time windows have been analysed, every thread stores its local result into the mining matrix through an *AtomicOr*.

- In case the currently-analyzed time window exceeds the slice of the execution trace loaded in the shared memory, as soon as all invariant templates have been falsified, the block of threads stops to check the same invariant templates on the following slices. This does not create divergence on threads because all the threads of a block deal with the same entry of the relation dictionary.

Experimental results

The sequential and the parallel approaches have been evaluated in comparison with Daikon version 5.2.0. For a fair comparison, Daikon has been configured such that it searched only for the same invariant templates implemented in my algorithms. This restriction does not affect the fairness of the comparison. In fact, the inclusion of a larger set of invariant templates would have the same effect on my algorithms as well as on Daikon, i.e., the verification time would increase proportionally with the number of invariant templates to be checked. The extension to the full set of Daikon's template is an ongoing activity.

For all experiments, my approaches and Daikon extracted the same set of invariants. Thus, from the accuracy point of view they are equivalent, while they

Trace length	Numeric variables	Boolean variables	Invariants number	Daikon time (s.)	Sequential time (s.)	Parallel time (s.)
1000000	15	15	0	27.3	2.8	3.2
3000000	15	15	0	74.4	8.5	9.1
5000000	15	15	0	118.3	13.9	14.9
1000000	10	10	0	21.6	2.0	2.5
1000000	30	30	0	47.6	5.6	6.4
1000000	50	50	0	73.9	9.1	9.3
1000000	15	15	120	20.3	6.5	3.3
3000000	15	15	120	51.6	19.7	8.8
5000000	15	15	120	82.3	32.8	14.8
1000000	10	10	55	15.4	2.9	2.3
1000000	30	30	465	35.2	25.6	6.1
1000000	50	50	1275	58.5	80.7	10.5

Table 3.1. Execution time (in seconds) to mine T-invariants from execution traces *with* and *without* invariants at varying of the trace length and the variable number.

Time window length	Numeric variables	Boolean variables	Invariants number	Daikon time (s.)	Sequential time (s.)	Parallel time (s.)
100000	50	50	0	$\approx 141 \times 10^5$	2378.9	39.5
500000	50	50	0	$\approx 209 \times 10^5$	1324.8	26.1
900000	50	50	0	$\approx 69 \times 10^5$	272.1	12.9
5	50	50	1275	$\approx 42 \times 10^5$	128.4	10.9
25	50	50	1275	$\approx 43 \times 10^5$	312.2	11.1
100000	50	50	1275	$\approx 326 \times 10^5$	$\approx 147 \times 10^4$	2887.8
500000	50	50	1275	$\approx 778 \times 10^5$	$\approx 832 \times 10^4$	8075.7
900000	50	50	1275	$\approx 273 \times 10^5$	$\approx 333 \times 10^4$	2949.6

Table 3.2. Execution time (in seconds) to mine TW-invariants from an one-million-long execution trace (violet rows refer to the best cases where time windows are short; the red row refers to the worst case where the length of the time windows is half of the trace).

differ from the execution time point of view. Performances have been evaluated on execution traces with different characteristics by running experiments on an AMD Phenom II X6 1055T (3GHz) host processor equipped with 8.0GB of RAM, running Linux OS, and connected to an NVIDIA GEFORCE GTX 780 with CUDA Toolkit 5.0. Results are reported for mining both T-invariants, covering conditions C3 and C4 as well as TW-invariants, covering conditions C1 and C2.

Execution time for mining T-invariants

The type of SUV (i.e., HW design or SW program), and the complexity of the SUV (in terms, for example, of memory elements, lines of code, cyclomatic complexity) are not particularly relevant to measure the performance of approaches for dynamic

invariant mining. The analysis of a long execution trace exposing several invariants among variables, even if corresponding to a functionally simple SUV, may require much more time than a shorter execution trace of a very complex SUV. Indeed, execution time of invariant mining depends on the number and length of the execution traces to be analyzed, the number of considered variables, and the number of invariants actually present in the traces. Thus, experimental results have been conducted on randomly generated execution traces with different values for such parameters by considering boolean and numeric (integer and real) data-type variables.

Table 3.1 reports the time[2] spent by the three approaches (Daikon, sequential algorithm and parallel algorithm) to analyze execution traces from which no invariant (above the central double line) and several invariants (below the central double line) can be mined (see Column *Invs*). For traces without invariants (but I observed the same behavior in case of very few invariants), my sequential and parallel approaches present similar execution times, which are one order of magnitude lower than Daikon's time. The speed-up achieved by the parallel algorithm thanks to the use of the GPU, is compensated in the sequential algorithm by optimization 1 (lines 24 and 46 of Algorithm 1), which allows the sequential algorithm to discard the entire execution trace as soon as all invariant templates have been falsified.The parallel algorithm, on the other hand, partially benefits from this optimization, since it must elaborate at least an entire slice of the execution trace. When the number of invariants that can be mined in the trace increases, the effect of optimization 1 decreases, thus the parallel algorithm becomes the most efficient solution thanks to its capability of analysing in parallel several instants of the execution trace.

Execution time for mining TW-invariants

The second experiment shows the performance of the two proposed approaches compared to Daikon for mining TW-invariants on at least one time window of an execution trace. This analysis is more time consuming than mining T-invariants on the whole execution trace, since a huge number of partially overlapping time windows must be iteratively analyzed. This is necessary, for example, for temporal assertion miners, where invariants extracted from different time windows are composed of means of temporal operators to create temporal assertions that hold on the whole execution trace [25].

Table 3.2 shows the results at varying time window lengths, by considering an execution trace with one million instants. When the length of time windows is low (violet rows), the sequential and the parallel algorithms require, respectively, few minutes and few seconds to complete the analysis, while Daikon is up to five orders of magnitude slower. For long time windows (hundreds of thousand of simulation instants), the parallel approach is two orders of magnitude faster than the sequential algorithm and three than Daikon. The worst case, as expected, occurs when the length of the time windows is half of the execution trace and the number of invariants is high (red row). It actually takes a couple of hours with the parallel algorithm, while it would take about 3 months with the sequential algorithms and 6 months

[2] Reported execution times include also the time required to read the execution trace and print the list of mined invariants. This time is not negligible and it is practically the same for the three approaches. Its removal would further amplify the difference among the scalability of the approaches.

with Daikon. Indeed, execution times reported for Daikon, and part of those of the sequential algorithm (highlighted by symbol $\approx$) have been estimated according to values achieved on shorter execution traces, because it would be unmanageable to run real experiments. The parallel algorithm, on the other hand, scales very efficiently also in these cases.

3.5.2 Mangrove

Methodology

In this section, I propose an alternative, which is called $Mangrove$[3], to the parallel algorithm of the methodology $Turbo$. $Mangrove$ even more greatly benefits from advanced graphics processing unit (GPU) programming techniques, such that the memory throughput of the GPU is significantly improved. As reported in the experimental results, the overall performance of this invariant miner algorithm are increased up to three orders of magnitude with respect to $Turbo$. The drawback of $Mangreove$ is that only trace invariants (def 3.2) can be extracted from an execution trace T.

The main mining function, in its sequential form, is reported in Algorithm 2. The inputs of the function are represented by an execution trace T of the SUV, an invariant template set $\mathcal{I}$, and a variable dictionary $\mathcal{D}$. The dictionary contains tuples of different arity composed by all the possible combinations of the variables $\mathcal{V}$ of the SUV. Such tuples represent the actual parameters to be substituted inside the formal parameters of the invariant templates during the mining phase.

[3] From the shape of the mangrove roots that resemble several parallel computation flows, in opposition with the Daikon [33] radish which is unique.

Algorithm 2 The invariant mining algorithm.

```
 1: function sequential_mining(D, I, T)
 2:     for all TUPLE ∈ D do
 3:         template_set = I
 4:         for all INSTANT ∈ T do
 5:             for all INV ∈ template_set do
 6:                 if ¬check_invariant(INV,TUPLE,INSTANT) then
 7:                     template_set = template_set \ INV
 8:                 end if
 9:             end for
10:             if template_set = ∅ then
11:                 break
12:             end if
13:         end for
14:         result = result ∪ ⟨TUPLE, template_set⟩
15:     end for
16: end function
```

The algorithm extracts all likely invariants for T that correspond to logic formulas included in $\mathcal{I}$, by substituting in the elements of $\mathcal{I}$ all the possible tuples of $\mathcal{V}$ belonging to $\mathcal{D}$, according to the respective arity. More precisely, the *check_invariant* function (line 5) checks if a specific template INV, instantiated with the current tuple of variables TUPLE, holds at simulation time INSTANT. When a counterexample is found for INV, it is removed from the template set (line 6) for the current tuple of variables. If all elements of the template set are falsified (line 8), the algorithm restarts by considering the next tuple in the dictionary, by skipping the remaining simulation instants of T. At the end, the algorithm collects all the pairs composed by the the survived templates and the corresponding tuples of the variable dictionary (line 11). The instantiation of the tuples in the survived templates represent the final set of likely invariants for T. The current implementation supports the invariant template sets reported in Table 3.3. Boolean and numeric templates include, respectively, only Boolean variables and numeric variables.

The proposed algorithm has a worst-case time complexity equal to $\mathcal{O}(|\mathcal{V}|^K \cdot |\tau| \cdot |\mathcal{I}|)$, where $\mathcal{V}$ is the number of considered variables, K is the arity of the invariant template belonging to $\mathcal{I}$ with the highest arity, $|\tau|$ is the number of simulation instants in the execution trace τ, and $|\mathcal{I}|$ is the number of invariant templates included in $\mathcal{I}$.

The parallel implementation for GPUs

The mining approach reported in Algorithm 2 is well suited for parallel computation. In fact, the problem can be easily decomposed in many independent tasks, each one having regular structure and fairly balanced workload. It implements the mining algorithm with the aim of exploiting the massive parallelism of GPUs and, at the same time, an inference strategy to reduce redundant checking of invariants. In addition, since reading the execution trace from the mass storage and moving it to the GPU device for the mining phase is computationally time consuming, *Mangrove* implements a strategy to overlap data transfers and mining phase.

Fig. 3.7 shows an overview of the GPU kernel organization and, in particular, how thread blocks are mapped to the trace variables for the reading and mining phases. Consider a matrix representing the execution trace, in which the columns represent the execution time instants and the rows hold the variable values (for the sake of clarity, the execution trace of the example in Fig. 3.7 consists of three variables). Each thread block is mapped to (i.e., it performs the reading and mining phase

	BOOLEAN			NUMERIC		
	UNARY	BINARY	TERNARY	UNARY	BINARY	TERNARY
TEMPLATE SET I	true, false				$=, \neq, <, >, \leq, \geq$	
TEMPLATE SET II	true, false	$=, \neq$	$Var_1 = Var_2 \text{AND} Var_3$ $Var_1 = Var_2 \text{OR} Var_3$ $Var_1 = Var_2 \text{XOR} Var_3$	$Var = Const$ $Var \neq Const$ $Var < Const$ $Var \leq Const$	$Var_1 = Var_2$ $Var_1 \leq Var_2$ $Var_1 < \sqrt{Var_2}$ $Var_1 = \log Var_2$ $Var_1 < Var_2 + 1$ $Var_1 = Var_2 * 2$	$Var_1 = {Var_2}^{Var_3}$ $Var_1 = \min(Var_2, Var_3)$ $Var_1 = \max(Var_2, Var_3)$ $Var_1 < Var_2 * Var_3$ $Var_1 \leq Var_2 + Var_3$

Table 3.3. *Template sets considered by the miner.*

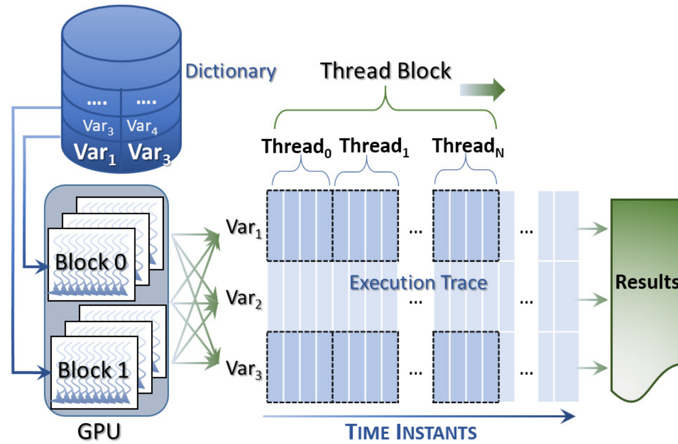


Fig. 3.7. Overview of block mapping and vectorized accesses for the parallel algorithm on GPU.

on) one, two, or three matrix rows if the mining process performs over a unary, binary, or ternary template, respectively (i.e., each thread block is mapped to a different entry of the variable dictionary). In each block, threads communicate and synchronize through shared memory. As for the standard characteristics of the GPU architectures, such hardware-implemented operations are extremely fast and their overhead is negligible. Communication and synchronization among block threads allow avoiding redundant checking of already falsified invariants and stopping the computation of the whole block as soon as all invariants for a particular set of variables have been falsified.

For each block, each thread is mapped to four columns. This allows enabling the *vectorized* accesses [59] of threads to the variable values in memory. In particular, each thread loads four consecutive 32-bit words instead of a single word to improve the memory bandwidth between DRAM and thread registers. In addition, the Boolean and numeric variables included in the variable dictionary are organized over bit and float arrays in *row-major order*. This allows the full coalescing of memory accesses by the GPU threads in the mining phase.

Mangrove computes the mining process by elaborating, in sequence, the unary templates, the binary templates, and, finally, the ternary templates reported in Table 3.3.

Optimization of the variable dictionary

The variable dictionary consists of a data structure that initially stores, for each invariant template, all the tuples of variables that must be substituted as actual parameters in the template during the mining phase. However, at run time, *Mangrove* implements some optimizations in the variable dictionary, to increase the efficiency of the mining. In particular, *Mangrove* optimizes the variable dictionary by discarding a tuple for a template when the answer of the relative checking phase can be derived from the results obtained from previous iterations of the mining procedure.

This allows saving time by avoiding redundant elaborations, as explained in the next paragraphs:

- The result of the mining over unary templates is exploited during the mining of binary templates. As a simple example, *Mangrove* searches for any Boolean variable, var_a, whose value is always equal (or always different) to any other Boolean variable, var_b. If such a condition occurs, the generation of the entry $< var_a, var_b >$ in the dictionary can be avoided since it is redundant.
- The result of the mining over unary and binary templates is used during the mining of ternary templates. For example, by considering the ternary mining phase on Boolean variables, the goal is to figure out which operator $op \in \{$AND, OR, XOR$\}$ can be validated over three different variables (e.g., var_a, var_b, and var_c). Through the already extracted unary and binary invariants, *Mangrove* automatically infers some ternary invariants without applying the checking procedure throughout the execution traces. For instance, the ternary invariant ($var_a = var_b$ AND var_c) reduces to check whether the binary invariant ($var_a = var_b$) occurs when ($var_b = var_c$) holds. Similarly ($var_a = var_b$ XOR var_c) reduces to check ($var_a \neq var_c$) when var_b is constantly set to *true*.

Data transfer and overlapping of the mining phase

The invariant mining process on the GPU consists of three main phases showed in Fig. 3.8(a): (i) reading of the execution trace from the mass storage (disk) and data storing in the host DRAM memory; (ii) data transfer from the host to the memory of the GPU; (iii) elaboration in the GPU device. The three steps work first on the numeric variables and then they are repeated for the Boolean variables. The time spent for such three phases and, in particular, the percentage of time spent by each phase over the total execution time depends on the invariant template and on the hardware characteristics. For instance, considering a magnetic mass storage disk, the reading and data transfer phases spend around the 80% and 20% over the total time, respectively while the mining time is negligible for unary templates. The three phases spend around 70%, 20%, and 10%, respectively, over the total time for binary templates. The percentage of the reading phase sensibly decreases in case of solid-state disks (SSDs).

Mangrove implements the invariant mining by overlapping the three phases as shown in Fig. 3.8(b). This allows totally hiding the cost of host-device data transfers and partially hiding the cost of the mining elaboration. Moreover, *Mangrove* implements the data transfer overlapping through asynchronous kernel invocations and memory copies (i.e., `cudaMemcpyAsync` in CUDA). Finally, a specific optimization has been implemented for Boolean variables: *Mangrove* stores the values of Boolean variables in arrays of bits to reduce the memory occupation (e.g., 5,000,000 values of a Boolean variable are stored in 600 KB). In addition, this array-based representation allows using bitwise operations to concurrently elaborate 32 Boolean values in a single chunk, thus speeding up the mining phase.

Experimental results

Experimental results have been run on a NVIDIA Kepler GeForce GTX 780 device with 5 GHz PCI Express 2.0 x16, CUDA Toolkit 7.0, AMD Phenom II X6 1055T

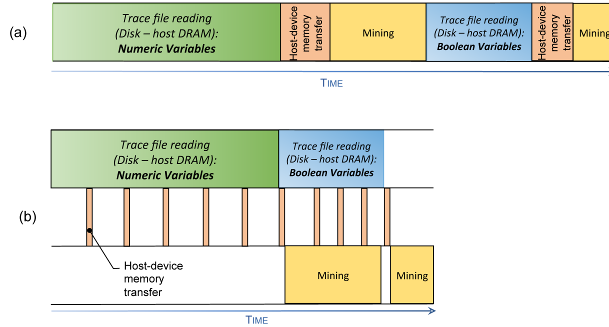


Fig. 3.8. The invariant mining phases (a), and the overlapped implementation of *Mangrove* for GPUs (b).

	Length	Boolean Vars	Numeric Vars	Invariants (Temp. set I)	Invariants (Temp. set II)
TRACE 1	5,000,000	15	15	0	0
TRACE 2	5,000,000	15	15	142	964
TRACE 3	5,000,000	50	50	0	0
TRACE 4	5,000,000	50	50	1,788	42,371

Table 3.4. *Characteristics of execution traces.*

		Daikon[33]	Sequential *Turbo*	Parallel *Turbo*	Mangrove
Template Set I	TRACE 1	103 s	< 1 ms	116 ms	< 1 ms
	TRACE 2	170 s	4,629 ms	116 ms	17 ms
	TRACE 3	287 s	2 ms	369 ms	< 1 ms
	TRACE 4	1366 s	52,160 ms	457 ms	182 ms
Template Set II	TRACE 1	2 m 34 s	22 ms	352 ms	< 1 ms
	TRACE 2	5 m 47 s	11 m 0 s	1,751 ms	140 ms
	TRACE 3	8 m 23 s	119 ms	3,145 ms	< 1 ms
	TRACE 4	32 m 54 s	7 h 45 m	71,314 ms	4,577 ms

Table 3.5. Comparison of the execution times with respect to state-of-the-art approaches.

3GHz host processor, and the Debian 7 Operating System. To evaluate the efficiency of *Mangrove* experiments have been conducted on different kinds of execution traces, whose characteristics are summarized in Table 3.4. Traces have been synthetically generated such that they expose from no invariant to thousands of likely invariants by considering the template sets reported in Table 3.3. They also differ in terms of number of considered SUV variables. These are the two parameters that most influence, together with the length of the trace, the execution time of the mining algorithm. Indeed, higher is the number of likely invariants exposed by the execution traces, higher is the time spent for their extraction.

The efficiency of *Mangrove* has been compared against the sequential mining approaches implemented, respectively, in [33] and in *Turbo*, and the parallel implementation proposed in *Turbo*. Table 3.5 shows the execution time required to extract the likely invariants according the first and second template sets on the traces reported in Table 3.3. For the parallel approaches, the times include the overhead introduced for data transfer between host and device. *Mangrove* provides the best results in all datasets by executing up to four orders of magnitude faster than the sequential state-of-the-art tool Daikon[4]. Compared to *Turbo*, *Mangrove* executes up to three orders of magnitude faster[5]. The improvements achieved in *Mangrove* with respect to the parallel approach implemented in *Turbo* are due to the implementation of a more efficient strategy for mapping thread blocks to entries of the variable dictionary, and to the vectorized accesses that best exploit the memory coalescence and the high memory throughput. These aspects are critical to improve the performance, since the memory bandwidth may limit the concurrent memory accesses. Table 3.5 shows that *Mangrove* is efficient also when no invariant can be mined (Traces 1 and 3) thanks to the capability of early terminating the search on a trace as soon as all templates have been falsified. On the contrary, the parallel implementation proposed in *Turbo* always requires to analyze the whole trace to identify the absence of likely invariants, thus wasting time.

3.6 Conclusions

In this thesis, I presented *Turbo* and *Mangrove*, two parallel approaches for mining likely invariants by exploiting GPU architectures. Both the approaches greatly reduce the execution time with respect to existing techniques, without affecting the accuracy of the analysis. Moreover, advanced GPU-oriented optimizations and inference techniques have been implemented in *Mangrove* such that execution traces composed of millions of clock cycles can be generally analyzed in less than one second searching for thousands of likely invariants. Experimental results have been conducted on execution traces with different characteristics, and the proposed approaches have been compared with sequential and parallel implementations of the most promising state-of-the-art invariant miners. Analysis of the results showed that *Turbo* and *Mangrove* outperforms existing tools for mining time-window and trace invariants.

[4] For a fair comparison, Daikon has been configured to search only for the invariants specified in the first and second template sets.

[5] The approach in *Turbo* has been extended in order to support also the template set II.

4

Assertion Mining

4.1 Introduction

Assertion-based verification is a common approach to check the functional correctness of a design model against its formal specifications. The desired behaviors are expressed through temporal assertions, i.e., logics formulas written by means of a temporal logic like, for example, Linear Time Logic (LTL) or Computation Tree Logic (CTL). Then, they are verified either statically by model checking, or dynamically by synthesizing assertion checkers that verify if the corresponding assertions are true or not during the simulation of the Design Under Verification (DUV). Unfortunately, assertion definition is a time-consuming and error-prone task, which requires high expertise to reason in terms of logic formulas. A low-quality set of assertions negatively affects the verification process. Thus, after their definition, assertions must be analyses in terms of *consistency* and *coverage* with respect to both the informal specifications and the actual implementation of the DUV, i.e., the verification engineer must guarantee that (i) assertions express expected behaviors in the correct way, (ii) all expected behaviors are expressed by the assertions, and (iii) the DUV implements all and only the expected behaviors.

An orthogonal and complementary approach to the manual definition of assertions is represented by assertion mining [4]. Assertions can be mined either statically, by analyzing the DUV source code, or dynamically, by focusing only on the analysis of the execution traces of the DUV. Static and dynamic analyses present complementary advantages and disadvantages concerning accuracy and scalability [31]. Even if assertions mined by dynamic approaches are guaranteed to be true only for the considered execution traces, dynamic mining is gaining more and more consensus, because it is more scalable, and it can be applied also in the case the source code of the DUV is not available. Figure 4.1 describes the general idea about dynamic assertion mining. The DUV model can be described at different abstraction levels targeting, for example, register transfer level (RTL) or transaction level model (TLM) hardware descriptions as well as software protocols and embedded software. Execution traces, generated by simulating the DUV, pass through an assertion miner tool, whose output is a set of candidate assertions that capture the behaviors exercised during simulation. The verification engineer then compares the mined assertions against the initial specifications to verify if all expected behaviors have been imple-

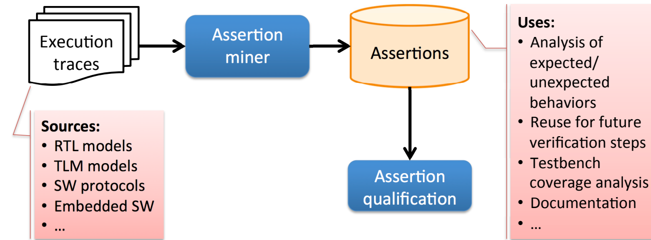


Fig. 4.1. Assertion mining overview.

mented in the DUV and potentially discover the presence of design errors. Finally, mined assertions can be used for documentation purposes too.

As a drawback, dynamic mining extracts *likely* assertions, which are guarantee to be true only for the considered execution traces, thus a formal checking of mined assertions against the DUV is generally performed to confirm they globally hold on the DUV. Moreover, behaviors not exposed by the considered execution traces cannot be mined. This shortcoming is in common with all dynamic-based verification approaches, and it is addressed by using coverage metrics to evaluate the quality of the considered execution traces.

4.2 State of the Art

Different approaches have been proposed in the literature for dynamic assertion mining of hardware designs. Seshia et al. proposed a gate-level approach that extracts assertions compliant with a predefined set of temporal templates [52].

Approaches that work at more abstracted levels have been proposed by Vasudevan et al. in [42] and [55], respectively, for RTL and TLM. The tool Goldmine, proposed in [42], exploits a decision tree-based algorithm that predicates on Boolean variables, and it generates LTL assertions in the form of implications, where only the *next* temporal operator can be included. The approach in [55] mines assertions in the form $always(a \rightarrow F_{[t1,t2]}(b))$, meaning that the TLM event b must occur from a minimum of $t1$ to a maximum of $t2$ instants after each occurrence of the TLM event a. The only approach that generates temporal properties considering arithmetic/logic expression was proposed in [15]. It is based on a cubic-complexity algorithm that relies on the Daikon invariant miner and on the generation of accepting automata to extract candidate assertions. However, the generated properties are hard-to-read, since these always involve all primary inputs and outputs of the design. Moreover, it does not define a ranking function to evaluate their quality.

Commercial tools are also available for automatic assertion generation at RTL, e.g., Atrenta BugScope [17] and Jasper ActiveProp [44]. The first generate SVA or PSL assertions where only the next temporal operator is considered. The second generates both structural and behavioral SVA next-based assertions, but no arithmetic/logic expressions are considered.

All of these existing approaches suffer of two drawbacks: (i) they rely on a predefined set of templates, thus they are unable to extract generic assertions, and

(ii) they tend to generate a long list of over-constrained assertions, which makes impossible their practical analysis by a human without a ranking strategy.

4.3 Objectives

To assist the manual definition of temporal assertions describing the functionalities implemented by an IP, I presented the assertion miners *ODEN* and *A-TEAM* in this thesis. Both miners work with atomic propositions, which allow them to be applied in hardware and software domain. Moreover, *A-TEAM* includes a fault-based coverage analysis to evaluate and compact the final set of mined temporal assertions.

4.4 Methodology

In the next section, the methodologies *ODEN* and *A-TEAM* are exposed. In particular, the main characteristics of *ODEN* with respect to the proposed approaches [15, 13] are:

- implementation of techniques to improve the quality of the mined assertions relying on (i) analysis of the cones of influence of the DUV, (ii) more efficient and accurate extraction and composition of atomic propositions, and (iii) classification of candidate propositions to be included in the final formulas to avoid the generation of temporal assertions depending on the testbench peculiarity rather than the DUV functionality. All these techniques pursue the goal of mining neither "too simple" nor "too complex" assertions to avoid drawbacks suffered by [15, 13];
- extension of the temporal patterns considered for assertion mining;
- optimization of the overall execution time.

A-TEAM extends the methodology previously defined in *ODEN*. The distinguishing features of *A-TEAM* are:

- Mined assertions are compliant with user-defined temporal patterns. Thus the tool is flexible, and not limited to a predefined set of templates;
- Mined assertions are less prone to be over-constrained with respect to the outcome of existing approaches, since the tool instantiates a few number of atomic propositions within each assertion. This makes the assertions more readable. Moreover, in this form, each assertion covers more easily an entire behavior of the DUV rather than a specific computational path (which corresponds to a specific case, among many others, belonging to a general behavior);
- Since behaviors not exposed by the considered execution traces cannot be mined, the tool includes a coverage analysis to measure the quality of the mined assertions. In case of a too low coverage the user can provide new templates to enrich the set of assertions. An heuristic-based minimization procedure is also implemented to reduce the final number of mined assertions without decreasing the overall DUV coverage.

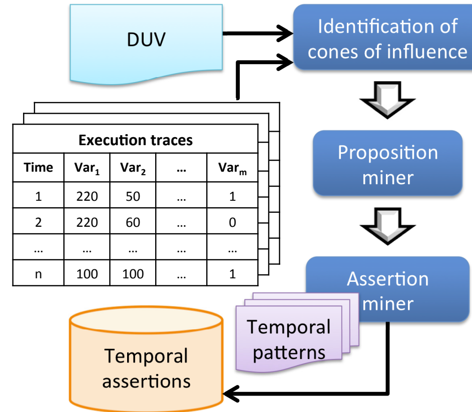


Fig. 4.2. The execution flow of *ODEN*.

4.4.1 Oden

Methodology

Figure 4.2 presents an overview of the approach implemented in *ODEN*. The inputs are a set of execution traces of the DUV. The output is a set of LTL assertions, compliant with the templates provided by the user, which hold on the execution traces supplied as input. *ODEN* works in three phases:

1. **identification of cones of influence:** the first step consist of analyzing the source code of the DUV and execution traces to extract the cone of influence for each primary outputs of the DUV. This step is necessary to prevent the assertion miner from generating assertions that mix variables belonging to different cones of influence. On the contrary, longer assertions could be generated that overlap unrelated behaviors, degrading both the readability and the quality of the mined assertions.
2. **mining of propositions:** execution traces are then partitioned according to the cones of influence and provided to the *proposition miner*. Each execution trace describes the values assigned to primary inputs (PIs) and primary outputs (POs) of the DUV at each simulation instant. For every cone of influence, the proposition miner is in charge of extracting propositions representing interesting relations between PIs and POs that appear frequently in the analyses execution traces.
3. **mining of temporal assertions:** the miner propositions are combined to create temporal assertions by the *assertion miner*. Mined assertions are in the form *antecedent* $\rightarrow$ *consequent*, where *antecedent* and *consequent* are temporal assertions reflecting temporal patterns described in the last section of the methodology *ODEN*. Considered variables are PIs and POs of the DUV, since I am interesting in capturing system behaviors at the boundary of the DUV. However, the methodology could be applied without changes to consider internal variables too.

Identification of cones of influence

The first step of the methodology consists in the identification of PIs that belong to the cone of influence of each DUV's PO. Given a target variable v, its cone of influence is represented by the set of variables that affect the value of v. This task is fundamental to better characterize the behaviors of the DUV avoiding the risk of generating assertions that capture unrelated behaviors in the same formula. I applied two complementary modalities to extract the cones of influence. When the DUV source code is available, tools based on static approaches can be adopted like, for example, CodeSurfer [21]. Currently, I have interfaced our methodology with CodeSurfer, since it provides a better support for C++ language, which can be automatically generated starting from popular hardware description languages like, VHDL, Verilog and SystemC by means of HIFSuite [14]. When the DUV source code is not available, the extraction of cones of influence can rely only on the analysis of the execution traces by adopting heuristics techniques like, for example, the solution proposed in Tane [43], which has the ability of producing a list of likely correlations among two or more columns of a table of values. When the DUV source code is not available, our methodology provides Tane with tables representing execution traces to extract functional dependences among PIs and POs. Independently from the adopted strategy, at the end of this phase, each execution trace is partitioned in different *slices* according to the extracted cones of influence.

Proposition miner

The purpose of the proposition miner is to generate formulas according to Def. 2.3, that will be used as *antecedents* and *consequents* for the final phase of the methodology. The proposition miner takes the slices of the execution traces[1] as input, and it works in two steps. It first analyses each slice to extract atomic propositions that describe simple relations between DUV variables, like, for example, $var1 > var2$, $var3 = True$, etc. Then, it composes atomic propositions to create more complex propositions that could represent antecedents or consequents of the final assertions, like, for example, $(var1 > var2) \wedge (var3 = True)$.

(Step 1 - Mining of atomic propositions): Mining of atomic propositions is performed by calling an external tool, i.e., Daikon [33], which is able to dynamically extract arithmetic/logic expressions among variables of the DUV by analyzing execution traces. In Daikon's terminology, atomic propositions are called *invariants*, since they hold throughout the analyzed trace. However, no temporal behavior can be observed by composing such invariants. Thus, execution traces are *tokenized* in sub-traces. Then, Daikon is called to extract invariants of such sub-traces. These invariants, being true only on some parts of the original execution traces, represent atomic proposition candidates for creation of temporal assertions, as described in the following steps of the methodology. Invariants that are true for all sub-traces are instead discarded. Despite of the fact that the next steps of the proposed methodology are independent from the way atomic proposition candidates are extracted,

[1] In the following of the *ODEN* methodology, I use the term execution trace instead of explicitly referring to its slices not to overload the writing. Indeed, each of the next steps is executed on the slices derived from the extraction of DUV cones of influence.

I have chosen Daikon to this purpose since it is one of the most powerful tools for such kind of inference. However, since Daikon execution represented the major bottleneck for the approach described in [15], in this proposal the use of Daikon inside the proposition miner has been optimized as follows.

- *Invocation on a reduced set of short sub-traces.* In order to extract the most complete set of atomic proposition candidates, execution traces have to be tokenized in an exhaustive way. Which means we should extract $\sum_{i=2}^{n-1} i$ sub-traces from an n-length execution trace. In order to avoid the generation of this large quantity of sub-traces, I studied how many invariants are generally mined changing the sub-trace sizes. After an empirical analysis on a large set of case studies, I indeed observed that sub-traces longer than 6 simulation instants very rarely provide new candidates w.r.t. shorter sub-traces. Thus, in the current methodology only sub-traces whose length is between 2 and 6 simulation instants are considered. This way, given an execution trace composed of n instants, the total number of sub-traces provided to Daikon is $\sum_{i=1}^{min(n-2,5)}(n-i)$.
- *Analysis of a reduced set of invariant patterns.* The list of invariant patterns considered by the Daikon's inference engine is very rich [53]. However, most of them are not interesting to mine temporal assertions for behavioral descriptions of hardware components or embedded SW and they can be removed to save execution time. For example, invariants typically occurring in software programs like $x.field$ *is null*, *array A is sorted*, etc., are irrelevant in our context. Thus, I restricted Daikon to search only for arithmetic/logic expressions involving the most common relational (e.g., $=, \neq, \leq, \geq, <, >$) and arithmetic (e.g., $+, -, ^*, \div$) operators. Moreover, I imposed also that constants are not allowed as operands of relational operators, with the only exception represented by the Boolean constants *True* and *False*. In most of cases, it is unlikely that atomic propositions like *variable* = *constant* play a decisive role for the functionalities of the design. Instead, it is generally more important to capture relations between variables.
- *More efficient invocation on sub-traces.* Daikon's execution flow is composed of three steps: (i) initialization of internal data structures according to the selected invariant patterns and the data types of considered variables, (ii) mining of invariants, and (iii) printing of results. By profiling the three phases on several case studies and different lengths of execution sub-traces, I derived interesting observations. In particular, I observed that the third phase is independent from the length of the sub-traces analyzed by Daikon and definitely negligible from the execution time point of view. On the contrary, the second phase strictly depends on sub-trace length. However, execution time related to the second phase is almost irrelevant (few milliseconds) for the very short sub-traces extracted by the tokenization procedure. The real bottleneck is represented by the first phase, which costs, in average, almost one second for each analyzed sub-trace independently from its length. Considering that in our approach the variables and the invariant patterns are always the same for every sub-trace, I optimized the Daikon's execution flow implementing this simple strategy: (i) create the initial data structures as usual (ii) by a deep copy save them in memory (iii) when a new sub-trace has to be analyzed, replace the local initial data structures with a copy of those previously saved. The time saved with this optimization is significant and it greatly reduces the impact of Daikon on the overall mining flow, as reported in the experimental results.

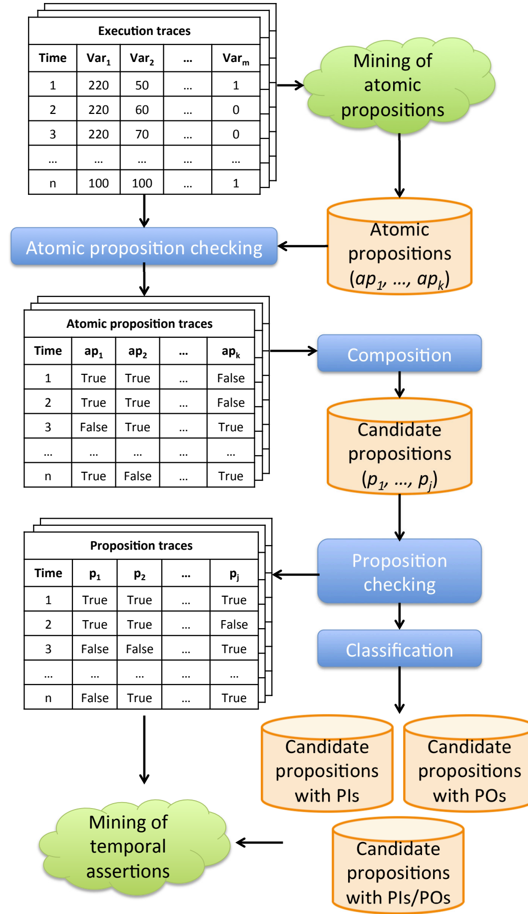


Fig. 4.3. Generation of candidate propositions.

(Step 2 - Generation of candidate propositions): The generation of propositions is performed according to the flow described in Figure 4.3. The set of atomic propositions is evaluated with respect to the execution traces. A checking procedure (*atomic proposition checking*) is executed to identify which atomic propositions are true in each instant of each execution trace. The output of this phase is represented by a table (*atomic proposition trace*) for each execution trace whose format is as follows. The first column refers to the time instants. Then, there is a column for each atomic proposition reporting its truth value for each time instant of the execution trace. Subsequently, a *composition procedure* generates a candidate proposition from each row of the atomic proposition trace by composing in an *AND* formula all atomic propositions that are marked as true. For example, in Figure 4.3, the first and the second atomic propositions (i.e., ap_1 and ap_2) are true at time instant 1, then a candidate proposition is created by composing ap_1 and ap_2 in the formula $p_1 := ap_1 \wedge ap_2$.

The next step (*proposition checking*) creates a new table (*proposition trace*) for each execution trace to identify which candidate propositions are true in each instant. Such trace will be then applied in the last step if the *ODEN* methodology to mine temporal assertions.

Finally, candidate propositions are classified according to the kind of variables (primary inputs, primary outputs or both) they involve. Such a classification is used to restrict the work space of the assertion mining algorithm and generate high-quality assertions. In particular, we can distinguish among:

- *PI propositions:* they involve only primary inputs of the DUV. They capture the behaviors of the testbenches used to simulate the DUV, while they cannot express anything about the behavior of the DUV. They are good candidates to be antecedents of temporal assertions.
- *PO propositions:* they involve only primary outputs of the DUV. They observe conditions occurring as a consequence of the DUV execution. They are definitely good candidates to be consequents of temporal assertions. However, they can be used also as antecedents when we are interested in capturing temporal implications between expected results of a DUV.
- *PIPO propositions:* they involve both PIs and POs of the DUV. They can be considered good candidates for both antecedents and consequents. However, when a PIPO proposition is used as a consequent, it could be appropriate to prune its atomic propositions that predicate only on PIs.

Assertion Miner

In the last phase of the methodology, the candidate propositions are combined according to a set of temporal patterns to create candidate temporal assertions. Given a candidate proposition p_a of type PI, PO or PIPO that acts as antecedent, and a set of candidate propositions $P = (p_c^1, ..., p_c^k)$ of type PO or PIPO that act as consequences, the considered patterns are the following:

1. *Next:* $always(p_a \rightarrow next\ p_c^i)$;
2. *N-next:* $always(p_a \rightarrow next[N]\ p_c^i)$;
3. *Until:* $always(p_a \rightarrow p_a\ until\ p_c^i)$;
4. *Alternating:* $always(p_a \rightarrow next\ (p_c^i\ before\ p_a))$.
5. *Next_or:* $always(p_a \rightarrow next\ (p_c^1 \vee p_c^2 \vee ... \vee p_c^k))$;
6. *N-next_or:* $always(p_a \rightarrow next[N]\ (p_c^1 \vee p_c^2 \vee ... \vee p_c^k))$;
7. *Until_or:* $always(p_a \rightarrow p_a\ until\ (p_c^1 \vee p_c^2 \vee ... \vee p_c^k))$.

These patterns allow to capture interesting behaviors between PIs and POs of the DUV according to the classification proposed in [29] that describes frequently used assertions for representing design specification. Patterns similar to number 1, 3, and 4 have been considered also in [52, 15, 13]. On the contrary, patterns 2, 5, 6 and 7 have never been considered by other temporal mining tools. Approaches based on Goldmine [54, 80, 78, 71] are instead oriented to capture chain of next events, like $p_1 \wedge next\ p_2 \wedge ... \wedge next[i]p_{i-1} \rightarrow next[i+1]p_i$. Such a kind of pattern is not considered in this work. I think it is more suited to predicate over internal variables of the DUV rather than PIs and POs, which are, instead, my target.

The assertion mining algorithm works as shown in Figure 4.4. For each of the considered patterns, a corresponding accepting automaton has been implemented.

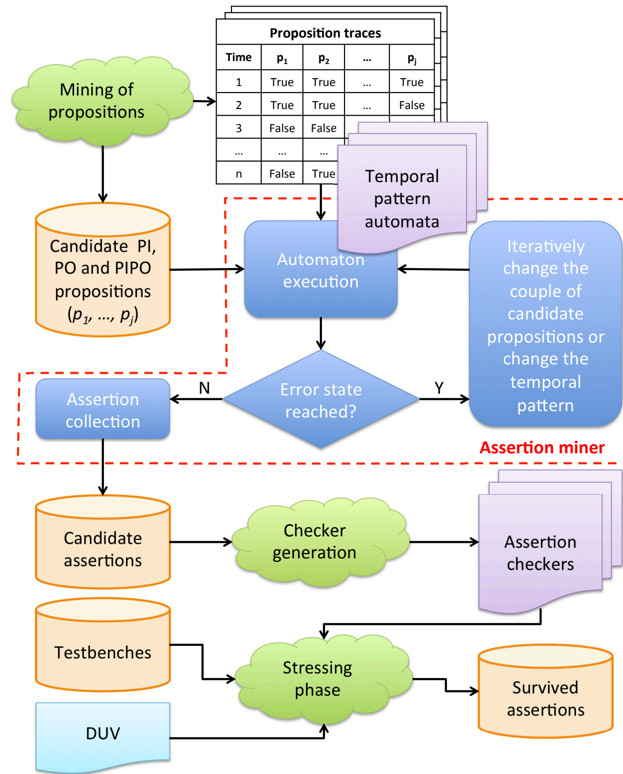


Fig. 4.4. Generation of temporal assertions.

Given one of the automata and given a couple of candidate propositions, the automaton is traversed by following each proposition trace generated by the proposition miner. If the error state is never reached for all the proposition traces, a candidate assertion is generated and stored by composing the two candidate propositions according to the considered temporal pattern. On the contrary, reaching the error state for at least one proposition trace is a sufficient condition to discard the candidate assertion. The proposed approach can be easily extended to support further temporal patterns by defining the corresponding automata and composing propositions accordingly.

The collected candidate assertions are then converted in checkers, by using, for example, IBM FoCs[2], and connected to the DUV. A different and very larger set of testbenches, with respect to the set initially used to generate the execution traces, is applied to stress the DUV and the candidate assertions searching for counterexamples. Each time a checker fails, the corresponding candidate assertion is discarded. Only assertions that survive to this stressing phase are definitely collected. The stressing phase is applied to increase the likelihood that the surviving assertions are satisfied by the DUV independently from the execution traces adopted for their extraction. Being a dynamic, not exhaustive, approach, we cannot be completely guaranteed, but larger is the testbench set higher is the probability of collecting

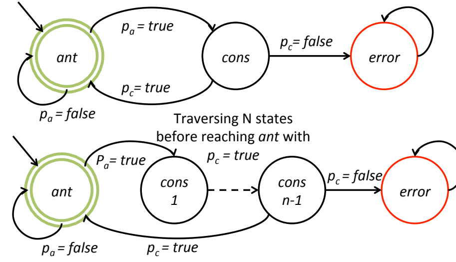


Fig. 4.5. Next (upper part) an N-next (lower part) pattern automata.

assertions that are satisfied by the DUV without the risk of escaping counterexamples. Since the mining procedure is much more expensive than simulating the DUV connected with checkers, it makes sense to use a reduced set of testbenches for the mining phase and a larger set of testbenches for the stressing phase.

In the remaining part of the section, I reported how the automata for the previously introduced temporal patterns works:

- Next-based patterns number 1 and 2 rely on the automata shown in Fig. 4.5. The only difference is represented by the number of states to be traversed before reaching the accepting state (*ant*) after the activation of the antecedent. In case the error state is reached the candidate assertion is discarded and a different couple of antecedent/consequent candidates is analyzed. The automata for patterns number 5 and 6 are similar, but in case the error state is reached at simulation instant t, on the assumption that p_a and p_c have been activated at least once before reaching t, an alternative searching procedure is activated. This procedure analyses the proposition trace to see if a different candidate proposition p_j is true at time t instead of p_c. If p_j is found, it is collected and the automaton restarts from the initial state searching for a new activation of the antecedent p_a in the rest of the proposition trace. When all proposition traces are completely traversed, collected propositions are composed in an OR formula together with p_c. Such a formula becomes the consequent of a next_or or N-next_or assertion where p_a is the antecedent. To avoid the risk a huge number of propositions are included in the OR formula, the error state can be reached a maximum number of times defined by the user. When this threshold is overcome the automaton stops and the couple of candidates p_a, p_c is definitely discarded. From my experience reasonable thresholds are between 2 and 4.
- The until pattern number 3 is similar to the next pattern number 1. Its automaton is depicted in Fig.4.6. The self loop of state *cons* allows that the precondition p_a happens an arbitrary number of times before p_c occurs. However, since $always(p_a \rightarrow p_a\ until\ p_c)$ is a logical consequence of $always(p_a \rightarrow next\ p_c)$, the self loop of state *cons* must be traversed at least once during the analysis of proposition traces to avoid that a next-based assertion is recognized also as an until-based assertion. On the contrary, the candidate assertion is written according to the next-based pattern instead of the until one. Concerning the until_or pattern number 7, considerations similar to the next_or pattern number 5 previously reported apply. In case the error state is reached at simulation

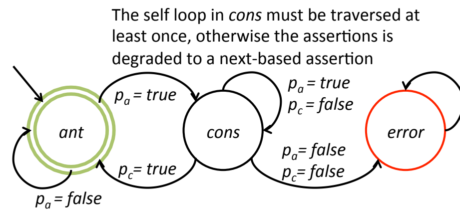


Fig. 4.6. Until pattern automaton.

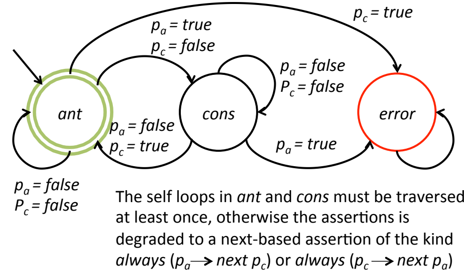


Fig. 4.7. Alternating pattern automaton.

instant t, an alternative searching procedure is activated to find if there exists p_j, different from p_c, which is is true at time t. When the execution trace is completely traversed, all collected propositions are composed together with p_c in an OR formula, which becomes the right operand of the until operator.

- The alternating pattern automaton is shown in Fig. 4.7. Similarly to the case of the until pattern, $always(p_a \rightarrow p_c\ before\ p_a)$ is a logical consequence of $always(p_a \rightarrow next\ p_c)$. Thus, the self loop of state *cons* must be traversed at least once during the analysis of proposition traces to avoid that a next-based assertion is recognized also as an alternating-based assertion. On the contrary, the candidate assertion is written according to the next-based pattern instead of the alternating one. Similar considerations apply for the self loop of state *ant*.

Experimental results

Experimental results have been carried out on an Intel Core2 Duo 2.2 GHz processor equipped with 2.0 GByte of RAM running Linux OS. Efficiency and effectiveness of the proposed mining methodology has been evaluated by considering the benchmarks reported in Table 4.1. *Thermostat* and *Breadmaker* are embedded SW (ESW) applications controlling, respectively, the heating system of an oven and a bread-making machine. *B03*, *B06* and *Dig_proc* are RTL behavioural descriptions modelling, respectively, a resource arbiter for four devices, an interrupt handler, and a digital filter implementing quadrature demodulation for two wavelengths. Finally, *Uart* is a TLM implementation of an UART controller. For *B06* and *BMaker* two cones of influence have been identified, only one for the other benchmarks. A set of execution traces for a total number of 10,000 simulation instants has been generated for all benchmarks.

DUV	*Typology*	*Cones*	*PIs*	*POs*	*Lines*
B03	RTL	1	4	4	141
B06	RTL	2	2	6	128
cone 1	RTL	-	1	4	-
cone 2	RTL	-	2	2	-
BMaker	ESW	2	4	4	552
cone 1	ESW	-	3	1	-
cone 2	ESW	-	1	3	-
Dig_proc	RTL	1	2	8	2580
Thermostat	ESW	1	2	1	56
Uart	TLM	1	9	5	14815

Table 4.1. Characteristics of benchmarks.

Table 4.2 reports the number of atomic propositions (AP), the number of propositions (P), and the number of temporal assertions (*Assertions*) extracted by the proposed approach before running the stressing phase. In particular, the number of assertions have been divided among next-based (X), until-based (U), and alternating (A). Concerning the N-next pattern, values 2 and 3 are considered for the parameter N. Finally, in the last two columns, the total execution time of the mining approach (*Time*) and the percentage of this time spent by Daikon (D) are shown.

As expected, the most time-expensive step of the miner is the extraction of atomic propositions performed by using Daikon. Daikon time is not so much affected by the number of considered variables. In fact, considering *B06* and *BMaker*, we observe that there is a low difference between the execution time related to the single cones of influence (where the considered variables are a subset of the total), and the execution time of the DUV without differentiating the cones of influence. Indeed, Daikon time is dominated by the time spent to initialize internal data structures, which depends mainly on the data type of the considered variables. For a Boolean variable only two invariant patterns have to be considered (i.e., $var = true$, $var = false$), which are very simple to be inferred. On the contrary, for numeric data types the number of invariant patterns is higher and their inference is more difficult. This justifies why benchmarks where only Boolean variables (e.g., bit and bit vectors) are involved (i.e., *B03*, *B06* and *Uart*) have an execution time lower than the other benchmarks, which are implemented by using integer (i.e., *Dig_proc*) or real data types (i.e., *BMaker* and *Thermostat*).

For *B03* a high number of propositions has been generated. This is due to the nature of the DUV that, being an arbiter among 4 devices, presents a high number of possible combinations among the four request signals and the four grant signals. Such different combinations give rise to a high number of next-based assertions according to the received requests. On the contrary, the sequential length of *B03* is too short to reflect until-based behaviors, and no evident alternating behavior is implemented by the arbiter.

For *BMaker* and *Thermostat* only until-based assertions are mined. This is consistent with the fact that their evolution depends on real data-type variables that evolve in a continuous, rather than discrete, way. Typical behaviors captured by

DUV	*AP*	*P*	*Assertions*				*Time*	*D*
			X	*U*	*A*	*Total*	*(s.)*	
B03	28	80	240	0	0	240	551	93%
B06 w/o cones	29	13	27	4	0	31	577	93%
B06 w/ cones	23	11	15	6	0	21	1078	93%
cone 1	14	6	12	2	0	14	540	93%
cone 2	9	5	3	4	0	7	538	93%
BMaker w/o cones	37	19	0	9	0	9	1641	89%
BMaker w/ cones	17	9	0	8	0	8	2949	89%
cone 1	8	4	0	3	0	3	1476	89%
cone 2	9	5	0	5	0	5	1473	89%
Dig_proc	15	4	12	0	0	12	1916	93%
Thermostat	4	3	0	3	0	3	1297	94%
Uart	20	19	57	0	16	73	394	92%

Table 4.2. Experimental results.

analyzing these benchmarks are "command is off until temperature is higher than setpoint" or "engine turns clockwise and engine turns fast until input of the mixer becomes false".

Extracted assertions have been then subjected to the stressing phase by stimulating the corresponding checkers connected to the DUV with up to 1 million stimuli. Table 4.3 reports the number of assertions for which the stressing phase was unable to found counterexamples at varying of the number of stimuli. For most of benchmarks, I found very few counterexamples by increasing the number of stimuli. Generally, by using a number of stimuli which is double (20,000) with respect to the length of execution traces adopted for the mining phase (10,000), the number of "survived" assertions stabilizes and no new counterexample is found any more. The only benchmark that does not converge on the number of survived assertions is *Uart*. Indeed, *Uart*, after a set of input is provided, requires 670 simulation instants before the corresponding result is observable at primary outputs. By using an execution trace of length 10,000 it means I can simulate completely no more than 15 different operations, which are too few for mining a set of assertions with a high degree of survival. However, this is not a problem of the proposed methodology, but a characteristic of the benchmark.

Finally, I report a comparison between *ODEN* and the approach proposed in [15]. Concerning the total execution time, the comparison is reported in Table 4.4. The table reports execution time at varying of the total number of considered simulation instants in the execution traces. Actual values are reported for *ODEN*; for the proposed methodology in [15], actual values are reported only for 100 and 1,000 simulation instants, since reaching 10,000 simulation instants becomes practically intractable for most of the considered benchmarks. The execution time for 10,000 simulation instants has been estimated on the basis of the tendency observed for shorter execution traces. By looking at the table, it appears that the increasing in execution time for the proposed approach is linear, at varying of the length of execution traces. On the contrary, for the proposed methodology in [15] the execution

Design	*Number of stimuli*					
	10000	*20000*	*40000*	*80000*	*100000*	*1M*
B03	240	191	177	171	171	171
B06 w/o cones	31	30	29	29	29	29
B06 w/ cones	21	21	21	21	21	21
cone_1	14	14	14	14	14	14
cone_2	7	7	7	7	7	7
BMaker w/o cones	9	9	9	9	9	9
BMaker w/ cones	8	8	8	8	8	8
cone_1	3	3	3	3	3	3
cone_2	5	5	5	5	5	5
Dig_proc	12	12	12	12	12	12
Thermostat	3	3	3	3	3	3
Uart	73	71	71	67	67	57

Table 4.3. Survived assertions after the stressing phase.

Design	*Length of execution traces*					
	ODEN			[15]		
	100	*1000*	*10000*	*100*	*1000*	*10000*
B03	6	58	551	569	11202	643227
B06 w/o cones	6	56	577	500	7920	118539
B06 w/ cones	11	105	1043	na	na	na
cone_1	5	51	505	na	na	na
cone_2	6	54	538	na	na	na
BMaker w/o cones	33	167	1641	474	5209	323040
BMaker w/ cones	60	296	2949	na	na	na
cone_1	30	149	1476	na	na	na
cone_2	30	147	1473	na	na	na
Dig_proc	13	135	1916	625	12721	730651
Thermostat	13	116	1297	394	7102	252501
Uart	14	134	394	629	10934	425375

Table 4.4. Comparison between execution time (in seconds) of *ODEN* and the proposed approach in [15].

time increases polynomially, till becoming unacceptable for long execution traces. This difference is mainly due to the different way Daikon is used in the extraction of atomic propositions, which represents the most expensive phase of both the methodologies. By adopting the optimizations described in the *Proposition miner* section of the *ODEN* methodology, the cost of Daikon's invocation on a set of sub-traces is almost 40 times lower than [15]'s approach.

A different comparison is related to the number of assertions extracted by *ODEN* and the proposed approach in [15] at varying the length of the execution traces.

Design	Length of execution traces						
	ODEN				[15]		
	100	*400*	*1000*	*10000*	*100*	*400*	*1000*
B03	132	230	240	240	1554	15448	38385
B06 w/o cones	40	39	37	31	204	586	608
B06 w/ cones	32	23	21	21	na	na	na
cone_1	18	16	14	14	na	na	na
cone_2	15	7	7	7	na	na	na
BMaker w/o cones	9	9	9	9	287	1266	834
BMaker w/ cones	8	8	8	8	na	na	na
cone_1	3	3	3	3	na	na	na
cone_2	5	5	5	5	na	na	na
Dig_proc	6	7	6	12	24	26	55
Thermostat	3	3	3	3	35	89	88
Uart	34	34	43	73	156	162	341

Table 4.5. Comparison between the number of assertions extracted by *ODEN* and the proposed approach in [15].

Results are reported in Table 4.5. We observe that the number of assertions extracted by the current methodology has an horizontal asymptotic trend, while for [15] the values generally keeps going to increase by augmenting the number of simulation instants. This highlights that the approach proposed is not dependent on the length of the execution traces, but on the number of different behaviors that execution traces expose. In fact, when the most of cases are covered by the execution traces, no new assertion is mined. On the contrary, in [15] the number of assertions keeps to increase for longer traces. The reason is evident by analyzing the assertions extracted by the two approaches. In [15], assertions are more related to the specific values assigned to PIs by the testbench. In the approach presented in this paper, assertions reflect symbolic relations between PIs and POs. For example, several assertions in [15] include atomic propositions of the kind *variable* = *constant*. Clearly, if the value's range of a variable is very large, the number of possible atomic propositions of this kind increases rapidly by using different stimuli. On the contrary, to avoid such a problem, in this work I explicitly discarded the possibility of comparing a variable with a constant. As a result, I *ODEN* generated a smaller set of assertions that focuses more precisely on the relation among PIs/POs that derives from the DUV functionality, discarding specific conditions that are just an instance of more interesting and more general behaviors.

Conclusions

In this thesis I presented *ODEN*, a mining approach for behavioral descriptions that automatically extract temporal assertions from execution traces. Mined assertions capture arithmetic/logic relations between PIs and POs according to a set of temporal patterns that can be easily extended. With respect to similar existing techniques, the proposed methodology points out an higher efficiency from the execution time point of view, and an higher effectiveness by considering the quality of mined assertions.

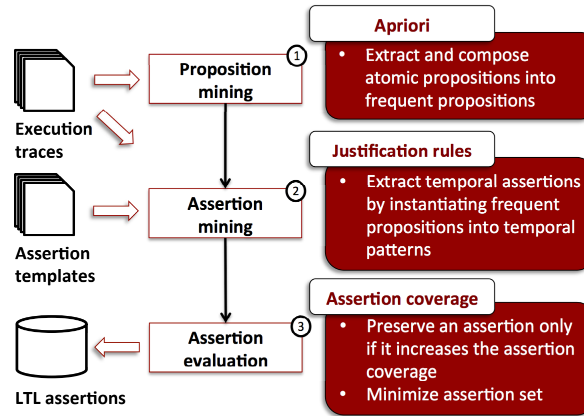


Fig. 4.8. The execution flow of *A-TEAM*.

4.4.2 A-TEAM

Background

The following definitions are necessary to formalize the methodology A-TEAM.

Definition 4.1. *A* ***template variable*** *tv is a place holder in a logic formula (def. 2.4) that can be substituted by a proposition (def . 2.3).*

Definition 4.2. *An* ***assertion template*** *is a logic formula composed of template variables through temporal operators and logical connectives.*

Methodology

Figure 4.8 presents an overview of the approach implemented in *A-TEAM*. The inputs are a set of execution traces of the DUV and a set of assertion templates defined by the user. The output is a set of LTL assertions, compliant with the templates provided by the user, which hold on the execution traces supplied as input. *A-TEAM* works in three phases:

1. **Mining of propositions:** The first step consists of the extraction of a set of propositions that frequently hold on the DUV's execution traces. These propositions are created by composing atomic propositions through *Apriori* algorithm [3]. At this stage, the temporal behaviour of the DUV is not considered yet. The mined propositions only describe the instantaneous status of the DUV at each simulation instant.
2. **Mining of LTL assertions:** The mined propositions are then composed into LTL assertions, compliant with the user-defined assertion templates. Each assertion template is represented as a syntax tree and a set of justification rules has been defined to instantiate propositions from the leaves to the root of the tree, such that the final assertion is guaranteed to be true on the execution traces provided as input.

3. **Evaluation of the mined assertions:** The goal of the third step is to evaluate the assertion coverage with respect to the DUV behaviours. Assertions that do not increase the coverage are discarded. Moreover, a minimization strategy is implemented to avoid collecting redundant assertions and to keep small the final set. The evaluation relies on a fault coverage-based analysis, according to the assumption that higher is the number of faults that make an assertion ϕ to fail, higher is the coverage of ϕ with respect to the DUV behaviors.

Proposition miner

The purpose of the proposition miner is to extract propositions that frequently hold on the execution traces to describe the instantaneous behaviors of the DUV at each simulation instant. These propositions are then provided to the assertion miner to mine temporal behaviors. The extraction of the propositions works as shown in Algorithm 3. It gets as input the set of DUV execution traces (*Traces*) and a set of atomic propositions that predicate on the output ports of the DUV (*apSetOut*). By default, *apSetOut* includes the following set of pre-defined atomic propositions for Boolean and bit vector data types (if any):

- $b_i = True$ and $b_i = False$, for each Boolean variable b_i that represents an output of the DUV;
- $B_i = c$, for each bit vector variable B_i that represents an output of the DUV, and for each constant c that can be assigned to B_i.

Moreover, *apSetOut* can be customized by the user, who is in charge of specifying further atomic propositions predicating on numeric data type ports (if any), like, for example, $v > 0$, $u \leq w + y$, etc. In this phase, the variables that represent inputs for the DUV are ignored, since their values depend on the testbench applied to stimulate the DUV, rather than on the DUV behaviors.

The output of Algorithm 3 is a set of propositions ($Propositions$), created by combining the atomic propositions belonging to *apSetOut*, which hold in some of the simulation instants of the $Traces$. We say that a proposition is frequent if it holds very often in the $Traces$[2]. Both, frequent and infrequent propositions are extracted. The intuitive idea is that the propositions that frequently hold in the DUV execution traces can represent a regularity in the behaviors of the design. Thus, in the phase 2 of the approach, *A-TEAM* initially directs the assertion miner to use the frequent propositions to generate temporal assertions that probably have a high coverage with respect to the DUV behaviors. Afterwards, *A-TEAM* asks the assertion miner to consider the infrequent propositions too, to generate temporal assertions covering the remaining corner cases of the DUV.

In detail, Algorithm 3 initially collects, inside *itemList*, a list of itemsets (lines 2-8) for all the execution traces. An itemset for the simulation instant t_j of a trace T_i represents the subset of the atomic propositions belonging to *apSetOut* that hold at t_j. Then, the proposition miner calls *Apriori* algorithm to identify frequent itemsets inside the execution traces (line 9). *Apriori* starts by generating sets containing only one atomic proposition. The sets with a low *support* are discarded [3]. Then, *Apriori*

[2] The threshold for saying a proposition is frequent is a parameter that can be set by the user.

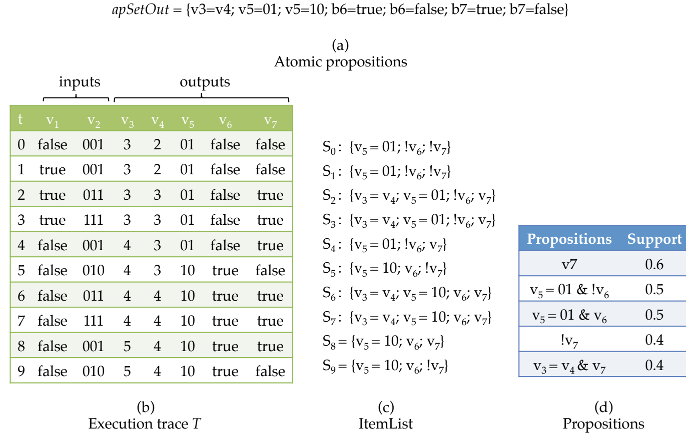
[3] Given a set S of atomic propositions, which are true all together n times inside an execution trace of length N, the support of S is defined as $support(S) = n/N$.

Algorithm 3

```
1: function propositionsMiner(Traces, apSetOut)
2:     itemList = ∅
3:     for all T_i ∈ Traces do
4:         for all t_i ∈ T_i do
5:             itemSet = ∅
6:             for all ap ∈ apSetOut do
7:                 if hold(ap, t_i, T_j) then
8:                     itemSet = itemSet ∪ {ap}
9:                 end if
10:                itemList = itemList ∪ itemSet
11:            end for
12:        end for
13:    end for
14:    FrequentSets = Apriori(itemList)
15:    Propositions = ∅
16:    for all set ∈ FrequentSets do
17:        p = True
18:        for all ap ∈ set do
19:            p = p AND ap
20:        end for
21:        Propositions = Propositions ∪ p
22:    end for
23:    return Propositions
24: end function
```

iteratively extends each survived set with a new atomic proposition, and again, the sets with low support are pruned. The algorithm stops when no set can be further enlarged without decreasing its support. *Apriori*'s complexity grows exponentially in the number of the considered sets. To bound its execution time, I perform this mining phase by providing *Apriori* with atomic propositions that predicates only on DUV output variables belonging to the same cone of influence, per each cone of influence. Finally, the proposition miner transform each frequent itemset returned by *Apriori* into a proposition, by combining, through the AND operator, the atomic propositions belonging to the same itemset (lines 10-14). At the end, the set of propositions is returned (line 15). The threshold for the support is a parameter that can be selected by the user.

As an example, let us consider the set of atomic propositions *apSetOut* listed in Figure 4.9a, and the execution trace T reported in Figure 4.9b, as inputs for Algorithm 3. Figure 4.9c shows, for each simulation instant t_i, the itemset S_i extracted by Algorithm 3, which collects the atomic propositions that are true at time t_i. Finally, Figure 4.9d shows the propositions generated from the frequent itemsets returned by *Apriori*, given the *itemList* of Figure 4.9c.



t	v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	false	001	3	2	01	false	false
1	true	001	3	2	01	false	false
2	true	011	3	3	01	false	true
3	true	111	3	3	01	false	true
4	false	001	4	3	01	false	true
5	false	010	4	3	10	true	false
6	false	011	4	4	10	true	true
7	false	111	4	4	10	true	true
8	false	001	5	4	10	true	true
9	false	010	5	4	10	true	false

Propositions	Support
v_7	0.6
$v_5 = 01$ & $!v_6$	0.5
$v_5 = 01$ & v_6	0.5
$!v_7$	0.4
$v_3 = v_4$ & v_7	0.4

Fig. 4.9. Generation of propositions.

Assertions Miner

The goal of the assertion miner is to generate LTL assertions by instantiating propositions in the user-defined assertion templates. Assertion templates represent specification patterns the verification engineer is interested in, but he/she does not know if they have been actually implemented, in some form, in the DUV. The set of mined assertions can confirm the expectations of the verification engineer, or they can contradict them through missing or unexpected temporal assertions. In particular, the miner searches for LTL assertions that instantiate the template $G(antecedent \rightarrow consequent)$, where *antecedent* and *consequent* can be formulas including logical connectives and temporal operators as reported in Def. 2.4, with the only restriction that the release (R) and until (U) operators can appear only on the *consequent*. For example, the verification engineer can be interested in discovering which propositions must replace the template variables a, b and c to create different concrete instances of the assertion template $G(a \rightarrow X(bUc))$ that hold on the DUV. A list of practice patterns, frequently adopted for representing design specifications, that can be used to define the assertion templates for our assertion miner has been proposed by Dwyer et al. in [30], but the user is free to provide *A-TEAM* with his/her own set.

Algorithm 4 illustrates the pseudo-code of the main function of the assertion miner, i.e., $makeAssert$. It takes as input an assertion template t, a proposition p extracted by the proposition miner, a set of atomic propositions on the DUV inputs $apSetIn$, a set of atomic propositions on the DUV outputs $apSetOut$, and a scope S, initially set at $\emptyset$, which is used to take care of simulation instants to be focused on during the mining procedure. Initially, the $makeAssert$ function is called on the assertion template $t : G(at_1 \rightarrow at_2)$, and then it recursively instantiates a temporal assertion compliant with t as follows:

- When t is $G(at_1 \rightarrow at_2)$ (line-3): The algorithm first generates a temporal assertion β for the consequent by recursively calling $makeAssertion$ on the assertion template at_2. Then, it applies $makeImply(at_1, apSetIn, \beta)$ to generate a new

Algorithm 4

```
1: function makeAssert(t, p, apSetIn, apSetOut, S)
2:     switch t do
3:         case G(at_1 → at_2)
4:             β=makeAssert(at_2,p,apSetIn,apSetOut,S)
5:             α = makeImply(at_1, apSetIn, β)
6:             return makeGobally(α, β)
7:         case tv
8:             if S is ∅ then
9:                 return p
10:            else
11:                return getInvariant(S, apSetOut)
12:            end if
13:        case at_1 & at_2
14:            α = makeAssert(at_1,p,apSetIn,apSetOut,S)
15:            S = evaluate(α)
16:            β = makeAssert(at_2,p,apSetIn,apSetOut,S)
17:            return α & β
18:        case at_1 | at_2
19:            α = makeAssert(at_1,p,apSetIn,apSetOut,S)
20:            S = evaluate(!α)
21:            β = makeAssert(at_2,p,apSetIn,apSetOut,S)
22:            return α | β
23:        case X[n](at)
24:            α = makeAssert(at,p,apSetIn,apSetOut,S)
25:            return X[n](α)
26:        case at U tv
27:            α = makeAssert(at,p,apSetIn,apSetOut,S)
28:            β = makeUntil(α)
29:            return α U β
30:        case tv R at
31:            β = makeAssert(at,p,apSetIn,apSetOut,S)
32:            α = makeRelease(β)
33:            return α R β
34: end function
```

assertion α for the antecedent at_1. The function *makeImply* works as shown in Algorithm 5 and it will be explained later in this section. However, at this time is worth noting that *makeImply* can return α in the form of a disjunction of formulas, namely $\alpha = \alpha_1 \mid \ldots \mid \alpha_n$. Finally, if both α and β are not NULL, *makeGlobally* decomposes α and it returns $G(\alpha_i \rightarrow \beta)$ for all α_i included in α, otherwise the result will be NULL.

- When $t = tv$, i.e. t is a template variable (line 7): If $S = \emptyset$, the algorithm instantiates t with the proposition p. On the contrary, the algorithm returns the proposition generated by $getInvariant(S, apSetOut)$. This function returns the formula $\bigwedge ap_j$, which composes with the AND operator all the atomic propositions ap_j belonging to $apSetOut$ that are true $\forall t_i \in S$. It returns, instead, NULL if no atomic proposition is true in S.
- When $t = at_1$ & at_2 (line 10): The algorithm first generates a temporal assertion α according to the assertion template at_1. Then, the function $evaluate(\alpha)$ updates the scope S by including only the simulation instants where α holds on the execution traces (line 12). Subsequently, it generates a temporal assertion β according to the template at_2 for the new scope S. Finally, it returns α & β, if both α and β are not NULL, otherwise the result will be NULL.
- When $t = at_1 \mid at_2$ (line 15): This case behaves similarly to the previous. The only difference is related to the updating of the scope S after α is generated. In this case, we are interested in discovering what is true when α is false. Thus, S is updated to include only instants where α does not hold (line 17).
- When $t = X[n](at)$ (line 20): The algorithm recursively calls *makeAssert* on the assertion template at to generate the assertion α. Then, it returns $X[n](\alpha)$ if α is not NULL, otherwise the result will be NULL.
- When $t = at\ U\ tv$ (line 23): The algorithm first generates a temporal assertion α according to the template at. Then, it applies $makeUntil(\alpha)$ to generate a new assertion β for tv. Finally, it returns $\alpha\ U\ \beta$ if both α and β are not NULL, otherwise the result will be NULL. The function *makeUntil* works as shown in Algorithm 6 and it will be explained later in this section.
- When $t = tv\ R\ at$ (line 27): This case behaves similarly to the previous. The only difference is that first we generate the right side of the release operator and then the left side by calling $makeRelease(\beta)$. The function *makeRelease* works similarly to *makeUntil*, according to the semantics of the operator R.

The function *makeImply* of Algorithm 5 is in charge of creating the antecedent α of an implication, given the consequence β that has been already mined, at the time *makeImply* is called. Let us remember that the only temporal operator that can appear in the antecedent α is X. Thus, the assertion template at passed to the *makeImply* is a conjunction $\bigwedge X[i](alpha_i)$, for some $i \in N$. According to the time spawn considered by the next operators included in at, the set of atomic propositions *apSetIn* is unrolled (line 4). For example, if $at = \alpha_0 \wedge X[3](\alpha_3)$, two instances of each atomic proposition in *apSetIn* are included in A, for their possible instantiation in, respectively, α_0 and α_3 placeholders. Then, a decision tree algorithm similar to the one adopted in [42] is used. It provides a set of formulas compliant with at, such that each of them implies β throughout the execution traces. However, the decision tree algorithm tends to overfit the antecedents, which become over-constrained by including atomic propositions in A that are useless to imply β. Thus, an optimization procedure is implemented to simplify the antecedents (lines 6-13). It decomposes each antecedent $a = \bigwedge X[i](alpha_i)$ in the set of formulas $da = \{X[i](alpha_i)$ s.t. α_i occurs in $a\}$. Then, it tries to prune each antecedent, such that a lower number of atomic propositions can be used to imply β. This is done by an iterative procedure that picks up a subset of formulas from da and checks if their conjunction implies β. The subset size is progressively increased starting from 2, such that, at the beginning, pairs of formulas included in da are considered. If no

Algorithm 5

```
1: function makeImply(at, apSetIn, β)
2:     α = false
3:     S = evaluate(β)
4:     A = unroll(at, apSetIn)
5:     antecedentSet = decisionTree(A, β, S)
6:     for all a ∈ antecedentSet do
7:         da = decompose(a)
8:         new_da = ∅
9:         size = 2
10:        while (size ≤ |da|) && (new_da = ∅) do
11:            new_da = prune(da, size, β)
12:            size + +
13:        end while
14:        new_a = recompose(new_da)
15:        α = α | new_a
16:    end for
17:    if α == false then
18:        return NULL
19:    else
20:        return α
21:    end if
22: end function
```

pair implies β, the size is increased and triples are considered, and so on. As soon as one iteration of the pruning phase succeeds in implying β, the procedure stops and the minimized antecedent is returned. In the worst case, the procedure returns exactly the same antecedent provided by the decision tree, when no minimization is possible. The final list of minimized antecedents are returned in OR (line 14). NULL is returned if no antecedent is mined for β.

The function *makeUntil* of Algorithm 6 is in charge of creating the right operand β of the formula $\alpha U \beta$, given α. The set of propositions returned by the proposition miner (section *Proposition miner* of the *A-TEAM* methodology) applied to atomic propositions predicating, this time, over both the DUV inputs[4] and outputs, are considered as possible candidates for β (line 4). The algorithm first extracts the set S of simulation instants t_i such that α holds till t_i but it does not hold at time t_{i+1}, for each sequence of occurrences of α in the execution traces (line 2). Then, for each proposition p that does not hold in the same instants where α holds except S (lines 4-5), it collects p if it holds for each instant included in S (line 6-9). A disjunction of all collected propositions is returned, or NULL in case no proposition is found.

Figure 4.10 exemplifies how *makeAssert* works, given the assertion template $G(vt_1 \rightarrow X(vt_2 U vt_3))$ and the proposition $(n_3 = n_4) \wedge b_7$. The first template

[4] Differently from the justification of the left side of an *until*, atomic propositions on inputs are also considered when the proposition miner is used for justifying the right side.

Algorithm 6

```
1:  function makeUntil(α)
2:      S = getLastInstants(α)
3:      β = false
4:      for all p ∈ Propositions do
5:          if evaluate(p) ∩ {evaluate(α) \ S}) = ∅ then
6:              found = true
7:              for all t_i ∈ S do
8:                  if p does not hold at t_i then
9:                      found = false
10:                     break
11:                 end if
12:             end for
13:             if found then
14:                 β = β | p
15:             end if
16:         end if
17:     end for
18:     if β == false then
19:         return NULL
20:     else
21:         return β
22:     end if
23: end function
```

matched by *makeAssertion* is $G(at_1 \rightarrow at_2)$. The algorithm then justifies the consequent of the implication, i.e., $X(vt_2 U vt_3)$, and then the template variable vt_1 representing the antecedent. The justification of the consequent requires to recursively call *makeAssert* on the *next* operator and successively on the *until* nested inside the *next*. Thus, the procedure begins trying to instantiate the template variable vt_2. Being the scope S still *empty*, the algorithm replaces $vt2$ with $n_3 = n_4 \wedge b_7$. Afterwards, it applies *makeUntil* to replace the variable vt_3. The instants where $n_3 = n_4 \wedge b_7$ is verified are $\{2, 3, 6, 7\}$. Thus, *makeUntil* collects and returns the proposition $!b_1 \ \wedge B_2 = 001$ which is verified at the instants $\{5, 8\}$. Once the justification of *until* and *next* cases is finished, the mined consequent is $X((n_3 = n_4 \wedge b_7)\ U\ (!b_1 \wedge B_2 = 001))$, and the recursion closes. Subsequently, *makeImply* is called to justify the antecedent of the implication. The propositions generated at the instants $\{1, 5\}$ are, respectively, $b_1 \wedge B_2 = 001$ and $!b_1 \wedge B_2 = 010$. Since both are exclusively verified in $\{1, 5\}$, the function replaces vt_1 with $(b_1 \wedge B_2 = 001) \vee (!b_1 \wedge B_2 = 010)$. At the end, *makeGlobally* breaks the disjunction in the antecedent to create the following two assertions:
$G((b_1 \wedge B_2 = 001) \rightarrow X((n_3 = n_4 \wedge b_7)\ U\ (!b_1 \wedge B_2 = 001))$;
$G((!b_1 \wedge B_2 = 010) \rightarrow X((n_3 = n_4 \wedge b_7)\ U\ (!b_1 \wedge B_2 = 001))$.

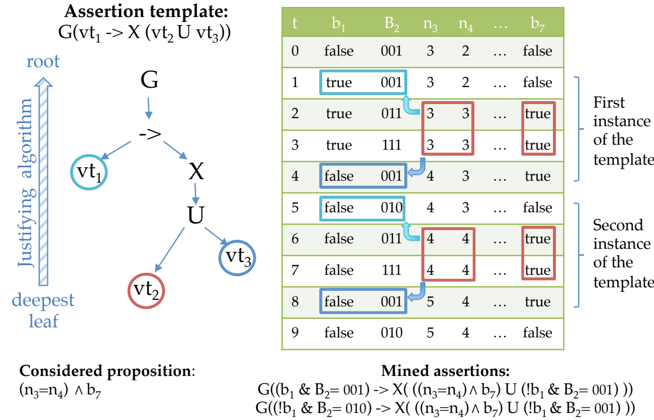


Fig. 4.10. Generation of a temporal assertion.

Algorithm 7

1: **function** $assertionsSection(U, X)$
2: $I = \emptyset$
3: **while** $\exists u \in U$ such that $u \notin \bigcup_{i \in I} u_i$ **do**
4: **for** $i = 1..n$ **do**
5: $with\ probability\ x_i\ I = I \cup \{i\}$
6: **end for**
7: **end while**
8: **return** I
9: **end function**

Assertion Evaluator

The goal of the third step of our approach is to measure the quality of the mined assertion by evaluating their coverage with respect to the DUV behaviors. The evaluation is based on the assertion coverage metrics proposed in [35]. A set of testable faults is injected in the DUV. Then, the set of faults detected by each mined assertion is collected. A fault is detected by an assertion if the assertion fails in the DUV affected by the fault, while it holds on the fault-free DUV. Given a set of assertions, their coverage is measured as the number of faults that are detected by at least one assertion in the set with respect to the total number of injected faults. Achieving 100% assertion coverage guarantees that the assertions cover all the behaviors affected by the faults. Higher is the quality of the adopted fault model, more accurate is the assertion coverage, as it happens when a fault model is used to measure the quality of an automatic test pattern generation process. During the mining phase, assertions that do not increase the overall coverage are discarded. Moreover, the final set of assertions is minimized as follows.

Given the sets $u_1, \ldots, u_n$, where u_i is the set of faults covered by the i-th assertion, *A-TEAM* applies an integer linear programming (ILP)-based approach to reduce the number of assertions without decreasing the overall assertion coverage. In

Table 4.6. Experimental results.

Benchmark characteristics				param.		#ass		size(ant)		cov (%)		time (s.)	
Name	PIs	POs	#F	#t	$\|tw\|$	A	D	A	D	A	D	A	D
uart_decode_3to8	7	8	16	1	1	11	28	4.6	3.5	100.0	100.0	1	1
uart_edge_det	1	4	14	3	4	4	11	2	2.3	100.0	100.0	1	1
uart_fifo_async16_sr	13	10	195	4	3	35	47	9.2	8.4	91.7	42.5	2970	161
uart_shift_reg_pl	18	16	127	2	4	59	112	7.8	5.4	100.0	70.0	3633	234
fpmul_packfp	35	32	163	2	2	88	119	3.3	2.5	96.3	88.9	581	40
fpmul_unpackfp	32	45	84	2	2	75	120	1.4	2	96.4	90.4	797	51
fpmul_fpround	36	36	124	1	1	75	139	1.4	3	100.0	89.5	147	67
fpmul_fpnormalize	36	36	141	1	1	108	189	2	3.4	100.0	100.0	56	40
apbBus_controller	142	183	141	1	2	387	232	3.2	3.2	90.0	65.4	1331	720

detail, a variable x_i is introduced for each i-th assertion, with the intended meaning that $x_i = 1$ when the i-th assertion is preserved, and $x_i = 0$ otherwise. We can then express the problem of minimizing the number of assertions in the following way:

$$minimize \sum_{i=0}^{n} x_i; \qquad subject\ to \sum_{i: u \in u_i} x_i \geq 1, \ \ \forall u \in U, \text{with } x_i \in \{0,1\}$$

In order to solve the ILP problem in polynomial time, I relax the constraint on variable x_i, by allowing $0 \leq x_i \leq 1$, instead of requiring $x_i \in \{0,1\}$. I then apply the CLP linear program solver (*https://projects.coin- or.org/Clp*) to solve the relaxed problem. Finally, a heuristics is used to decide if preserving or discarding each assertion, according to values x_i provided by the solution of the relaxed ILP problem. The heuristics is illustrated in Algorithm 7. It takes in input the set of faults $U = \{u | u \in u_i\}$ covered by at least one assertion, and the result, $X = (x_1, .., x_n)$, of the relaxed ILP problem. The set I collects the indexes of the assertions to be preserved; at the beginning it is initialized at $\emptyset$, i.e., no assertion is preserved (line 2). Until a fault $u \in U$ exists that it is not covered by any assertion, the algorithm iteratively preserves the i-th assertion with probability x_i (lines 3-4). When all faults in U are covered by at least one assertion, the algorithm returns I. The obtained solution is a $log(|U|)$ approximation of the optimal solution [81], where each mined assertion cannot be discarded without decreasing the coverage of the DUV.

Experimental results

Experimental results have been carried out on an Intel core i7 equipped with a 4.0 GHz processor and 8.0 GByte of RAM, running Linux OS. The effectiveness and efficiency of A-TEAM have been compared with the decision-tree based approach implemented in [42]. The comparison has been performed on the set of benchmarks showed in Table 4.6: some components of a UART, some components of a floating point multiplier, and an APB bus controller. Columns on the left part of the Table report the size in bits of the primary inputs (*PIs*) and primary outputs (*POs*), and the number of faults injected for the assertion coverage evaluation procedure (*#F*). The well-know stuck-at fault model has been adopted, and only testable faults have been considered. It is worth noting that the complexity of dynamic assertion mining depends on the number of PIs and POs of the DUV, on the number of DUV behaviors, and on the length of the analyzed execution traces. On the contrary, the number of DUV code lines is less relevant, as the source code is not considered. The execution traces adopted for mining the assertions have been created by simulating each

benchmark for 10,000 instants. Then, the table reports the configuration parameter for A-TEAM and the decision tree-based approach, i.e., the number of templates[5] defined to mine assertions with A-TEAM ($\#t$), and the width of the time window considered for the decision tree ($|tw|$). To be fair in the comparison, the same value for $|tw|$ has been fixed also in A-TEAM, when the decision tree procedure is called inside the *makeImply* function. Finally, for each benchmark, the table reports the comparison between A-TEAM (Columns A) and the decision tree-based approach (*Columns D*) in terms of: number of mined assertions ($\#ass$), average number of atomic propositions involved in the antecedent of the mined assertions ($size(ant)$), achieved assertion coverage (*cov (%)*), and execution time in seconds (*time (s.)*).

The following considerations are derived from the comparison of the results. A-TEAM generally extracts a lower number of assertions, with a lower number of atomic propositions in the antecedents. On the opposite, the number of atomic propositions in the consequents (not reported in the table) is always 1 for the decision tree-based approach, while it varies between 1 and 4 for A-TEAM. Antecedents are shorter thanks to the *decompose-prune-recompose* procedure adopted by the *makeImply* function, which minimizes the set of atomic propositions originally returned by the *decisionTree* function. As a consequence, A-TEAM assertions are not over-constrained in the antecedents, which positively impacts on their capability of covering, in average, a higher number of behaviors per assertion. In the few case where the size of the antecedent is higher (e.g., for *uart_fifo_async16_sr*), this depends on the fact that the decision tree-based miner gave up before completely exploring a DUV path, while A-TEAM went deeper in the analysis finding, at the end, an assertion that increases the total coverage. In general, the assertion coverage achieved by A-TEAM is always higher, even if the number of generated assertions is lower than the decision tree-based approach. This further proves that assertions generated by A-TEAM cover in a better way the DUV behaviors affected by the injected faults. This depends also on the capability of A-TEAM of generating assertions according to different temporal templates, which appears particularly evident for *uart_fifo_async16_sr*, where the A-TEAM coverage, based on 4 templates, doubled the decision-tree coverage that mines just one kind of assertion. Finally, being shorter (and fewer), the A-TEAM assertions are more easy and fast to be analyzed by a human. As a drawback of A-TEAM the higher accuracy is paid in terms of execution time, which is in some cases one order of magnitude higher that the decision-tree based approach. However, this is due to the higher number of assertion kinds that A-TEAM search for.

Conclusions

In this thesis I presented an assertion mining tool, A-TEAM, which automatically extracts assertions from execution traces of the DUV independently from its abstraction level (e.g., TLM, RTL, gate level). The miner is guided by user-defined templates, thus differently from existing approaches, it is not restricted to a set of predefined patterns. In addition, a coverage metrics and an integer linear programming-based approach are used to compact the set of mined assertions. Experimental

[5] Remember that the decision tree-based approach, instead, generates only one kind of assertions, in the form $G(a \to b)$, where only the X temporal operator can occur in a and b.

results showed that in comparison with the approach proposed in [42], A-TEAM generates a lower number of simpler assertions, but it achieves a higher coverage of DUV behaviors according to the coverage metrics proposed in [35]. Future works will be devoted to reduce the execution time by further optimizing the justification of the antecedents of implications, which currently represent the bottleneck of the approach.

5

Vulnerability Detection

5.1 Introduction

In the past decade, the number of firmware attacks has been on the rise [66, 47]. For instance, erroneous hardware configurations let attackers set protected memory locations as writeable [46]; vulnerable update routines allowed the execution of malicious code [45]; and vulnerable interrupt handlers were exploited to attack a firmware by performing operations when the CPU was in the most privileged execution mode [28].

Consequently, more sophisticated validation techniques and tools are necessary to guarantee an effective identification of firmware vulnerabilities. Unfortunately, an exhaustive formal validation of the whole system is not any longer feasible for nowadays complex embedded systems. Then, verification engineers are more and more required to prioritize the validation effort to target the most exercised and vulnerable features of a design. However, this collides with the intractable diversity of firmware vulnerabilities, which makes their detection a very challenging issue. As a consequence, vulnerabilities hidden in unlikely execution paths risk to escape the validation.

As many classes of vulnerabilities are difficult to find without simulation, but exhaustiveness of the analysis is also important, recent works have combined symbolic simulation and assertion checking in order to verify firmware execution flows (see Section 5.2). These approaches rely upon a user-defined set of formal assertions describing behaviours the firmware should not implement. Such assertions, which are generally derived from a set of security requirements written in natural language, are first turned into checkers, and then verified in each execution path of the firmware. If a checker fails, (e.g. a counter example for the corresponding assertion is found), then the user is alerted that a security requirement was not satisfied. However, if all assertions pass, no security vulnerability is detect and the system is deemed free from back doors. Unfortunately the definition of assertions is a difficult and error-prone manual task. Omitting the definition of an assertion exposes to the risk of an incomplete validation process, possibly leading to the incapability of detecting actual vulnerabilities.

5.2 State of the Art

I am not aware of any platform that can automatically generate assertions/properties highlighting the rare corner cases of a firmware. Other systems extending KLEE, which is the symbolic-simulator engine of *DOVE*, have been proposed [6, 48, 68, 5, 73, 61] in literature. *FIE* [26] is a platform built on top of KLEE for detecting bugs in small, simple firmware programs for the MSP4030 family of microcontrollers. *FIE* needs of the firmware's source code to perform symbolic simulation of the firmware, and it currently supports finding two type of bugs: 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 a firmware writes to a read-only memory location or to locked flash. S^2E [20] is a platform from École Polytechnique Fédérale de Lausanne (EPFL) built on top of the QEMU virtual machine [7] and the KLEE symbolic execution engine. The S^2E's novelty consists of its ability to scale to large real systems by selectively executing symbolically only those parts of a system that are of interest to the tests. On the contrary of *FIE*, S^2E simulated directly the binary code of a program. It allows to analyze properties and behaviors (e.g. number of cache misses) by applying external plugins while running execution paths. In [6] a prototype of a tool is proposed to specifically detect unauthorized read accesses performed by interrupt handlers. Given a snapshot of SMRAM, its base address, and the address of the variable interrupt handler in SMRAM, the tool uses S^2E to search for concrete examples that cause an interrupt handler to read memory locations outside of SMRAM. *DDT* [48] is a system for testing closed-source binary device drivers against undesired behaviors, like race conditions, memory errors, resource leaks. DDT has two main components: a set of pluggable bug checkers and a driver exerciser. The exerciser simulate the driver along its execution paths. The dynamic checkers watch the execution and flag undesired behaviors. DDT provides a default set of checkers, and this set can be extended with an arbitrary number of other checkers for both safety and liveness properties. *KleeNet* [68] and *T-Check* [51] are instead tools that use symbolic analysis to generate test cases for sensor networks and find safety and liveness errors in sensor network applications running on TinyOS.

5.3 Objectives

To overcome the manual definition of assertions and the related risks, I presented *DOVE* (Detection Of firmware VulnErabilities) framework in this thesis. By exploiting concolic testing and model counting, *DOVE* can automatically generate assertions pinpointing corner cases that could hide security vulnerabilities of a firmware running in a hardware platform.

5.4 Background

A *control - flow symbolic simulation* is a way of "exploring" all execution paths of a program. It works by considering symbolic values in specific locations of a program such as variables, input values or memory locations. A symbolic value represents all the feasible values that can be assigned. Afterwards, an initial symbolic state is

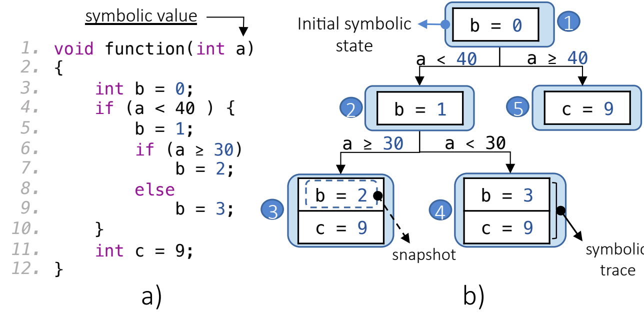


Fig. 5.1. Generation of a symbolic tree through the control-flow symbolic simulation of a program.

created to start simulating the program "step-by-step". At a high level, a symbolic state represents a running process having a register file, stack, heap and program counter. If during the simulation a symbolic state encounters a conditional statement, then new states are generated to independently follow each feasible execution path. For instance, let us consider the function shown in Figure 5.4a with a symbolic variable a. At the beginning the symbolic state 1 is created as reported in Figure 5.4b. Then the symbolic states 2 and 5 are generated from the state 1 because of the condition at line 4. Next the states 3 and 4 are generated from the state 2 because of the condition at line 6.
When a new symbolic state is generated, an *edge* is defined between the state that reached the condition, and the new one. Each edge is labelled with the constraints that have to be satisfied to follow the path simulated by the new symbolic state. For instance, the constraint $a < 40$ labels the edge between the states 1 and 2, meanwhile $a \geq 40$ is the constraint between states 1 and 5. The assignment of a value x to a variable w that occurs in the symbolic simulation is called a *snapshot* of the variable w. Let us take the variables b and c of the function in Figure 5.4a as example. During the symbolic simulation of the function we can observe four different snapshots for b, namely: $\{(b = 0); (b = 1); (b = 2); (b = 3)\}$, and one snapshot for c, namely: $\{(c = 9)\}$. A sequence of snapshots that occur during the execution of a symbolic state is called *symbolic trace*. The symbolic state 4 of Figure 5.4b has the symbolic trace $\langle(b = 3); (c = 9)\rangle$. The snapshot $(b = 3)$ is first generated because of the assignment at line 9. Meanwhile the snapshot $(c = 9)$ is generated because of the assignment at line 11. Throughout this thesis I refer to the whole set of symbolic states and edges generated by the symbolic simulation as a unique data structure named *symbolic tree*.

5.5 Methodology

5.5.1 DOVE

Figure 5.2 presents an overview of execution flow implemented in *DOVE*. The input parameters are a *firmware* in binary code and an *abstract hardware model* of the hardware where the firmware is executed. The output is a set of temporal assertions

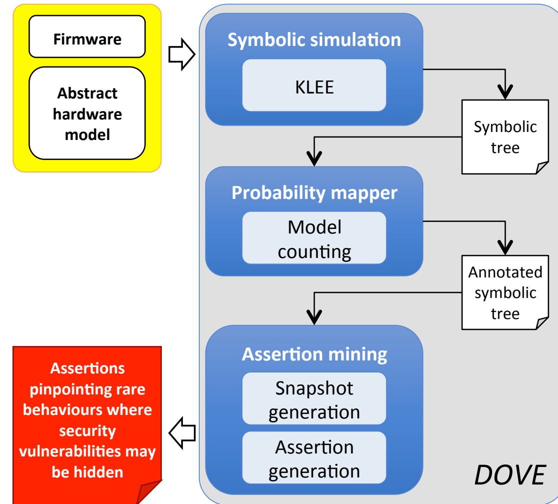


Fig. 5.2. The execution flow of the framework *DOVE*.

highlighting corner cases that may hide security vulnerabilities of the firmware under verification. *DOVE* works in three phases:

1. **Symbolic simulation:** The first phase consists of simulating the firmware with the purpose of maximizing its execution-path coverage. This objective is achieved by using KLEE[18], a symbolic simulator capable of concurrently executing different execution paths by exploiting symbolic values. The result of this step is a *symbolic tree* tracing how the values of the registers and/or memory locations of the abstract hardware model change after the execution of each instruction of the firmware.
2. **Probability mapper:** Each path from the root to a state of the previously generated *symbolic tree* is then analyzed. The objective of this analysis is to get the probability of following a given firmware's execution path. In particular, *DOVE* applies a weighted model-counting based approach to count the different solutions that satisfy the constraints representing the conditions encountered along a path π. Afterwards, the number of solutions is applied to get the probability of executing π.
3. **Assertion generator:** In the third and last phase, the goal is to generate formal assertions representing difficult-to-traverse execution paths that possibly hide security vulnerabilities that escape traditional verification approaches. Assertions are ordered according to the probability of observing specific values in the registers and/or memory locations of the abstract hardware model during the execution of the firmware.

Symbolic Simulation

The symbolic simulation of the firmware works as shown in Algorithm 8. It takes 5 input parameters: an abstract hardware model M, a firmware f, a set W' of memory locations and/or registers that will be filled out with symbolic values, a set W of memory locations and/or registers that will be traced after the execution

Algorithm 8

1: **function** $symbolicSimulation(M, f, W', W, n)$
2: $v = makeInitialSymbolicState(M, f)$
3: $V = v$
4: **for** i in $range(0, n)$ **do**
5: $V' = \emptyset$
6: **for all** $v \in V$ **do**
7: $fillOutSymbolicVariables(v, W')$
8: $fetch(v)$
9: $V' = V' \cup execute(v)$
10: **end for**
11: $V = V'$
12: **for all** $v \in V$ **do**
13: $trace(v, W)$
14: **end for**
15: **end for**
16: $T = makeSymbolicTree(V)$
17: **return** T
18: **end function**

of each instruction of the firmware, and an upper bound threshold n fixing the maximum number of instructions possibly executed in each feasible execution path of the firmware.

In detail, Algorithm 8 generates an initial symbolic state v by using the function $makeInitialSymbolicState(M, f)$ (line 2). This function runs a primitive of *KLEE* that instantiates the initial symbolic state, and performs the initialization steps of the hardware model M. In particular, the initialization steps of M consists in loading the binary code of the firmware f into the text memory and setting all registers and memory locations with an initial value. At the end of this step, v represents an instance of the abstract hardware model ready to execute the first instruction of the firmware. Next, the state v is inserted into the set V, which represents the pool of active symbolic states (line 3). Afterwards, for a fixed number of loops (lines 4-15), Algorithm 8 performs the following two macro phases for each state v in V: 1) it loads and executes a firmware instruction, 2) it takes a snapshot of the abstract model's variables.
In detail, in the first macro phase Algorithm 8 generates an empty set of symbolic states V' (line 5). Next, for each state v in V (lines 6-10), it fills out the variables of the hardware model of v by using the function $fillOutSymbolicVariables(v, W')$ (line 7). In particular, for each variable w' in W' $fillOutSymbolicVariables$ runs a primitive of *KLEE* to generate a new symbolic value x. Then, the corresponding variable w' in v gets the value x. After that, Algorithm 8 loads a new instruction of the firmware in v by using the function $fetch(v)$ (line 8). This function reads the program counter (PC) of the hardware model, and loads into the instruction registers (IR) the addressed instruction of the firmware. Finally, through the function $execute(v)$ (line 9), the loaded instruction is simulated. The result of this simulation is a set of symbolic states. In particular, if the instruction in IR is an arithmetic or

Algorithm 9

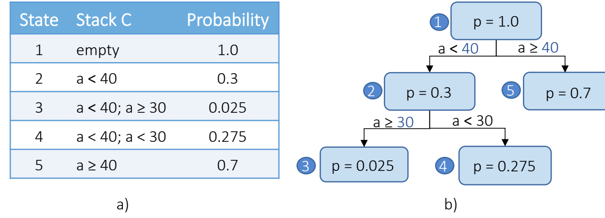
1: **function** $probabilityMapper(v, C)$
2: $p = computeProbability(C)$
3: $setProbability(v, p)$
4: **for all** b in $edges(v)$ **do**
5: $push(C, b.contraint)$
6: $probabilityMapper(b.node, C)$
7: $pop(C)$
8: **end for**
9: **end function**

memory related statement (e.g., load/store), then the state of the hardware model of v is updated by simulating the instruction in IR, and v is returned. On the contrary, if the instruction in IR is a conditional statement involving symbolic values, then a new symbolic state v' is generated for each feasible branch of the condition. Each v' inherits from v: (1) the current state of the abstract hardware model; (2) the sequence of constraints on the symbolic variables that was satisfied from the initial symbolic state to reach v'. Finally, the set of generated v' states are returned as result. When all the symbolic states v of V simulated an instruction of the firmware, the set V' collecting all the states returned by the function *execute* is assigned to V (line 11).
The second macro phase of Algorithm 8 makes for each symbolic state v of V a snapshot of the variables of the hardware model (lines 12-14) by using the function $trace(v, W)$. In particular, for each variable w of W a snapshot of the current value of w in v is generated and added in the *symbolic trace* of v. When all symbolic states in V have simulated a number of n instructions, Algorithm 8 generates a symbolic tree by using the function $makeSymbolicTree(V)$ (line 16).

Probability Mapper

The probability mapper calculates the probability of reaching any of the symbolic states from the root of the symbolic tree. It traverses the symbolic tree with a depth-first based strategy. At each visited state v, a solver is applied to count the solutions of the conditions encountered along the execution path from the root to v. The counted solutions are then used to compute the probability of v. The result of this phase is an annotated symbolic tree reporting for each symbolic state its probability of being reached starting from the root.

Algorithm 9 illustrates the pseudo-code of the function *probabilityMapper*. It takes as input parameters a symbolic state v, which initially is the root of the symbolic tree, and a stack of constraints C, which initially is empty. Then, it recursively computes the probability p of all the states in the tree by means of the function $computeProbability(C)$ (line 2), which is explained later in this section. As an example, let us consider to apply Algorithm 9 to the symbolic tree of Fig. 5.4b. Figure 5.3a shows, for each symbolic state, the constraints collected in the stack C, and the probability of reaching the state. Figure 5.3b reports the corresponding annotated symbolic tree.



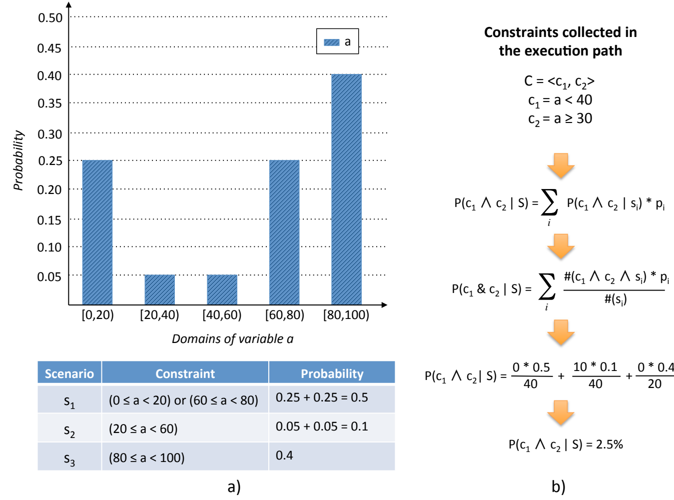
State	Stack C	Probability
1	empty	1.0
2	a < 40	0.3
3	a < 40; a ≥ 30	0.025
4	a < 40; a < 30	0.275
5	a ≥ 40	0.7

Fig. 5.3. Probability of reaching a symbolic state by satisfying the constraints along its execution path.

The function *computeProbability* calculate the probability of satisfying a set of constraints $C = \langle c_1, \ldots, c_n \rangle$ corresponding to an execution path π in the symbolic tree. Each c_i is an expression that involves arithmetic-logic operators among variables representing the condition to traverse the i-th edge of pi. These variables are either primary inputs of the model under validation or they depend on primary inputs. It is worth noting that the values assumed by primary inputs could be not uniformly distributed. Thus, the probability of satisfying C is computed according to the actual probability distribution of the input variables, which is derived by performing a dynamic trace profiling of the system under validation. Based on this distribution, *DOVE* computes the probability of satisfying C by exploiting the model-counting strategy proposed in [36]. Let D be the join of the range domains of all input discrete variables, and let $S = \langle (s_1, p_1), ..., (s_n, p_n) \rangle$ be a complete partition of D, where each s_i represents a different input scenario, *i.e.*, the subset of the feasible input values for the input variables characterized by the same probability p_i, with $\sum_i p_i = 1$. The value of p_i is derived by trace profiling. It represents the probability that the input values provided to the system at time t belongs to s_i. From the law of total probability [65], the probability of satisfying the set of constraints C according to S can be computed as: $P(C|S) = \sum_i P(C|s_i) * p_i$. Then, by applying conditional probability, we can rewritten the previous formula as $P(C|S) = \sum_i P(C \ \wedge \ s_i) * 1/P(s_i) * p_i$. Furthermore, $P(c)$ can be computed as $\#(c)/\#(D)$, where c is a constraint, and the operator $\#(.)$ returns the number of element of D satisfying c. Applying this definition of probability, the previous formula can be finally rewritten as $P(C|S) = \sum_i \#(C \ \wedge \ s_i) * 1/\#(s_i) * p_i$.
Let us consider the set of constraints C listed in Figure 5.4a, and the set of scenarios $S = \langle (s_1, p_1), (s_2, p_2), (s_3, p_3) \rangle$ derived from the probability distribution of the input values of the variable a (Figure 5.4a) as an example. By computing the conditional probability formula (1), we have that C has zero probability of being satisfied in the first and third scenario since no value either in s_1 or s_3 satisfies all the constraints belong to C. On the contrary, for the scenario s_2, ten values on forty can satisfy all constraints in C. By weighting with the probability of s_2, namely p_2, we have $P(C|S) = 2.5\%$.

Assertion Generation

The goal of the assertion generator is to generate formal assertions pinpointing corner cases that may hide security vulnerabilities. The assertion generator works by using first the function *snapshotGenerator* shown in Algorithm 10 to get a set of snapshots. Then, it uses the function *assertionMiner* shown in Algorithm 11



Scenario	Constraint	Probability
s_1	($0 \le a < 20$) or ($60 \le a < 80$)	0.25 + 0.25 = 0.5
s_2	($20 \le a < 60$)	0.05 + 0.05 = 0.1
s_3	($80 \le a < 100$)	0.4

Fig. 5.4. Getting the probability of satisfying a set of constraints given a set of scenarios partitioning the input space domain.

to generate a LTL temporal assertion (def. 2.5) for each snapshot. The generated assertions are in the form *G(antecedent → consequent)*, where *antecedent* and *consequent* may involve only the LTL operator X. Moreover, *antecedent* is composed only of the constraints collected along an execution path of the symbolic tree. The *consequent* is a snapshot, namely a specific value in a register or a memory location of the hardware abstract model.

The function *snapshotGenerator* gets as input parameters an annotated symbolic tree T, and the set W of traced variables during the symbolic simulation. At the beginning, Algorithm 10 initializes the list of snapshots S with $\emptyset$ (line 2). Next, for each variable w of W (lines 3-10), it collects in S' the set of snapshots of w by visiting the tree T (line 4, $getSnapshots(T, w)$). Then, for each snapshot s' of S', Algorithm 10 gets the probability p' of observing s' during the simulation of the firmware by using the function $DFSAllPaths$ (line 6). The intuitive idea is to reach all symbolic states belonging to different execution paths where the snapshot s' was traced for the first time. The probability p' of observing s' is, therefore, the sum of the probabilities p of reaching the identified symbolic states. The function $DFSAllPaths$ implements this strategy through a depth-first search algorithm. It gets as input parameters a symbolic state v, which is initially the root of the annotated symbolic tree T, and a snapshot s'. If s' is traced in v, then $DFSAllPaths$ returns the probability p of v. On the contrary, $DFSAllPaths$ forwards this search to each node reachable from v through a recursive call. If a leaf node has not the snapshot s', then $DFSAllPaths$ returns the value 0. All returned values are then summed up and returned as final result. At the line 7, Algorithm 10 annotates the snapshot s' with the probability p' by using the function $annotateSnapshot(s', p')$. The annotated snapshot s is then inserted into the list S (line 8). At the end of Algorithm 10, the list S of snapshots is sorted in accordance with their probability (line 11), and returned as result (line 12).

As an example, let us consider the annotated symbolic tree of Figure 5.3b as input

Snapshots	Probability
(b=0)	1.0
(b=1)	0.3
(b=2)	0.025
(b=3)	0.275
(c=9)	0.7+ 0.275 + 0.025 = 1.0

a)

Ranking	Assertions
4	(b=0)
2	G (a < 40 → X[1](b=1))
1	G ((a < 40 & a ≥ 30) → X[2](b=2))
3	G ((a < 40 & a < 30) → X[2](b=3))
4	G (a ≥ 40 → X[1](c=9))

b)

Fig. 5.5. The probability of observing a snapshot and the corresponding generated assertion.

of Algorithm 10. Figure 5.5a shows, for each snapshot s, the probability of observing s during the simulation of the firmware.

The second step of the assertion generation phase is to generate an assertion for each snapshot. The function $assertionMiner$ works as shown in Algorithm 11. It gets as input parameters the annotated symbolic tree T and a list of snapshots S sorted by using the probability. At the beginning, Algorithm 11 initializes a list of assertions A with $\emptyset$ (line 2). Next, for each snapshot s in S (lines 3-8), it searches by using the function $BFSearch(T.root, s)$ (line 4) a symbolic state v of T where the snapshot s was traced. To keep compact the antecedent of the assertions the function $BFSearch$ implements a breadth-first search based strategy to identify the state s. In detail, the function defines at the beginning a frontier F of states, which initially contains only the root node of T. If a state v of F traced the snapshot s, then v is returned as results. On the contrary, $BFSearch$ enlarges F by collecting all the reachable states of T from a state already in F. The set F is continually enlarged as long as a state v that traced s in its symbolic trace is identified. The antecedent for the snapshot s is then generated by collecting in inverse order the constraints of the edges of the annotated symbolic tree T from the state v to the root (line 5 $getRevOrderConstraints(v, T.root)$). A new assertion having the snapshot as consequent is generated (line 6), and insert into the list A (line 7). At the end, the Algorithm 11 returns the list of generated assertions (line 9). As an example, let us consider the annotated symbolic tree of Figure 5.3b, and the snapshots of Figure 5.5a as inputs of Algorithm 11. Figure 5.5b shows, for each snapshot s, the generated assertion with the corresponding ranking value (lowest is the most rare corner case). Let us consider the assertion ranked as the most rare corner case, e.g. $G((a < 40 \wedge a \geq 30) \rightarrow X[2](b = 2))$. This assertion reports: if the constraint $(a < 40) \wedge (a \geq 30)$ is satisfied, then the variable b gets the value 2 after 2 simulated instructions. This assertion belongs to an unlikely simulation flow as only with probability 2.5% the values of a can satisfy the antecedent of the assertion. The other assertions in Fig. 5.5b have instead a much higher probability because many distinct values of a satisfy their antecedents. The snapshot at the first row does not involve the variable a. It reports that immediately b has value 0 without any constraint. A verification engineer can then be addressed to investigate if the detected unlikely execution flow may hide a security vulnerability.

Framework

Figure 5.6 shows the simulation environment of the proposed framework. It consists of an ARM instruction set simulator (ISS), a firmware loader that copies the firmware binary code under verification into an internal memory of the hardware

Algorithm 10

```
1: function snapshotGenerator(T, W)
2:     S = ∅
3:     for all w in W do
4:         S′ = getSnapshots(T, w)
5:         for all s′ in S′ do
6:             p′ = DFSAllPaths(T.root, s′)
7:             s = annotateSnapshot(s′, p′)
8:             S = S ∪ s
9:         end for
10:    end for
11:    sortByProbability(S)
12:    return S
13: end function
```

Algorithm 11

```
1: function assertionMiner(T, S)
2:     A = ∅
3:     for all s in S do
4:         v = BFSearch(T.root, s)
5:         antecedent = getRevOrderConstraints(v, T.root)
6:         a = makeAssertion(antecedent, s)
7:         A = A ∪ a
8:     end for
9:     return A
10: end function
```

model, and a memory-mapped register interface (MMRI) that allows users to specify which memory address a register is mapped to (if any).
Given a new firmware implementation, a verification engineer is supposed to customize the following components of the framework before starting verifying the firmware.

Memory mapped-register: if, for the scenario of interest, some registers are memory mapped, then the user has to provide the memory address of each memory mapped register. In detail, *DOVE* provides the user with the array *memoryMapped*. During the simulation of a load/store instruction of the firmware, *DOVE* scans *memoryMapped*. If the addressed memory location of a load instruction matches an address in *memoryMapped*, then the value of the corresponding register is returned to the firmware. Similarly, in case of a simulated store instruction, the corresponding memory mapper register is updated with the new value provided by firmware.

Symbolic values: To perform symbolic simulation of the firmware some memory locations of the abstract hardware model have to be filled out with symbolic values. This part is the most crucial of the verification process as a too large num-

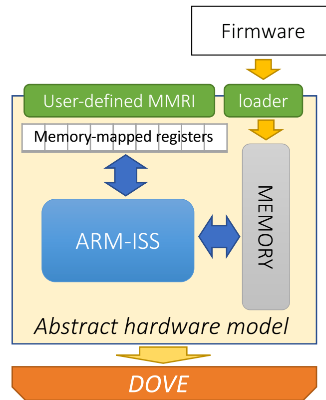


Fig. 5.6. Internal overview of the abstract hardware model.

ber of symbolic variables can lead to the path explosion problem: the number of globally feasible paths is roughly exponential in the size of the whole system. On the contrary, an insufficient number of symbolic variables can lead to not detecting the security vulnerability of the firmware since the unlikely execution flow will not be simulated. With *DOVE* a user can perform a selective symbolic simulation of the functionalities of the firmware. In particular, it can address the framework to continually generate new symbolic values for only the memory locations that will be read by the functionality under verification. *DOVE* provides the user with the *C++* function:

- **void** *fillOutSymbolicVariables*(**Address** *addresses*);

It gets as input parameter an array of memory addresses. *DOVE* will store a new symbolic value in each addressed register or memory location before executing an instruction of the firmware.

Trace values: To generate assertions some memory locations and/or registers of the abstract hardware mode have to be traced. The definition of these locations is up to the user, which can address *DOVE* to consider specific as well as generally used registers and memory locations. *DOVE* provides the user with the *C++* functions:

- **void** *trace*(**Address** *addresses*);
- **void** *traceCPU*(**Register** *regIndices*);

The first function gets as input parameter an array of memory addresses, meanwhile the second gets an array of CPU registers' indices.

CaseStudy

I evaluated the effectiveness and the efficiency of *DOVE* in two case studies concerning the validation of a memory protection mechanism and of an interrupt service handler. In both cases, the experimental results have been carried out on a 2.6 GHz Intel Core i5 processor equipped with 8 GByte of RAM and running Linux OS.

Case study 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 writeable unless the flag *bioswe* is set. Moreover, each attempt to change this flag causes an interrupt which 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 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, as reported in [46].
Consequently, if the firmware does not properly check the values provided to its input interface, an attacker can exploit it to first disable the memory protection mechanism, and then to write in a protected memory location. I run *DOVE* in the above vulnerable context. We configured DOVE such that it considered symbolic values for the registers were the IP core interface was mapped. 30 assertions were generated. In Fig. 5.7 the generated assertions are represented by the blue points. They are ordered according to the probability of traversing their corresponding execution paths of the firmware. The assertion with the lowest probability (i.e., $1.4e{-}39$) is:
$a1 = G((X(offset = 25) \wedge X[3](data = 0) \wedge X[9](offset = 49) \wedge X[11](data = 1) \wedge X[17](23 > 300 + offset)) \rightarrow X[22](bioswe = 1))$.

By analyzing the meaning of $a1$, we observe that it captures the situation where the values generated by the IP interface asks the firmware to: 1) disable the interrupts (when $offset$ is 25, the generated physical address hits *gbl_smi_en*); 2) enable the protected memory (similarly, *bioswe* is hit when offset has value 49); 3) perform a write in its own code. All memory addresses satisfying the constraint $(23 > 300{+}offset)$ hit the text memory. The behavior captured by $a1$ actually highlights that the firmware can be exploited to attack the system as reported above. Other top ranked assertions ($a2$, $a3$) highlighted the same behavior In fact, they correspond to execution paths where the memory mapped registers are manipulated by the firmware in a different order and with different values. This proves *DOVE* was able to focus the attention on rare and dangerous execution paths representing the presence of a security vulnerability. Subsequently, I removed the vulnerability by modifying the firmware such that it cannot write into the memory mapped register *gbl_smi_en*. Then I run *DOVE* again. This time *DOVE* generated 23 assertions represented by the orange points in Fig. 5.7. The probabilities of this new set of assertions is much higher compared with the probabilities of blue assertions. Thus, no extremely rare behavior is highlighted this time. In addition, the top-ranked orange assertions (*b1*, ..., *b4*) still represent behaviors where the firmware tries to access memory-mapped registers, but none of them corresponds to a path where *gbl_smi_en* is modified.

Case study 2: interrupt service handler

Interrupt handlers are stored in a protected and inaccessible memory location. The base address of this location is stored in a CPU internal register (e.g., *SMBASE*). When an interrupt is triggered by the firmware, the CPU switches to the supervisor

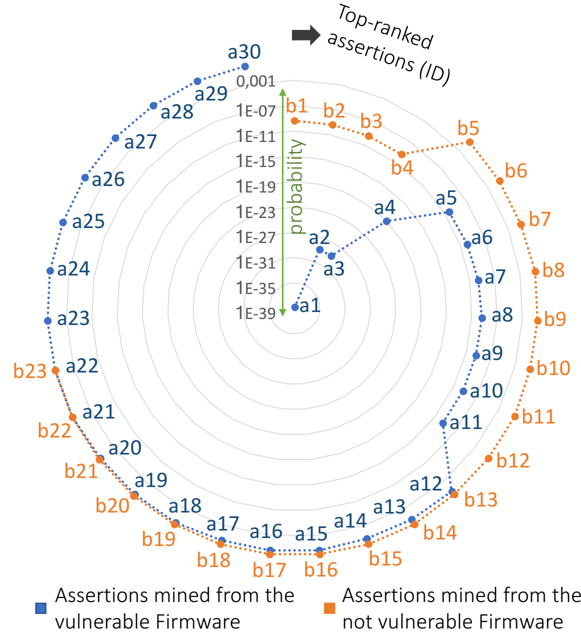


Fig. 5.7. Case study 1: Probabilities of assertions generated by *DOVE*, with (blue) and without (orange) firmware vulnerability.

mode, copies the values of *SMBASE* in a location of the protected area, and starts fetching the instructions of the proper interrupt handler. When the handler returns, the CPU restores *SMBASE* by coping back its value from the protected area and switches to the user mode. In this scenario, a security attack can exploit a vulnerable implementation of an interrupt handler to execute unauthorized operations when the CPU is in a privileged mode as reported in [28]. In a not-attacking execution flow, the interrupt handler is supposed to write values in a buffer outside the protected area. However, if the interrupt handler does not check the correctness of values provided as inputs, an attacker can exploit this vulnerability to overwrite the value of *SMBASE* stored in the protected area. When a second interrupt then occurs, a memory location referred to a malicious code is read by the CPU owing to a different value in *SMBASE*.

We run *DOVE* in the above scenario and it generated 60 assertions involving the program counter (PC) of the CPU. Figure 5.8 shows in the x axes the memory addresses corresponding to the instructions pointed by the PC during the simulation, and in the y axes the probability that the PC points to those memory addresses. The top ranked assertions generated by *DOVE* pinpointed the corner cases in the execution flow of the firmware where unauthorized operations are performed, thus guiding the verification engineers towards suspicious behavior that may correspond to security holes. In particular, the assertion associated to the most unlikely execution path (probability= $1.4e{-}18$) was:
$G(X[37](mem[0] = 0) \wedge X[48](mem[4] = 0x4108) \rightarrow X[68](PC = 0x3910))$.

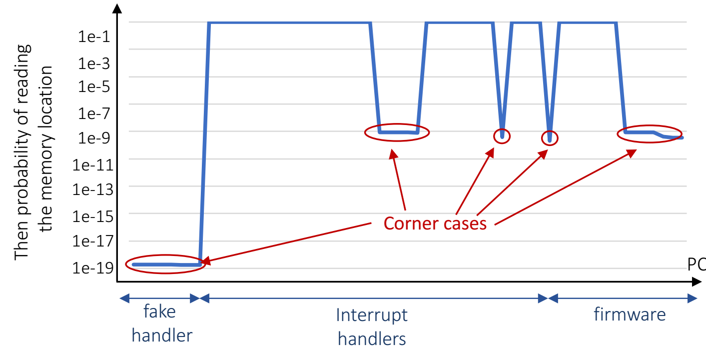


Fig. 5.8. Probability of reading the memory location addressed by the program pointer register.

Table 5.1. Case study 1: Symbolic states, assertions and execution time.

Instr.	States	Assertions	Sim. time	P. mapper time	Gen. time	Total time
5	1	9	< 0.1 s.	< 0.1 s.	< 0.1 s.	< 0.1 s.
10	7	17	< 0.1 s.	0.2 s.	< 0.1 s.	0.2 s.
15	37	21	0.6 s.	1.3 s.	< 0.1 s.	1.9 s.
20	37	23	0.8 s.	1.3 s.	< 0.1 s.	2.1 s.
25	188	27	3.2 s.	8.7 s.	< 0.1 s.	11.1 s.
30	188	30	4.3 s.	8.9 s.	< 0.1 s.	13.2 s.

This assertion reports how specific values processed by the interrupt handler allow the CPU to fetch an instruction from the memory address $0x3910$, where I stored a fake (malicious) handler to reset the value of $SMBASE$.

DOVE scalability

Table 5.1 and Table 5.2 report information to analyze the scalability of the approach implemented in *DOVE*. In particular, their columns refer, from left to right, to the number of instructions executed by the firmware and consequently to the number of states generated by the symbolic simulation, the number of assertions generated by *DOVE*, the time required by the three phases of *DOVE*, and the total execution time concerning for the two use cases.

As expected, the higher is the number of instructions executed by the firmware, the larger is the set of generated symbolic states. The number of generated assertions increased as well, nonetheless the final set of assertions is small and generated in a few seconds. For the first case study I stopped when no more symbolic state was generated, while in the second case study I terminated the simulation when no new assertion was mined.

Conclusions

In this thesis I proposed DOVE, a framework to detect firmware's vulnerabilities that could be exploited by an attacker to breach the system security. *DOVE* is based on a

Table 5.2. Case study 2: Symbolic states, assertions and execution time.

Instr.	States	Assertions	Sim. time	P. mapper time	Gen. time	Total time
25	1	19	< 0.1 s.	< 0.1 s.	< 0.1 s.	< 0.1 s.
50	11	41	0.1 s.	0.2 s.	< 0.1 s.	0.3 s.
75	95	57	1.1 s.	2.4 s.	< 0.1 s.	3.5 s.
100	211	58	2.3 s.	6.8 s.	< 0.1 s.	9.1 s.
125	340	60	7.0 s.	12.5 s.	< 0.1 s.	19.5 s.
150	456	60	10.8 s.	20.5 s.	< 0.1 s.	31.4 s.
175	594	60	19.6 s.	31.3 s.	< 0.1 s.	51.0 s.
200	3343	60	63.7 s.	170.3 s.	< 0.1 s.	234.3 s.

symbolic simulation engine that exhaustively explores the firmware's computational paths and provides the verification engineers with a ranked set of assertions. These assertions describe corner cases in the firmwares execution where vulnerabilities could remain undetected by applying traditional verification approaches. *DOVE* effectiveness and efficiency have been evaluated in two actual scenarios. In both cases, *DOVE* was able to highlight the vulnerabilities in a few seconds by generating a compact set of assertions. The set up of the verification process performed with *DOVE* required to add a few lines of code inside an ARM-based abstract model of the target hardware. This proved that *DOVE* can be easily and quickly customized to address other kinds of vulnerabilities with respect to the ones considered in our case studies.

Part III

Extra-Functional Verification

6

Power State Machine

6.1 Introduction

Power state machines (PSMs) are a well-known formalism to model and simulate the time-based energy consumption of IP cores for early virtual prototyping of system-on-chips (SoCs) [10, 11, 70, 58, 56, 49]. In this context, the PSMs of IPs included in the model of the target SoC are monitored by a power manager to allow the exploration of different dynamic power management solutions [8].

In a PSM, the energy behaviors of the IP are associated to a set of states. In its simplest form, the power consumption of each PSM state is modeled as a constant value derived by a designer estimate or from the IP's data sheet [10, 11]. When a higher level of accuracy is desired and more precise information about the IP's energy behaviors are available, the power consumption of a PSM state is computed by a more complex function. For example, in [70, 56], such a function is derived by means of a calibration process based on linear regression, which exploits, as golden reference, power traces generated at gate level, where the IP's power consumption can be more precisely estimated. However, despite of the wide adoption of PSMs, in the most of the works either the presence of PSMs is assumed [10, 11, 8] or they are manually defined starting from a more or less precise knowledge of the functional blocks composing the target IP [58, 49]. Only in a few cases, automatic approaches are proposed to create the association between PSM states and their power consumptions, but the identification of such states remains manual [70, 56]. Unfortunately, such a manual definition reveals to be inappropriate for the power characterization of complex designs leading to the generation of a less-accurate simplified model.

6.2 State of the Art

Several methods were proposed for estimating the energy consumption on different abstraction levels in literature. Luca Benini first introduced the power state machines in [9]. In this first approach, the PSMs were manually defined by the user and uniquely applied to optimize system-level power management policies.

The approach proposed by Lebreton [50] required to manually instrument the source code of a SystemC/TLM component to accordingly keep updated its power

profile (for instance, on, running, off). An average power consumption is then associated at each power profile.

PowerSim [38] is instead an automatic approach. It requires modifying SystemC Simulation kernel to monitor C++ operators and provide a power consumption estimation based on the performed arithmetic/logic operation. The advantage of this approach is that the model does not have to be changed since a modified SystemC simulation kernel is applied. However, due to the modified SystemC simulation kernel, this approach cannot be integrated into commercial design environments, which have a proprietary SystemC kernel that usually cannot be changed. Additionally, an estimation of communication, especially of abstract models (e. g. TLM), is possible only with restrictions because merely the assignment operator can be used for power annotation. Also in the approach, the Power consumption estimators have to be provided by the user.

The approach proposed by Lorenz [57], which is based on PSM, does not need to instrument the source code of the component in any way, and it is able to provide the most suitable power model by monitoring the input/output ports of a component. This approach relies on the definition of a *protocol state machine* (PrSTM) and of a *power state machine* (PSM). The main task of the PrSM is to monitor the primary input and output ports of an IP, and accordingly trigger state transitions in the PSM. Each power state of the PSM is defined with a power model providing the current power consumption of the monitored IP. In its simplest form, the power model is a constant value. However, it can be more complex such as a mathematical function. Like the previously introduced approach, the approach proposed in [57] is still a manual solution where the user is in charge of defining each component, from the state of PrSTM to the power model in each power state of the PSM.

Industrial tools have been proposed as well. For instance *PrimeTime PX* [67] and *Xilinx Power Estimator* [34] are industrial tools both performing accurate gate-level power analysis. The drawback of these approaches is their execution time. Their execution time is in fact proportional to the complexity and number of memory elements of the design.

6.3 Objectives

To overcome the manual definition of the power state machine for an IP, I presented PsmGen (Power state machine Generator) in this thesis. By analyzing functional and power traces, PsmGen can automatically generate a model describing the power consumption of an IP meanwhile it is performing its functionality at RTL level.

Background

This section reports preliminary definitions that are necessary to understand the proposed approaches.

Definition 6.1. *A **power trace** is a finite sequence $\Delta = \langle \delta_1, \dots \delta_n \rangle$, where δ_i is the dynamic energy consumption of M at simulation instant t_i according to the formula $\delta_i = \frac{1}{2} V_{dd}^2 f C \cdot \alpha(t_i)$, being C the total switched capacitance, V_{dd} the supply voltage, f the clock frequency, and $\alpha(t_i)$ the switching activities of M at time t_i.*

Definition 6.2. *A* ***power state machine*** *is defined as a 7-tuple* $PSM = \langle I, O, S, S_0, E, \lambda, \omega \rangle$*, where* I *is the input alphabet,* O *is the output alphabet,* S *is a set of states,* $S_0 \subseteq S$ *is the set of initial states,* E *is a set of enabling functions* $e : I \to \{0, 1\}$*,* $\lambda : S \times E \to S$ *is the transition function,* $\omega : S \to O$ *is the output function that produces the power consumption.*

Figure 6.1 shows an example of a PSM composed of three power states that characterize the power consumption of the IP when it is turned off, idle and operating with three different constant values (0mW, 15mW and 100mW). Input symbols are associated with the values that can be assumed by the primary inputs of the IP, (i.e, *on*, *ready* and *start*). Enabling functions are represented as guards associated to edges.

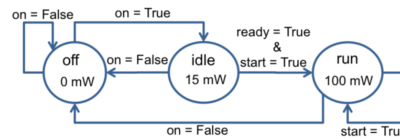


Fig. 6.1. An example of a power state machine.

6.4 Methodology

6.4.1 PsmGen

Figure 6.2 presents an overview of Power State Machine Generator (PSMGen) methodology. It leverages information from several functional and power traces to define the power states describing the power consumption of the IP when it is executing its functionalities. To achieve this goal, the PsmGen performs the following three phases:

1. **Generation of training traces:** The first step is the generation of training traces. In this phase, the IP is simulated by using a set of user-defined test cases. The result of this step is a functional trace and the corresponding power trace for each provided test case.
2. **Generation of PSMs:** In the second step, PsmGen mines temporal assertions from the functional traces. Such assertions are logic formulas that capture the functional behaviors of the IP over time. From them, the states and the transitions of the corresponding PSMs are generated. Then, each state of a PSM is associated to a power consumption by exploiting the reference power traces.
3. **Combination and optimization:** The obtained PSMs are then merged to generate a compact set of PSMs. These PSMs are finally implemented into a SystemC module.

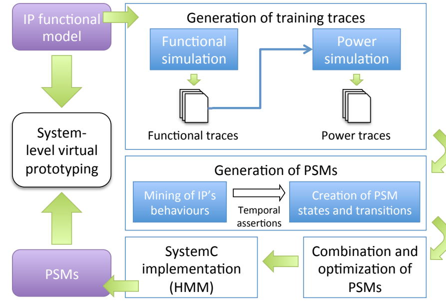


Fig. 6.2. PSMGen methodology overview.

Generation of training traces

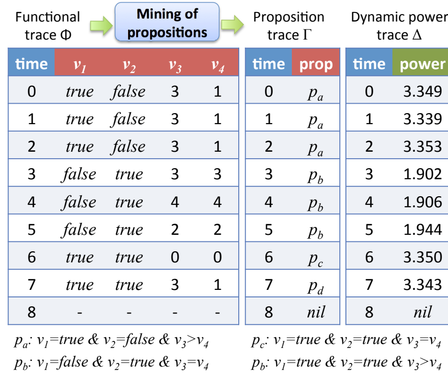
The way by which the functional and power traces are generated is independent from the proposed methodology. However, their quality profoundly impacts on the accuracy of the final PSM. In particular, if the functional traces are unable to cover all the functionalities implemented by the IP, the resulting PSM will be incomplete, thus leading to a wrong power consumption estimation. On the other side, the use of not-accurate power traces for characterizing the energy consumption associated with the states of the PSM will negatively reflect on their precision. To rely upon a high-quality set of training traces, the IP is synthesized to gate level with *DesignCompiler* [27]. Then, a set of functional traces T=$\langle\tau_1, ...\tau_n\rangle$ is generated with *ModelSim* [62] by simulating the IP with the provided test cases. Next, for each functional trace a power trace is generated with *PrimeTime* [67] [1]. At the end of this first phase, a set of functional T=$\langle\tau_1, ...\tau_n\rangle$ and power Δ=$\langle\delta_1, ...\delta_n\rangle$ traces are returned as result.

Throughout this chapter, I directly refer to the i-th (δ_i) trace of Δ as the power trace generated by stimulating the IP with the i-th (τ_i) functional trace of T.

Generation of PSMs

(Step 1 - Mining of functional behaviors of the IP): Given a set of functional traces, the behaviors of the corresponding IP is captured through a set of proposition traces that are automatically generated by a mining procedure. It works in two phases. In the first phase, for each functional trace Φ, the procedure extracts a set of atomic propositions, which hold frequently on Φ, predicating over the PIs and POs of the IP. The atomic propositions represent relations between PIs and POs of the IP that hold in a set of subtraces of Φ. The output of this phase is represented by a matrix m, where the generic element in position $[i, j]$ reports the truth value of the j-th atomic proposition at the i-th time instant of the functional trace. In the second phase, the atomic propositions are combined into a set $Prop$ of propositions, such that in each simulation instant of Φ one and only one of propositions belonging to $Prop$ holds. In particular, a composition procedure generates one proposition from each row of the matrix m by composing in an AND formula all atomic propositions

[1] *PrimeTime* also requires the internal switch activity of an IP to estimate its power consumption. I record such activity during the functional simulation of the IP.



time	v_1	v_2	v_3	v_4
0	*true*	*false*	3	1
1	*true*	*false*	3	1
2	*true*	*false*	3	1
3	*false*	*true*	3	3
4	*false*	*true*	4	4
5	*false*	*true*	2	2
6	*true*	*true*	0	0
7	*true*	*true*	3	1
8	-	-	-	-

time	prop
0	p_a
1	p_a
2	p_a
3	p_b
4	p_b
5	p_b
6	p_c
7	p_d
8	*nil*

time	power
0	3.349
1	3.339
2	3.353
3	1.902
4	1.906
5	1.944
6	3.350
7	3.343
8	*nil*

p_a: v_1=*true* & v_2=*false* & $v_3>v_4$
p_b: v_1=*false* & v_2=*true* & $v_3=v_4$
p_c: v_1=*true* & v_2=*true* & $v_3=v_4$
p_b: v_1=*true* & v_2=*true* & $v_3>v_4$

Fig. 6.3. A functional trace and its proposition and power traces.

that are marked as true. Finally, the proposition trace is obtained by identifying which proposition is true in each simulation instant of the functional trace. The extracted propositions are used to generate temporal assertions that capture the functional behaviors of the IP exposed by the functional trace. Such behaviors are then mapped on states of the PSM.

An example of proposition trace generation is reported in the left side of Fig. 6.3, Given, the functional trace Φ, atomic propositions that frequently hold on it are, for instance, $v_1 = true$, $v_2 = false$, $v_3 > v_4$, etc. After their extraction, the mining procedure generates the proposition trace composed of propositions p_a, p_b, p_c and p_d that hold, respectively, in the intervals $[0, 2]$, $[3, 5]$, $[5, 6]$ and $[6, 7]$.

(Step 2 - Generation of states and transitions): The assumption under the use of a PSM to model the dynamic power of an IP is that there is a correspondence between a specific functional behavior (characterized by a switching activity) and its energy consumption. Thus, to generate a PSM, first the IP's functional behaviors are mined, then the IP's energy consumption is associated to each of them.

Before presenting technical details, let me describe the intuitive idea underlying the proposed approach. By observing a time window between two simulation instants we can observe that the functional behaviors of an IP follow two temporal patterns, namely, *next* and *until*. These two patterns generally alternate when the IP is operating, such that we can observe several consecutive instants where the IP remains in a (sequence of) stable condition(s) from the functional point of view (*until* pattern), followed by an arbitrarily-long sequence of jumps among different states (*next* pattern), before reaching a new stable condition. Moving from one behavioral pattern to another, the energy consumption varies as well. Thus, the basic idea for the automatic generation of a PSM consists of capturing the sequence of *until* and *next* patterns exposed by the IP during its activity and associating to them the corresponding energy consumption.

More formally, given s_i and s_j characterizing two functional states of an IP, the meaning of the *next* and *until* patterns can be described as follows:

- the *next* pattern $s_i X s_j$ corresponds to the LTL temporal assertion $(state = s_i) \rightarrow next(state = s_j)$ specifying that after s_i, at the next instant, the IP moves to s_j;

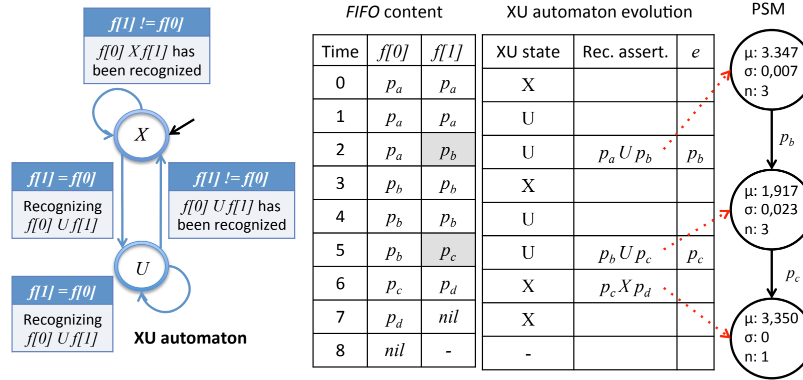


Fig. 6.4. The XU automaton and the exemplification of Algorithm 12.

- the *until* pattern $s_i U s_j$ corresponds to the LTL temporal assertion $(state = s_i)$ *until* $(state = s_j)$ specifying that s_j is preceded by a sequence of time instants where the IP remains in s_i.

According to the mining procedure previously described, we can associate a functional trace with a proposition trace that formalizes the functional behaviors of the IP as a sequence of propositions holding on the different time instants. Thus, to automatically extract the functional behaviors of the IP it is sufficient to search for the *until* and *next* patterns inside the proposition trace. For example, let me consider the proposition trace Γ reported in Figure 6.3. Starting with the time instant 0, Γ exposes the following behaviors: $p_a U p_b$, $p_b U p_c$, and $p_c X p_d$, respectively, in the intervals $[0, 2]$, $[3, 5]$, and $[6, 7]$. Thus, we derive that there are three functional behaviors that must be associated to three power states of the PSM.

Entering in the technical details, to automatically extract the *next* and *until*-based behaviors, and thus defining the corresponding PSM states, I defined the *PSMGenerator* procedure described in Algorithm 12, and the *XU* automaton shown in left side of Fig. 6.4. The inputs of the algorithm are a proposition trace Γ, a dynamic power trace Δ, and a reference to the *PSM* that will be created. The core of *PSMGenerator* is represented by the *XU* automaton. At the beginning, the *XU* automaton is initialized by the function *XU_initialize* (line 2) whose role consists of filling in a *FIFO* data structure f with the two propositions corresponding to instants 0 and 1 of Γ, and setting the current state of the automaton to X. Then, the function *XU_getAssertion* (line 5) iteratively traverses the *XU* automaton till an assertion corresponding to one of the temporal patterns X, U is identified in Γ. During the traversal, each time a transition is taken on the *XU* automaton the *FIFO* scrolls forward by one position on Γ. As soon as a temporal assertion is identified in Γ, the function *XU_getAssertion* returns the triplet $\langle p, start, stop\rangle$. The *start* and *stop* indexes capture the time interval where p holds in Γ. For each mined temporal assertion, the following steps are then performed (lines 7-13):

1. *getPowerAttributes* is called (line 7) to collect the triplet $\langle \mu, \sigma, n\rangle$, where, $n = stop - start + 1$ is the number of time instants where the assertion holds, μ is the mean of the energy consumption values reported in the dynamic power trace Δ in

Algorithm 12 PSM generation.

1: **procedure** PSMGENERATOR($\mathit{\Gamma}, \mathit{\Delta}, PSM$)
2: XU_initialize($\mathit{\Gamma}$)
3: $prev_s = nil$
4: **while** $true$ **do**
5: $\langle p, start, stop \rangle$ = XU_getAssertion($\mathit{\Gamma}$)
6: **if** $p == nil$ **then** break **end if**
7: $\langle \mu, \sigma, n \rangle$ = getPowerAttributes($\mathit{\Delta}, start, stop$)
8: new_s = createPowerState($p, \langle \mu, \sigma, n \rangle$)
9: addState(new_s, PSM)
10: **if** $prev_s \neq nil$ **then**
11: $\langle t, e \rangle$ = createTransition($prev_s, new_s$)
12: addTransition($\langle t, e \rangle, PSM$)
13: **end if**
14: $prev_s = new_s$
15: **end while**
16: **end procedure**

the time interval $[start, stop]$, and σ is their standard deviation. In the following I will refer to the triplet $\langle \mu, \sigma, n \rangle$ with the term *power attributes*.

2. *createPowerState* is called (line 8) to create a new state of the PSM characterized by the temporal assertion p and by the power attributes $\langle \mu, \sigma, n \rangle$. The output function of the state is represented by the constant value μ. The new state is then added to the PSM by the function *addState* (line 9).
3. for all the new states except the first, *createTransition* is called (line 11) to create a transition t between the new state new_s and the previously extracted state $prev_s$. The enabling function e labelling t is represented by the proposition included in element $f[1]$ of the *FIFO* when *XU_getAssertion* stops and recognizes a pattern for $prev_s$. Finally, *addTransition* adds the transition to the PSM (line 12).

To clarify how the *PSMGenerator* procedure works, the right side of Fig. 6.4 exemplifies its application to the proposition trace $\mathit{\Gamma}$ and the power trace $\mathit{\Delta}$ reported in Figure 6.3. The *XU* automaton initially moves from X to U because at time 0 the condition $f[1] = f[0]$ is satisfied. This means we are going to see at least two consecutive instances of the proposition p_a in $\mathit{\Gamma}$ (at times 0 and 1), and thus we are going to recognize an *until* pattern. Then, at time 1 the *FIFO* is scrolled forward, and the automaton remains in U because $f[1] = f[0]$ is still true. Finally, at instant 2 the automaton exits U and comes back to X because $f[1] \neq f[0]$, and consequently *XU_getAssertion* recognizes the assertion $p_a U p_b$ and returns the triplet $\langle p_a U p_b, 0, 2 \rangle$ which corresponds to the first state of the PSM. The state is then populated with the mean and the standard deviation corresponding to the values of the power trace $\mathit{\Delta}$ in the interval [0,2]. At instant 3, the automaton moves again to U starting the capture of a new *until* pattern. The exit condition from U is reached at the instant 5 when $f[1] \neq f[0]$ with the recognition of $p_b U p_c$ in the interval $[3, 5]$. Thus a new state is created and connected to the first state with a transition whose enabling function is p_b, namely the value of $f[1]$ at time 2 when the first state based on the

until pattern was created. In a similar way, the *PSMGenerator* procedure mines the final state corresponding to $p_c X p_d$ and it completes when *nil* is encountered. At the end, the PSM reported on the right of Fig. 6.4 is obtained.

(Step 3 - Simulation of a single PSM): The PSM generated by the previous methodology is in the form of a chain of states, where each state has a unique successor and a unique predecessor. Its simulation is synchronized with the simulation of the corresponding IP by connecting primary inputs and outputs of the IP to the PSM. In this way, at each simulation instant, the values assumed by PIs and POs of the IP are passed as inputs to the PSM, which decides how to move according to the temporal assertion characterizing its current state. When the PSM is in a state s it checks its associated temporal assertion p, whose satisfiability depends on the functional behavior captured through PIs and POs of the IP. If p follows the *until* pattern $p_a U p_b$, the PSM stays in s till p_a is true and it moves to the next state as soon as p_b becomes true. On the contrary, if p follows the *next* pattern $p_a X p_b$, the PSM moves to the next state at the next simulation instant. The enabling function of the transition outgoing from the current state s is satisfied by construction, because, in both cases, it corresponds to the exit condition represented by the activation of proposition p_b.

It is worth noting that if the PSM extracted by Algorithm 12 is stimulated by adopting a functional trace different from the one used for its generation, the power consumption estimated by the PSM may be wrong when it reaches a state characterized by an unexpected temporal assertion. This is due to the fact that the PSM exactly resembles the temporal assertions mined in the proposition trace extracted from the reference functional trace. For example, let us consider a PSM reaching a state s whose temporal assertion is $p_a U p_b$. Then, suppose that when the PSM enter s, p_a is true for a while till p_a becomes finally false, but at that instant p_b continues to remain false as well. In this case, the PSM cannot traverse the outgoing transition of s because p_b is expected. Thus, it remains in s loosing the correct synchronization with the functional trace and generating a wrong power estimation. This limitation is overcome by generating and combining together several PSMs extracted by a set of different functional traces. More the functional traces are representative of all combinations of IP behaviors, lower is the probability of loosing the synchronization between the functional trace and the power simulation. The combination and simulation of different PSMs corresponding to the same IP and their optimization is described in the next sections.

Combination and optimization of PSMs

Given a set of PSMs $\mathcal{P}$, generated for the same IP according to the procedure proposed in the previous section, I propose an automatic methodology to create a reduced and optimized set of PSMs $\mathcal{P}^{opt}$.

The first step of the methodology consists of calling the *simplify* procedure for each PSM in $\mathcal{P}$. The PSMs extracted by Algorithm 12 are in the form of a chain of states. The effect of *simplify* is to shorten such chains, if possible. Thus, for each PSM in $\mathcal{P}$, *simplify* iteratively merges into a single state, a sequence of adjacent states which are "mergeable" from the power point of view. Intuitively, two adjacent states s_i and s_{i+1} are mergeable when the means μ_i and μ_{i+1} of energy consumptions associated respectively to s_i and s_{i+1} are "similar", and their standard deviations σ_i

and σ_{i+1} are "low". For now, the meaning of terms "similar" and "low" is intuitively understandable to capture the notion of mergeable states, while technical details are reported in the section *Quantifying the mergeability of power states* at the bottom of this section.

As reported in the algorithm 12, a state s of a PSM is characterized by the two triplets $\langle p, start, stop\rangle$ and $\langle \mu, \sigma, n\rangle$. When a sequence of adjacent mergeable states $\langle s_i, \ldots, s_{i+j}\rangle$ is found, they are substituted by a new state s_{new} whose triplets $\langle p_{new}, start_{new}, stop_{new}\rangle$ and $\langle \mu_{new}, \sigma_{new}, n_{new}\rangle$ are computed as follows:

- $start_{new} = start_i$; $stop_{new} = stop_{i+j}$; and $n_{new} = n_i + n_{i+1} + \cdots + n_{i+j}$;
- $p_{new} = \{p_i; p_{i+1}; \ldots; p_{i+j}\}$, which represents that first p_i holds in the interval $[start_{new}, stop_i]$, then p_{i+1} immediately follows in the interval $[start_{i+1}, stop_{i+1}]$, and so on till p_{i+j} finally holds in the interval $[start_{i+j}, stop_{new}]$.
- μ_{new} and σ_{new} are, respectively, the mean and the standard deviation of the dynamic energy consumption values reported in the time interval $[start_{new}, stop_{new}]$ of the reference power trace.

Finally, s_{new} is connected with the predecessor s_{i-1} of s_i and the successor s_{i+j+1} of s_{i+j} through the transition outgoing from s_{i-1} and ingoing to s_{i+j+1}. The procedure iteratively executes till no new mergeable state is found. Figure 6.5(a) graphically exemplifies the effect of $simplify$ on a sequence composed of two states.

After the application of the *simplify*, the resulting PSMs are merged into a reduced set $\mathcal{P}'$ by means of the *join* procedure. It works similarly to *simplify* by collapsing mergeable states, but in this case they are not required to be adjacent and they can belong to different PSMs. As a consequence, the triplets characterizing the new state are computed in a different way with respect to *simplify*. In particular:

- $start_{new}$ and $stop_{new}$ become two arrays whose generic element i contains, respectively, the start and stop value of the merged state s_i, while n_{new} becomes the sum of values n of the merged states;
- $p_{new} = \{p_i || p_j || \ldots || p_k\}$, which represents that each time s_{new} is entered one of the assertions characterizing the set of merged states $\{s_i, s_j, \ldots, s_k\}$ is satisfied and then, when it becomes false, s_{new} is left.
- μ_{new} and σ_{new} are, respectively, the mean and the standard deviation of the energy consumption values reported in the time intervals $[start_i, stop_i]$ of the reference power trace for each merged state.

Finally, s_{new} is connected with the predecessors and the successors of all merged states through the transitions, respectively outgoing from the predecessors and ingoing to the successors. The procedure iteratively executes till no new mergeable state is found. Figure 6.5(b) graphically exemplifies the effect of $join$ on two not-adjacent states.

At the end of the *join*, we obtain a new set $\mathcal{P}'$ of more compact PSMs, whose cardinality can be lower than the cardinality of the original set $\mathcal{P}$, if the *join* merged at least two states belonging to different PSMs of $\mathcal{P}$. It is worth noting that, in a particular case, the *join* generates a not-deterministic PSM. This happens when the *join* merges states that are characterized by the same temporal assertions and have the same enabling functions in the respective ingoing transitions and the same enabling functions in the respective outgoing transitions. In this case, when we enter such a collapsed state during simulation, we cannot deterministically decide

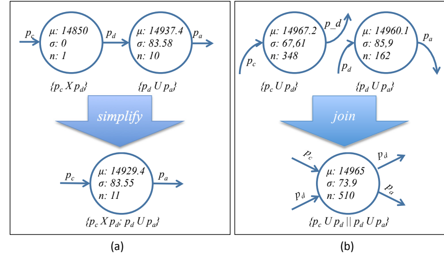


Fig. 6.5. Exemplification of procedures *simplify* and *join*.

the transition to be traversed when the state is left. The simulation of such a not-deterministic PSMs is guaranteed by exploiting a Hidden Markov Model (HMM) based strategy, as described in the section *Simulation of multiple PSMs*.

The final step of the methodology transforms the set of PSMs $\mathcal{P}'$ in a more accurate final set $\mathcal{P}^{opt}$ by acting on power states with a "too high" standard deviation σ. These states has a high probability of being data-dependent from the energy consumption point of view, i.e., when the IP is in one of such states the energy consumption strictly depends on the sequence of data provided to IP's primary inputs. Thus, the use of a constant, represented by the mean μ of energy consumption values, to characterize the power of such data-dependent states is inaccurate. To improve the precision of the power estimation for such a kind of states, we substitute μ in a state s with a function extracted by applying a linear regression between the Hamming distances of consecutive input values exposed in the functional trace and the corresponding values in the power trace. This substitution is applied only for states with an strong linear correlation between Hamming distances of inputs and corresponding values in the power trace, which is a necessary condition for achieving an accurate result from the linear regression [75].

Quantifying the mergeability of power states: The mergeability of power states is evaluated by comparing the power attributes of the target states according to three different cases:

1. comparison between s_i and s_j, where $n_i = n_j = 1$. This happens when both the states are characterized by a temporal assertion respecting the *next* pattern. In this case, we can affirm that s_i and s_j are mergeable when $|\mu_i - \mu_j| < \varepsilon$, where ε is the tolerance fixed by the designer.
2. comparison between two power states s_i and s_j, where $n_i > 1$ and $n_j > 1$. In this case the states are both characterized by the *until* pattern. To identify if it is worth merging the two states, the Welch's t-test [82] is performed on the power attributes $\langle \mu, \sigma, n \rangle$. Such a test is generally used to determine if two sets of data are significantly different from each other with an arbitrarily percentage of error. For lack of space I omit its mathematical formulation, which can be retrieved in [82].
3. comparison between two power states s_i and s_j, where $n_i > 1$ and $n_j = 1$. This happens in the tentative of merging an *until*-based state with a *next*-based state. Similarly to Case 2, I use a different formulation of the t-test to see if the single sample represented by state s_j can be merged with the larger set of values represented by s_i.

Simulation of multiple PSMs

While the basic simulation for one single PSM has been previously presented, this section deal with the concurrent simulation of the complete set of PSMs associated to an IP. A method for resynchronizing the PSMs when unknown behaviors are encountered is presented too.

The simulation of multiple PSMs, obtained after the application of the *simplify* and *join* procedures, has two main differences: (i) the power states can be characterized by more than one assertion, and (ii) some PSMs may be non-deterministic. Concerning the first aspect, when the PSM enters a state s characterized by a sequence of assertions $\{p_i; p_{i+1}; \ldots; p_{i+j}\}$ (as a result of *simplify*), it expects they are satisfied in a cascade fashion one after the other. Thus, first the PSM checks if p_i is satisfied (for an unbounded period of time in case of *until* pattern, or just for one time instant for a *next* pattern). Then, when p_i becomes false (in case of *until*) or simply at the next instant (in case of *next*), the PSM repeats the same analysis for p_{i+1}, and so on till p_{i+j}, when it leaves s according to the enabling function of the outgoing transition. On the contrary, if one of the assertions fails the analysis, i.e., it is not satisfied when expected, it means the PSM has reached an unknown functional behavior. In this case, the simulation continues but the PSM state is not changed till a *resynchronization procedure* allows it jumping to a different state from which a known behavior can be recognized. During the resynchronization period the power estimation provided by the PSM is not reliable.

When a *join* merges a set of states, the resulting state s is characterized by a set of concurrent assertions of the form $\{p_i || p_j || \ldots || p_k\}$. In this case, at least one of these assertions must be satisfied when entering s, otherwise the resynchronization procedure is called. When exactly one assertion is satisfied the simulation proceeds and s is left by traversing the outgoing transition corresponding to the satisfied assertion. For example, in Figure 6.5(b) the merged state is left by traversing the transition labeled with p_d (respectively p_a) when $p_c U p_d$ (respectively $p_d U p_a$) is satisfied. In some cases, the *join* procedure can generate a state where the set of characterizing assertions includes two or more instances of the same assertion. When such identical assertions are satisfied in a state s, a *not-deterministic choice* must be taken to exit s. It is worth noting that a not-deterministic choice could be necessary also at the very beginning of the simulation, when we need to choose the starting state among all the initial states that can be activated. Remember that an IP is associated to a set of PSMs derived from different functional/power traces, thus we have a set of initial states.

To efficiently manage the not-deterministic choices and the resynchronization procedure, I adopted a statistical approach based on a Hidden Markov Model. HMMs are frequently used in temporal pattern recognition, thus they are well suited in our context to predict the state with the highest probability of being the correct choice in case of non-determinism or when a resynchronization is necessary. A HMM is defined as a quintuple $\langle Q, E, A, B, \pi \rangle$, where $Q = \{Q_1, \ldots, Q_m\}$ is set of hidden states; $E = \{E_1, \ldots, E_n\}$ is a set of observable events; $A = \{a_{ij}\}$ and $B = \{b_{jk}\}$ are two matrices such that their elements $a_{ij} = P[x_{t+1} = Q_i | x_t = Q_j]$ (with $1 \leq i, j \leq m$) and $b_{jk} = P[E_k | Q_j]$ (with $1 \leq j \leq m, 1 \leq k \leq n$) represent, respectively, the probability of reaching the state Q_i at time x_{t+1} starting from the state Q_j at time x_t, and the probability of observing the event E_k at state Q_j; π=$\{p_i\}$ is a vector such that its element $p_i = P[x_0 = Q_i]$ (with $1 \leq i \leq m$) represents the probability

of being in state Q_i at time 0. By contextualizing the definition of a HMM to the problem of predicting the next state of a PSM, I implemented a model where Q contains the states of all the generated PSMs and E contains their characterizing assertions. Elements $\{a_{ij}\}$ and $\{b_{ij}\}$ of matrices A and B are then defined according to, respectively, the number of transitions exiting from state i to reach state j, and the number of times the same assertion j has been included (by *join* operations) into the set of assertions characterizing the state i. Finally, the value of the i-th element of the vector π is computed by counting the number of functional traces that have originated a PSM with i as its initial state. Given this formalization, during the simulation of the HMM the next state is chosen by applying the *filtering* approach, i.e., a state-of-the-art procedure to predict the next (hidden) states according to a sequence of observations, which in our case are the functional behaviors captured by the temporal assertions mined on the proposition traces. When a wrong state s is predicted (i.e., the simulation cannot exits s because none of its characterizing assertions is satisfied when expected), the HMM reverts to the last valid state and it follows a different path by fixing to 0 the probability of reaching again the same wrong state in the matrix A. In case all transitions exiting from the current state bring to a wrong state, an unexpected behavior is encountered. This highlights that the functional traces used for generating the PSMs where incomplete with respect the the ones used for the simulation. In this case, the simulation of the model proceeds by remaining in the last valid state till a known behavior is finally recognized in the future instants.

Experimental results

The proposed methodology has been implemented in an automatic tool that generates a SystemC model of the extracted PSMs. Its effectiveness and efficiency have been evaluated by generating the PSMs for the RTL Verilog descriptions of the IPs reported in Table 6.1: a multiplier-accumulator from the Synopsys DesignWare Library, and the implementations of a 1KB RAM memory and the AES and Camellia encryption/decryption algorithms from the Open Core Library. SystemC models of the considered IPs have been obtained from the original Verilog descriptions by using HIFSuite [14]. Table 6.1 reports the number of code lines, the size in bits of PIs and POs, the time required for the gate-level synthesis by using Synopsys DesignCompiler, and finally the number of memory elements in the gate-level netlist.

The results of a first experiment are reported in Table 6.2. Above the dashed line, the results are referred to the simulation of the IPs by using the set of test sequences defined for their functional verification, thus they are assumed to cover the most of IP behaviors We will refer to such a testset with the name *short-TS*. Below the line, a longer set of test sequences has been applied to stimulate the IP's functionality several times with different set of data. We will refer to such a testset with the name *long-TS*. The precise number of test sequences, which correspond to the total length of the functional traces used to extract the IP's behaviors, is shown in Column *TS*. Column *PX* refers to the time required for generating a corresponding set of reference power traces by using Synopsys PrimeTime PX. The time required by our tool for the generation of the PSMs is then shown in Column *PSMs gen.*, while the number of PSMs' states and transitions are reported, respectively, in Columns *States* and *Trans.* Finally, Column *MRE* reports, as a measure of PSMs' accuracy, the mean relative error (MRE) obtained by comparing the power values estimated through the

simulation of the PSMs with respect to the reference values provided by PrimeTime PX. Analyzing the results for each benchmark, we first observed that RAM presents a high correlation between the Hamming distance of two consecutive input data and its energy behavior Thus, the linear regression-based approach adopted in our tool works satisfactory when it relates the RAM's internal switching activity with the power consumption by observing the behaviors of PIs and POs. For this reason, the MRE is very low, even if RAM behaves as a data-dependent IP from the energy consumption point of view (when it operates in the writing modality). MultSum is a data-dependent IP too. Its MRE is a bit higher than RAM, since to effectively capture its functionality by observing PIs and POs it requires to correlate PIs and POs values on an time window wider than the one currently considered by the linear regression mechanism implemented in the tool. On the contrary, AES and Camellia are not data-dependent. However, differently from RAM and MultSim which have no subcomponent, AES and Camellia are composed by a set of subcomponents. In this case, it could be more difficult extracting, in an automatic way, the correlation between the IP's behaviours and the switching activity by observing only the changes in IP's primary inputs and outputs, without a visibility on internal signals connecting the subcomponents. This is due to the fact that the switching activity is distributed among subcomponents that could present power behaviors poorly correlated to each other. This is exactly the case of Camellia. On the other hand, the subcomponents of AES present a stronger correlation, and thus its MRE is sensibly lower than Camellia. As a final consideration on Table 6.2, we observe that, with the exception of Camellia, the MREs below the dashed line are not sensibly improved with respect to their counterparts above the line. This confirms the fact that high-quality PSMs can be generated from functional traces obtained by simulating the IP with the same testbenches adopted for their functional verification.
In conclusion, Table 6.3 reports a performance evaluation and a further accuracy analysis on the PSMs generated by using the *short-TS* set. Columns 2 and 3 show the time required to simulate with the *long-TS* set, for 500,000 instants, respectively, the SystemC model of the IPs (*IP sim.*) and the same IP model connected to the PSMs (*IP+PSMs*). Then, Column 4 shows the simulation overhead due to the presence of the PSMs with respect to simulating the IPs without PSMs. As shown, the overhead ranges between 3% and 26% and it is inversely proportional to the complexity of the IP. An even more significant fact is observed by comparing Column *IP sim.* of Table 6.3 with the values reported in Column *PX* under the dashed line of Table 6.2. This shows that estimating the power values by simulating the PSMs is up to two orders of magnitude faster than using PrimeTime PX. This speed-up is not paid in terms of accuracy, as shown by the last two columns of Table 6.3, which report the MRE and the percentage of wrong-state predictions obtained by simulating the PSMs, generated from the *short-TS* testset, with the *long-TS* testset.

Conclusions

The thesis presented a methodology for the automatic generation of PSMs by adopting an approach based on (i) dynamic mining of temporal assertions to extract the IP's functional behaviours from a set of functional traces, and (ii) a calibration process to extract the associated power behaviours from a corresponding set of references power traces. Finally, a Markov model was defined to implement a SystemC simulatable model of the PSMs. The power estimation obtained by a system-level

IP	*Lines*	*PIs*	*POs*	*Syn. time (s.)*	*Memory elements*
RAM	101	44	32	140.2	8192
MultSum	45	49	32	18.8	225
AES	1089	260	129	42.6	670
Camellia	777	262	131	75.2	397

Table 6.1. Characteristics of benchmarks.

IP	*TS*	*PX (s.)*	*PSMs gen. (s.)*	*States*	*Trans.*	*MRE*
RAM	34130	169.0	1.2	9	18	0.30 %
MultSum	12002	19.5	0.6	2	2	4.03 %
AES	16504	144.8	0.7	5	7	3.45 %
Camellia	78004	74.5	5.7	5	10	32.66 %
RAM	500000	5316.7	20.1	9	18	0.29 %
MultSum	500000	750.1	22.6	3	4	3.27 %
AES	500000	3626.0	115.6	13	29	3.09 %
Camellia	500000	2699.0	221.2	5	11	32.64 %

Table 6.2. Characteristics of the generated PSMs.

IP	*IP sim. (s.)*	*IP+PSMs (s.)*	*Overhead*	*MRE*	*WSP*
RAM	13.8	17.5	26.4%	0.29%	0%
MultSum	20.4	24.2	18.4%	3.97%	0%
AES	93.4	98.7	5.6%	3.11%	0%
Camellia	277.1	286.9	3.5%	32.64%	20%

Table 6.3. Simulation times and accuracy evaluation.

simulation of the automatically generated PSMs is up to two orders of magnitude faster than running a state-of-the-art gate-level power simulator like PrimeTime PX without a significant loss of accuracy for all IPs, but Camellia, which is composed by a set of subcomponents whose power behaviours are low correlated to each other. To mitigate the limitation highlighted by Camellia, I foresee, as future works, the automatic generation of a power model based on hierarchical PSMs that distinguishes among IP subcomponents.

Part IV

Conclusions

7

Conclusions and Future works

In the context of system-level virtual prototyping, this thesis presented techniques and tools addressing the creations of a unified SoC verification environment making automatic the extraction of functional and extra-functional properties that characterize the behaviors of a SoC. The outcome is represented by a set of different approaches to automatically: generate invariants and assertions describing the functionalities of a design, detect the security vulnerability of firmware, and create power models describing the power consumption of a design. In detail, the main contributions presented in this thesis were:

1. The first GPU-based invariant miners in literature. By exploiting GPUs, both the presented approaches greatly reduced the execution time with respect to existing techniques, without affecting the accuracy of the analysis. Time-window invariant mining is a new concept that certainly must be further investigated and improved. Nevertheless, the proposed invariant miners are the first introducing time-window invariant and exploiting GPUs.

2. An assertion miner exploiting user-defined templates to mine temporal assertions. By using user-defined templates, we have the advantage to precisely define what behaviors the system is supposed to implement. This allows us to more clearly detect if the designed functionalities are exhibited during the simulation of the SoC. Moreover, a fault-detection based strategy was applied to evaluate the coverage of the mined assertions and provide to the user with a compact set of assertions.

3. The first security vulnerability miner that exploits concolic testing and model counting to automatically generate assertions pinpointing corner cases that could hide security vulnerabilities of a firmware running in a hardware platform.

4. Finally, the first power state machines generator which combines the analysis of power and functional traces to automatically generate a model describing the power consumption variation of an IP meanwhile it is performing its functionality at RTL level.

However, among all of these contributions, the main lesson learned by developing this thesis come from the time spent to automatically generate assertions pinpointing

security vulnerabilities. I came up with a solution assuring several advantages that push me to continue extending the developed approaches in the near future:

- All the approaches for dynamically mining invariants and assertions are heavily affected by the quality of the analyzed execution traces. On the other hand, symbolic simulation is an excellent strategy to generate high-coverage test cases for complex programs automatically. Therefore, a first possible extension of this thesis is the extraction of invariants and assertions from execution traces symbolically generated instead of purely relying upon test cases provided by the user. Several academic and industrial (VCS Synopsys [1]) tools exist to turn a Verilog/VHDL design into a C or binary code program. A C or binary program symbolic engine [18, 72] can be consequently applied to generate high-quality execution traces. The afterward mined invariants and assertions are more confidently true in design under verification since they are extracted from traces generated by traversing as many design's execution branches as possible.

- Secondly, we can exploit the symbolic simulation of a design under verification to generate a symbolic tree as described in the approach $DOVE$. The mining of invariants and assertions through the analysis of a symbolic tree can be more efficient rather than only considering a large set of purely concrete execution traces. Afterwards, the mined assertions can be respectively evaluated with the fault-based evaluation strategy defined in $A{-}TEAM$, and further "stressed" with $Mangrove$ on several simulation traces with millions of clock cycles.

8

Published contributions

The work carried on to develop this thesis led to a total of IX publications. Hereby are listed how the different which publications contributed to the Chapters in the Part II of this thesis.

The methodologies for invariant mining discussed in Chapter 3, have been presented in:

- Danese, A., Piccolboni, L., and Pravadelli, G.,
  "**A parallelizable approach for mining likely invariants**"
  in Proceedings of the ACM/IEEE 10th International Conference on Hardware/Software Codesign and System Synthesis (CODES 2015), pp. 193-201
- Bombieri, N., Busato, F., Danese, A., Piccolboni, L., and Pravadelli, G.,
  "**Exploiting GPU architectures for dynamic invariant mining**"
  in Computer Design of the 33rd IEEE International Conference on Computer Design (ICCD 2015), pp. 192-195

The methodologies for temporal assertion mining discussed in Chapter 4, have been presented in:

- Danese, A., Ghasempouri, T., and Pravadelli, G.,
  "**Automatic extraction of assertions from execution traces of behavioural models**"
  in Proceedings of the ACM/IEEE conference on Design, Automation & Test in Europe (DATE 2015) pp. 67-72
- Danese, A., Filini, F., and Pravadelli, G,
  "**A time-window based approach for dynamic assertions mining on control signals**"
  In Very Large Scale Integration (VLSI-SoC), 2015 IFIP/IEEE International Conference on (VLSI-SoC 2015) pp. 246-251
- Danese, A., Mocci, J.,and Pravadelli, G.
  "**Fault model qualification by assertion mining**" in Proceedings of the IEEE 17th Latin American Test Symposium (LATS 2016) pp. 45-50
- Danese, A., Dalla Riva, N., and Pravadelli, G.,
  "**A-TEAM: Automatic template-base assertion miner**"
  in Proceedings of the ACM/IEEE conference on Design, Automation Conference (DAC 2017) pp. 1-6

The methodologies for security vulnerability detection discussed in Chapter 5, have been presented in:

- Danese, A., Bertacco, V., and Pravadelli, G.,
  "**Work-in-Progress: DOVE:Pinpointing firmware security vulnerabilities via symbolic control flow assertion mining**"
  in Proceedings of the ACM/IEEE 12th International Conference on Hardware/Software Codesign and System Synthesis (CODES 2017), pp. (to appear)
- Danese, A., Bertacco, V., and Pravadelli, G.,
  "**Symbolic assertion mining for security validation**"
  in Proceedings of the ACM/IEEE conference on Design, Automation & Test in Europe (DATE 2018), pp. (to appear)

The methodologies for the automatic generation of power state machines discussed in Chapter 6, have been presented in:

- Danese, A., Zandona, I., and Pravadelli, G.,
  "**Automatic generation of power state machines through dynamic mining of temporal assertions**
  in Proceedings of the ACM/IEEE conference on Design, Automation & Test in Europe (DATE 2016), pp. 606-611

References

1. Vcs: https://www.synopsys.com/verification/simulation/vcs.html.
2. Y Abarbanel, I Beer, L Glushovsky, S Keidar, and Y Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *Proc. of CAV*, pages 538–542, 2000.
3. Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.
4. Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proc. of ACM POPL*, pages 4–16, 2002.
5. Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.
6. Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R Tuttle, and Vincent Zimmer. Symbolic execution for bios security. In *Proc. of USENIX WOOT*, 2015.
7. Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of USENIX Annual Technical Conference*, pages 41–46, 2005.
8. L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Trans. on VLSI*, 8(3):299–316, 2000.
9. Luca Benini, Robin Hodgson, and Polly Siegel. System-level power estimation and optimization. In *Proceedings of the 1998 international symposium on Low power electronics and design*, pages 173–178. ACM, 1998.
10. Luca Benini, Robin Hodgson, and Polly Siegel. System-level power estimation and optimization. In *Proc. of IEEE ISLPED*, pages 173–178, 1998.
11. R. Bergamaschi and Y.W. Jiang. State-based power analysis for systems-on-chip. In *Proc. of ACM/IEEE DAC*, pages 638–641, 2003.
12. Janick Bergeron. *Writing testbenches: functional verification of HDL models*. Springer Science & Business Media, 2012.
13. Michele Bertasi, Giuseppe Di Guglielmo, and Graziano Pravadelli. Automatic generation of compact formal properties for effective error detection. In *Proc. of ACM/IEEE CODES+ISSS*, pages 1–10, 2013.
14. Nicola Bombieri, Giuseppe Di Guglielmo, Michele Ferrari, Franco Fummi, Graziano Pravadelli, Francesco Stefanni, and Alessandro Venturelli. Hifsuite:

Tools for hdl code conversion and manipulation. *EURASIP J. Embedded Syst.*, pages 4:1–4:20, 2010.
15. Marco Bonato, Giuseppe Di Guglielmo, Masahiro Fujita, Franco Fummi, and Graziano Pravadelli. Dynamic property mining for embedded software. In *Proc. of ACM/IEEE CODES+ISSS*, pages 187–196, 2012.
16. Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *J. of Object Technology*, 5(5):31–58, 2006.
17. BugScope. `http://www.atrenta.com/about-bugscope.htm5`.
18. Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of USENIX OSDI*, volume 8, pages 209–224, 2008.
19. Xueqi Cheng and Michael S Hsiao. Simulation-directed invariant mining for software verification. In *Proceedings of the conference on Design, automation and test in Europe*, pages 682–687. ACM, 2008.
20. Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.
21. CodeSurfer. `http://www.grammatech.com/research/technologies/codesurfer`.
22. Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proc. of ACM/IEEE ICSE*, pages 422–431, 2005.
23. CUDA. `http://docs.nvidia.com/cuda`.
24. Daikon. `http://plse.cs.washington.edu/daikon/pubs`.
25. Alessandro Danese, Tara Ghasempouri, and Graziano Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *Proc. of ACM/IEEE DATE*, pages 1–6, 2015.
26. Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proc. of USENIX Security*, pages 463–478, 2013.
27. DesignCompiler. `http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx/`.
28. Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. System management mode design and security issues. *IT Defense*, February 2010.
29. Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proc. of ACM/IEEE ICSE*, pages 411–420, 1999.
30. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 411 –420, 1999.
31. Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proc. of WODA*, pages 24–27, 2003.
32. Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.
33. Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

34. Xilinx Power Estimator. https://www.xilinx.com/products/technology/power/xpe.html.
35. Andrea Fedeli, Franco Fummi, and Graziano Pravadelli. Properties incompleteness evaluation by functional verification. *IEEE Trans. Comput.*, 56(4):528–544, 2007.
36. Antonio Filieri, Corina S Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proc. of IEEE ICSE*, pages 622–631, 2013.
37. Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Inf. Process. Lett.*, 77(2-4):97–108, 2001.
38. Marco Giammarini, Simone Orcioni, and Massimo Conti. Powersim: power estimation with systemc. In *Solutions on Embedded Systems*, pages 285–300. Springer, 2011.
39. Sudheendra Hangal and Monica S Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM, 2002.
40. Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *Proc. of ACM/IEEE DAC*, pages 775–778, 2005.
41. R. Hastings and B. Joyce. Joyce. purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter USENIX Conference*, 1991.
42. S. Hertz, D. Sheridan, and S. Vasudevan. Mining hardware assertion with guidance from static analysis. *IEEE Trans. on CAD*, 32(6):952–965, 2013.
43. Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.
44. Jasper. http://www.jasper-da.com.
45. Corey Kallenberg, John Butterworth, Xeno Kovah, and C Cornwell. Defeating signed bios enforcement. Technical report, MITRE, 2013.
46. Corey Kallenberg, Sam Cornwell, Xeno Kovah, and John Butterworth. Setup for failure: defeating secure boot. In *Proc. of SyScan*, 2014.
47. Corey Kallenberg and Xeno Kovah. *How Many Million BIOSes Would you Like to Infect?* CanSecWest, 2015.
48. Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with ddt. In *Proc. of USENIX Annual Technical Conference*, 2010.
49. H. Lebreton and P. Vivet. Power modeling in SystemC at transaction level, application to a DVFS architecture. In *Proc. of IEEE ISVLSI*, pages 463–466, 2008.
50. Hugo Lebreton and Pascal Vivet. Power modeling in systemc at transaction level, application to a dvfs architecture. In *Symposium on VLSI, 2008. ISVLSI'08. IEEE Computer Society Annual*, pages 463–466. IEEE, 2008.
51. Peng Li and John Regehr. T-check: bug finding for sensor networks. In *Proc. of ACM/IEEE IPSN*, pages 174–185, 2010.
52. Wenchao Li, Alessandro Forin, and Sanjit A Seshia. Scalable specification mining for verification and diagnosis. In *Proc. of ACM/IEEE CAD*, pages 755–760, 2010.
53. Invariant list of Daikon. http://plse.cs.washington.edu/daikon/download/doc/daikon.html\#Invariant-list.

54. Lingyi Liu, Chen-Hsuan Lin, and Shobha Vasudevan. Word level feature discovery to enhance quality of assertion mining. In *Proc. of IEEE ICCAD*, pages 210–217, 2012.
55. Lingyi Liu and Shobha Vasudevan. Automatic generation of system level assertions from transaction level models. *J. Electronic Testing*, 29(5):669–684, 2013.
56. D. Lorenz, K. Gruettner, and W. Nebel. Data-and state-dependent power characterisation and simulation of black-box RTL IP components at system level. In *Proc. of Euromicro DSD*, pages 129–136, 2014.
57. Daniel Lorenz, Kim Grüttner, and Wolfgang Nebel. Data-and state-dependent power characterisation and simulation of black-box rtl ip components at system level. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 129–136. IEEE, 2014.
58. Daniel Lorenz, Philipp A. Hartmann, Kim Grttner, and Wolfgang Nebel. Non-invasive power simulation at system-level with SystemC. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 7606 of *LNCS*, pages 21–31. Springer, 2013.
59. Justin Luitjens. CUDA pro tip: Increase performance with vectorized memory access. `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/`, December 2013.
60. Roongko Doong Marat Boshernitsan and Alberto Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. of ISSTA*, pages 169–180, 2006.
61. Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701. ACM, 2016.
62. ModelSim. `https://www.mentor.com/products/fpga/model`.
63. Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proc. of ACM FSE*, pages 11–20, 2002.
64. OpenCL. `http://www.khronos.org/opencl`.
65. Wiebe R Pestman. *Mathematical statistics: an introduction*, volume 1. Walter de Gruyter, 1998.
66. MAfee 2017 Threats Predictions. `https://www.mcafee.com/us/resources/reports/rp-threats-predictions-2017.pdf`.
67. PrimeTime PX. `http://www.synopsys.com/Tools/Implementation/SignOff/Pages/PrimeTime.aspx`.
68. Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proc. of ACM/IEEE IPSN*, pages 186–196, 2010.
69. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *ACM Trans. on Computer Systems*, pages 391–411, 1997.
70. S. Schurmans, Diandian Zhang, D. Auras, R. Leupers, G. Ascheid, Xiaotao Chen, and Lun Wang. Creation of esl power models for communication architectures using automatic calibration. In *Proc. of ACM/IEEE DAC*, 2013.

71. D Sheridan, L Liu, H Kim, and Shobha Vasudevan. A coverage guided mining approach for automatic generation of succinct assertions. In *Proc. of IEEE VLSI Design*, pages 68–73, 2014.
72. Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.
73. Jiří Slabỳ, Jan Strejček, and Marek Trtík. Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 207–221. Springer, 2012.
74. Monica S. Lam Sudheendra Hangal. Tracking down software bugs using automatic anomaly detection. In *Proc. of ACM/IEEE ICSE*, pages 291–301, 2002.
75. T.D.V. Swinscow and Campbell M.J. *Statistics at square one.* BMJ Publishing Group, 2009.
76. Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering likely method specifications. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *LNCS*, pages 717–736. Springer, 2006.
77. Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering likely method specifications. In *ICFEM*, pages 717–736. Springer, 2006.
78. Shobha Vasudevan, David Sheridan, and V. Athavale. Automatic generation of assertions from system level design using data mining. In *Proc. of IEEE MEMOCODE*, pages 191–200, 2011.
79. Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the conference on Design, automation and test in Europe*, pages 626–629. European Design and Automation Association, 2010.
80. Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. Goldmine: automatic assertion generation using data mining and static analysis. In *Proc. of ACM/IEEE DATE*, pages 626–629, 2010.
81. Vijay V Vazirani. *Approximation algorithms.* Springer Science & Business Media, 2013.
82. B. L. Welch. The generalization of 'student's' problem when several different population variances are involved. *Biometrika*, 34(1/2):pp. 28–35, 1947.