UNIVERSITY OF BREMEN

DOCTORAL THESIS

Evidence-Oriented Tracing and Verification
The Declaration of Timeprints

Rehab MASSOUD

A thesis submitted in partial fulfillment of the requirements
for the Doctor of Engineering (Dr.-Ing) degree

at the

University of Bremen
Dept. of Computer Science and Mathematics

December 22, 2021

Doctoral Thesis

Evidence-Oriented Tracing and Verification
The Declaration of Timeprints

Thesis Author:
Rehab MASSOUD

Examination Committee

First Examiner:
Prof. Dr. Rolf DRECHSLER

Second Examiner:
Prof. Dr. Valeria BERTACCO

A thesis submitted in partial fulfillment of the requirements
for the Doctor of Engineering (Dr.-Ing) degree

at the

University of Bremen
Dept. of Computer Science and Mathematics

Date of the doctoral colloquium: December 22, 2021

UNIVERSITY OF BREMEN

Abstract

Evidence-Oriented Tracing and Verification
The Declaration of Timeprints

by Rehab MASSOUD

With the unprecedented jump in reliance on digital systems, the expectations about their reliability, safety and correctness rise as well. The complexity of these systems is increasing, while the means for ensuring and checking their correctness and safety are lagging behind. The analysis and inspection of potential root-causes (behind such failures) focus on finding bugs, noncompliance or shortage within the design and/or process artifacts. This analysis also uses coarse traces (e.g. telemetry or diagnostics) obtained from in-field execution. Although having accurate and independent traces of the system's in-field operation would be very useful in these situations; such traces are not available, because they imply prohibitive amounts of data to store and process.

From user's perspective, transparently and continuously logged traces are needed as evidence of in-field execution. Evident traces need to be independent (not just self reporting), accurate (not coarse, and triggered automatically by the physical execution taking place), and affordable (light: so that they can be efficiently logged, stored and processed). To reach this, the thesis proposes periodically logging a light-weight temporal trace-print from physical signal execution, which we call the Timeprints-trace. Such trace is meant to act as a signature of the period of execution it summarizes. From these signatures, some details can be checked and properties about them can be formally verified. We call this form of verification: "Evidence-Oriented", in which we formally check offline (and on-demand), and prove properties of what happened. The *Timeprints* open the door for a wide range of deployment-phase accurate timing-properties verification; some of which are also shown in the thesis.

Acknowledgements

In our time, the scientific and academic community is guarded by invisible gates. Someone from the community has to open the door and show you the way to join. I knocked few doors, until one was opened to me by Rolf. Rolf has built a competent and supportive research group, which welcomes students and guides them through their journey of becoming scientists. He also gave me and his other students the freedom needed to grow and learn. At the same time he was always there, holding the groups together, maintaining resources for all members, and always responsive and present whenever he is called. Thanking my professor sounds to me similar in a sense to thanking my parents, for doing their role in the best they can, and for always being there for us so we can be. Thanks!

I'm also thankful for the generous support and mentorship I received from Valeria. She gave me an example to look-up at on how to give consistent and extended support even to remote students, and how to handle communication and set targets successfully. She invited me to attend her research group meetings all over the past two years, which made a big difference during the pandemic and helped me stay on track, following up the recent advances on the field while writing and editing my thesis.

I also want to gratefully thank Hoang M. Le for looking at the very early fragments of this thesis and helping me with his comments to get it in shape. I consider him a real contributor with his invaluable comments and feedback; not only to the thesis but also to the ideas which developed through our discussions, and for even the greedy algorithm idea and code.

Early conversations with Daniel Grosse and Jannis Stope also contributed shaping the ideas at early stage to what they become. I also thank them for providing support and advice regarding the paper writing, academic research and process set-up.

Scientific-wise also, I'd like to thank Peter Chini, Roland Meyer, and Prakash Saivasan for their contribution in getting the first elegant mathematical formulation of the logs and Timeprints. I also thank Wen-ling for her time and discussions about delays formulation.

In my journey, I had support from so many people –outside the university– that it would require writing many books to thank them all. In my first year, my dear mother Nahed passed away. She is the one whom supported me the most; I owe her the means of mostly all my strength. She planted the tree and was happy with me starting my PhD journey and was sure I will finish. Her belief in me was the main

motivation which kept me determined to reach the finish line; even when things sounded so hard. I want to thanks my sisters: Mona, Mariam and Amira and their kids, Youssef, Adam, Talin and the new born Lina, and my father Hosni. They were always supportive and there for me. Also my extended family, which supported me in various ways I cannot count.

The PhD journey is never an easy one, and in my experience, it would have never been possible to complete without such support from family, friends, colleagues, and professors. Most of this thesis was written in 2020, where the pandemic hit many hard. My family and friends supported me unconditionally, and I owe them more than the here written thanks. I consider them real contributors in all my work, who didn't ask for their names to be mentioned. They took my calls (real face to face contact was not possible in many cases due to the pandemic), calmed my frustration and were always there for me. I feel indebted to mention friends' names. Asmaa, Doria, Fatma, Noha, Heba and Heba (they know themselves), Eman, Rania, Omar, Eman, Maha, Etaf, Helia, Christoph, Zoe, Magda and Lisset. I'd also mention the names of my colleagues who supported me: Nabila, Frank, Fritjof, Timo, Kenneth, Tim, Vladimir, Amr, Mehran, Saeideh, Alireza, Aron, Hassan, Khushboo and Nils. Also I'd love to thank deeply: Regine, Uwe, Birthe, Ingrid, Lisa and Kiki for all their kindness, help and support.

I'd also like to thank Mary Sanders and the international office team at the university of Bremen. Their support via face-to-face, over the phone advice and through workshops, were major support and great enabler for me and other PhD students. They organized novel workshops that were of extreme help to me and many other PhD students. This acknowledgment part of every thesis I read was always my favorite part; and it was also for me the most emotional one to write.

Contents

List of Figures

List of Tables

List of Abbreviations

CAT	**C**ycle **A**ccurate **T**racing
CPS	**C**yber **P**hysical **S**ystem
EOT	**E**vidence **O**riented **T**racing
EOV	**E**vidence **O**riented **V**erification
RT	**R**eal **T**ime
RTS	**R**eal **T**ime **S**ystem
RV	**R**un-time **V**erification
SoC	**S**ystem **o**n **C**hip
TBV	**T**imeprints **B**ased **V**erification
TC	**T**race **C**ycle
TP	**T**ime**p**rint

Dedicated to my mother…

Chapter 1

Introduction

Our world today is witnessing unprecedented dependence on digital systems, and more specifically on Cyber-Physical Systems (CPSs)[1]. CPSs are digital-systems that interact with their surrounding physical environment. They have both cyber and physical parts, working together. Hardware and software components (constituting the *cyber* part) are working in synergy to control the system's interaction with the *physical* environment. While the reliability of the physical components is relatively straight forward to check and assess (according to methods and practices that have been stable for decades), the verification of cyber components is relatively new, and more tricky and complex. The rising speed, size and complexity of hardware and software themselves make their verification challenging; let alone verifying their interaction with one another and with their environment.

At the electronics industry side, the complexity growth is taking a new form. The transistor feature size has met a saturation-point, in terms of Moore's law projections [122, 175]. While this seems, in the first sight, to entail saturation in the complexity rise, as no more transistors can be fit on the same silicon area, the complexity however did not stop growing. On the contrary, chips are now being stuffed together in 2.5D, 3D, and designed as composable building blocks (*chiplets*) [86, 104]. This enabled even larger chips size than whatever existed before [42]. At the software side, millions of lines-of-code is becoming typical in such systems [50]. Hence, complexity, on all levels, is still *exponentially* on the rise.

In theory, Formal verification methods can "fully" assess the safety and design correctness of systems. However, in practice, advancements in formal verification methods, techniques and technologies are unable to cope with the increasing complexity. This is illustrated by many recent fatalities, in critical areas, where several formal methods are already applied. For example, in 2018 and 2019, two Boeing airplanes crashed. In both crashes, an automated safety function (within a system

[1] In 2021, ARM® (the processor IP company) announced on their website that there are already more than 125 billion devices shipped, which contain at least one ARM core [17].

called MCAS), which was designed to compensate for the impact of a new structural design, was cited as the most probable technical cause of these crashes [126]. The automated MCAS (Maneuvering Characterization Augmentation System) used un-reliable sensors readings, with which it has overridden the pilots' inputs, causing consecutive automated nose-down commands. In the automotive domain, the first *"accident"* in which an *"under-test"* automated vehicle *"killed"* [2] a pedestrian, was reported in March 2018 [139]. Several other autonomous vehicle accidents [52, 82] were also reported. While the industry is trying to push the technology to new frontiers, it has to handle the associated rising complexity, so as not to lead such disasters.

In order to understand the limitations of the existing verification methodologies and techniques, we show some of their major categories with their associated properties in the following Table 1.1:

Feature \ Method	Testing	Simulation	Model-Checking	Run-Time Verification
What gets Verified?	Trace from System under Test	Trace from model	Model vs Specifications	Trace vs Specification
Observability	Test-points at check time	All modeled signals can be recorded	Depends on abstraction	Available traces
Coverage	Limited by test-cases	Limited by scenarios	All traces of the model, checked vs specs	One finite-trace from reality vs specs
When is it used?	After design	After design	After design	At deployment

TABLE 1.1: Testing, Simulation, Model-Checking and Run-Time Verification

Testing and simulation are similar in that the system (or its model) is run (executed) with exactly one configuration from a start point to a specified point. On the contrary, model-checking checks every possible execution on of a model. While simulation and model-checking use models, testing and run-time verification can be run over the actual physical system. Formal verification techniques (e.g. model-checking) are applied to abstract designs; rather than the real systems. Verifying that the "reality" (the actual conditions and implementation) meet the formally verified properties and specifications was not considered under the umbrella of "Formal Verification" until the introduction of Run-Time verification techniques [21].

[2] Italics are used here to emphasis the controversial usage of words; for example: to accuse a semi-automated machine with committing a crime is at least controversial. The author argues that humans developing/designing the learning approach of these systems (even fully automated self-learning machines) should be held responsible for the actions of what they have made.

Various levels of abstraction are used to encode real world problems (artifacts and specifications) using mathematical representations, which can be processed by a computer. Formal methods use models/abstractions to express both the system, and the properties the system should fulfill allover its term of operation. These methods then apply logical rules and mathematical algorithms to check that the system meets its specification in every possible case. In this context, the more accurate and realistic the models are, the more complex, computationally-expensive and hence challenging, the verification-task becomes. Moreover, the increase of a systems' complexity, size and features raise the computational effort to even more prohibitive levels.

Many techniques are used to work around complexity. For example, Abstraction-refinement start with simplified high level model/abstraction. Then, the model is refined, i.e. details retrieved later, as needed [48]. However, when it comes to checking the realized system's resulting behavior (rather than a model's simulation), it is not always possible (e.g. sporadic faults) to repeat the execution exactly to retrieve details which have been abstracted-away. Statistical model-checking was introduced as a compromise between speed of testing and the thoroughness of model-checking [3]. To avoid the model-checking state-space explosion problem, symbolic model-checking [37] was introduced. Symbolic simulation, for example as in [31], constitute another class of techniques used to increase coverage of simulation. Similar to abstraction refinement, symbolic and statistical model-checking are also limited to design time. In conclusion, many methods, which scale well and work around complexity for design models, cannot handle complexities associated with tracing/verifying their physical realization.

Notice that, even if we assume availability of enough computational resources to handle complexity, the ability to reflect reality by models is still inherently limited. For today's digital signals, if we want our samples to be perfect, tracing a single binary signal value will incur the generation many Gigabytes of data per second. The increasing operating frequencies and the corresponding generated data, make the detailed tracing of operations a completely non-practical approach.

On one hand, most formal verification methods, e.g. model-checking are applied to design artifacts, which are typical system abstractions. On the other hand, trace-based verification (i.e. verifying properties traces resulting from test or simulation runs) is more focused on checking actual or simulated execution traces; rather than the abstraction which generated them.

Run-Time Verification (RV), is one methodology that started to shift the formal verification usage to run-time traces. RV checks the actual system execution at run-time, not the design producing that single trace of execution. Doing the check at

run-time, avoids logging and processing a prohibitive amount of data. RV checks the execution on the fly, against some pre-specified properties. This means RV is strictly limited to checking these pre-specified properties. In the case of unexpected problems, sporadic behaviors, or in any case where other properties –rather than these pre-specified properties– need to be checked, only the pre-specified check-results are available as a trace to support further analysis.

Tracing and Verifying CPS Things get more challenging when we speak about CPS. These are systems that interact in real-time with their environment. In this case, not only the *what* (which events have happened) is relevant, but also the *when* (which express the exact timing of these events). This environment reaction time, becomes another artifact which needs to be measured; besides other functional data. Measurements of real world values are taken by sampling, triggered usually with a clock, and the measured values are quantized. Logging detailed tracing samples continuously from in-field operations is an unacceptable burden. Hence, again, either coarse samples or only abstracted version of the measurements are recorded. Notice also that mathematical formulas used to abstract and describe continuous behaviors are approximate as well.

The Proposed Solution To extend our capabilities to enable reasoning about unexpected behaviors and problems root-causes, we need a transparent form of tracing that can still keep some details of execution, while not tied to specific checks. Ideally this tracing shall be efficient yet accurate; i.e. provides the finest possible level of detail needed to inspect a wide range of unexplored properties. This is what *Timeprints* –as traces– try to provide: transparency, accuracy and compactness; to unlock as much of execution details whenever investigation is needed.

To enable an accurate tracing of real executions, and to better scale the advantages of formal methods to run-time execution traces, this thesis introduces a new methodology: *Evidence-Oriented Tracing (EOT)* and a novel technique: *Timeprints-Based Verification (TBV)*. Similar to run-time verification, evidence-oriented tracing checks the run-time behavior of an already-implemented physical artifact while it is being executed in field. However, unlike run-time verification, the check itself is not necessarily done at run-time. EOT depends on logging what we call *Timeprints*, that summarize the continuous execution in light weight signatures. These signatures are of acceptable-size to log, store and process. Evidence-Oriented Verification is not limited to pre-deployment defined specifications. Rather, by using formal reconstruction, Timeprints can accurately and transparently retrieve other execution

details. Timeprints are more independent from specs defined or expected before deployment, which makes them capable of capturing sporadic and un-expected behaviors.

The price paid is the computation time and resources needed for reconstructing the needed details, from the logged Timeprints. However, this effort needs to be conducted only on-demand and over a finite time-window of interest, for which a wide range of specs and checks can be carried on. Timeprints-based verification –in the form presented in this thesis– is focused on post-mortem root-cause analysis of unexpected failures. Hence, it does not aim at replacing RV but rather complements it. The post-mortem analysis of failures is already highly needed for today's CPS and Autonomous systems monitoring. It can also be used to facilitate many debugging and verification activities.

Such accurate tracing and monitoring enables not only learning from such un-expected failures; but furthermore can act as evidence to assign liability at the level of modules within complex systems. With the pursued level of transparency, safer-/better designs will have more chances to be valued and acknowledged. Also, the public trust in complex systems can be built on accurate and transparent measures.

The rest of this chapter introduces the context and terminology needed to present Evidence-Oriented Tracing and Timeprints-Based Verification. This is followed by a brief description of the main contribution and the key ideas behind it. The chapter also gives an overview of the thesis structure in terms of its chapters and how they relate to each other.

1.1 Context and Basic Terminology

As mentioned before, the increasing number of systems, signals, their frequencies at the various fronts of complex, autonomous, distributed and interactive systems, come with all their burdens to the verification community. This community took on its own, the duty of verifying the correctness and safety of the developed systems. Verification as a term, is used differently in different domains. In this thesis, when *verification* is mentioned, it is mainly concerned about verifying correctness (system's compliance with some specifications), and safety (system does not go into any bad –non-safe– state) [3]. Moreover, we focus on verifying the system's behavior after deployment.

[3] Colloquially what a statement like "a system is safe to function in field" means: it will not endanger lives or cause harm to humans or their valuable possessions. This statement implies considering all the possible cases before-hand (to be able to meet the expectation implied by this statement); this is naturally limited by our ability to reason about all of them in advance.

Formal verification provides mathematical proofs of correctness and safety (or of their violation), based on models/mathematical-abstraction. The main challenge is to find correct and suitable abstractions/models that express the system in-field behavior faithfully enough to prove/refute meeting the required specifications, while keeping computational effort reasonable. This challenge is magnified when formally verifying systems that interact with the physical world, i.e Cyber-Physical Systems (CPS) [11]. While systems and their models are discrete, *time* and the physical environment are continuous. Factors as pressure, temperature, speed, ...etc, are changing continuously versus time. Addressing challenges associated with CPS resulted in a plethora of research at many frontiers, few examples of such areas are: discrete abstractions of hybrid systems [14], reactive systems model-checking [187], dynamic systems invariants [10], and run-time verification [145], among others.

In this context, tracing and online monitoring of signals, while operating in-field, are considered as means for verifying the run-time behavior of CPS. The challenge from an engineering perspective, is how to obtain indicative, accurate and expressive yet compact information (logged) about the execution.

One classical long-standing challenge in verification in this context can be seen as having two sides: 1) finding finest suitable abstractions (which corresponds to efficient yet detailed informative logs about the execution), needed to express time for generic-purpose verification[4], and 2) finding efficient algorithms to solve the resultant problem of checking the properties given such abstractions (informative logs). Many formal temporal logic variants, and their respective algorithms for verifying the resultant formulas, appeared over the past 70 years in the computer science community, see for example: [57, 68, 135]. Although in this thesis we do not introduce a new logic, we do propose a new paradigm, described in terms of existing logic. Below, we first define some basic terminology, of which our usage here might differ from the existing usage, or where the existing usage is already varying according to the fine specialization.

1.1.1 Terminology: Properties

Here, we call both specifications of behavior, and characterizations of systems: *properties*. In this context, any system model can be described as well in terms of a set of properties, expressed over a set of variables. The desired behavior, or the one the system is required to avoid, can also be described as properties. This way, the verification problem boils down to solving the satisfiability problem, expressing the

[4] By Generic-purpose verification, we mean it is not tied to design-time-predefined properties.

satisfaction of the specified behaviors by the specified system. In this set-up, the input to the satisfiability engine is going to be composed of both properties: the ones representing the system/behavior abstraction, and those representing the specifications it is expected/inspected to fulfill or violate.

Moreover, in the context of tracing, the traces obtained from the field are also considered as execution specifications. We consider them behavioral properties, but ones which are known to hold, in contrast to behaviors that are to be checked-

1.1.2 Terminology: Signals

In the thesis, we shall refer to the signal being traced as *Signal under Tracing* (SuT), or simple *signal* when clear from the context. We will usually use the letter s to refer to it. A set of traced signals, would normally take the letter S. One signal can be associated with a pin in reality, or to a test-point in the physical design where the electrical voltage or current are being monitored. Our signals are specified over a defined finite period of time.

In digital signals, a binary signal would take one of the two binary values. Other standard digital signal values as high impedance, undefined, ...etc are not covered in detail as the binary case, and their considerations will only be covered briefly in the last chapters. The intuition here is: we are tracing signals which take place in reality, not in simulation, or on any other modeling-based method. This means, for pins, we are interested in tracing interactions, where for example, a high impedance output is expected to be already driven by where it is interpreted an input.

Moreover, we are more focused here on signals' changes. The notion of *change-signals* refers to signals which take the value 1, where the traced signal value changes, and is 0 otherwise. For change signals, we directly apply the assumption that they take one of the two binary values.

Change signals are typical derivatives, but we will keep the analysis in this thesis at the level of dealing with them as simple boolean variables/values. We also assume these values are traced at clock-edges. And that for every signal there is a corresponding clock, over which the signal measurements is done at its falling-or-rising edge; this is how our tracing takes place.

1.1.3 Terminology: Trace

A trace is itself also a signal, generated automatically as a trail resulting from continuous measurement of some SuT, or triggered directly by variations in it. To refer to a *trace*, usually the letter τ or t are used. In the literature, a trace is composed of

consecutive trail of states and/or predicate evaluations, taken over a course of execution of a model or an automata. This traditional meaning is different from ours. Here, the trace corresponds to one actual execution of the real system, rather than the model. Ideally, a trace from the real physical execution would correspond to a specific single model execution. But the notion of trace in evidence-oriented tracing is fundamentally different from a model-generated trace. The difference stems from how the trace is actually generated; even if their mathematical representation was similar. Moreover, a model-generated trace is mostly described with a formula or consecutive states of an automaton, while our trace is a sequence of values, where each value has taken place at its respective clock-edge.

1.2 Thesis Overview

The thesis addresses the challenge of finding evident, independent, light weight yet accurate means of tracing complex and fast changing signals in today's sophisticated systems. We start at the level of one signal. For that, we propose *Timeprints* as a candidate, to solve the temporal aspect of the problem[5]. Timeprints are compressed signatures, which give clues concerning the exact timings of change in traced signals. We also show how Timeprints can be used as efficient, independent and accurate traces, that can be saved, and later consulted, to provide evidence.

The thesis also introduces *Timeprints-Based Verification*. Instead of going higher or coarser at the system level to reduce the trace-size, Timeprints themselves compress the traced data into small size. The traces here are not the classical systems models execution-traces, but rather seen from the real execution perspective; as expressive periodic left-overs resulting from electrical signals behavior. The main idea is obtaining regularly a temporal log from signals execution, the so called *Timeprints-trace*. Timeprints are meant to act as signatures or summaries of the execution; from which details can be checked and properties about them can be verified. The importance of having accurate traces of systems execution lies in their ability to act as evidence. The need for accurate and transparent traces, that are capable of revealing or proving specific details of execution, becomes more important than ever with the growing dependence on autonomous systems and artificial intelligence.

The key idea to enable efficient yet accurate tracing, is two fold. *First*, **not** logging what is known or proven to hold. For example, when formal-verification and RV are used, no need to log what is proven to hold by them. The *second* enabler

[5]Temporal aspect refers to the timing of one particular traced signal, rather than spatial which refers to where the traced signal is (i.e. which signal).

idea is bringing the **timing** (at which changes in the signals take place) to the front as the main traced artifact. The trace generation can be described briefly as follows. First, the signal tracing process is split into consecutive (back-to-back) finite trace-cycles. Within a trace-cycle, a signal's value-change instance gets assigned an encoded timestamp. At the end of each trace-cycle, these encoded timestamps are aggregated into a logged timeprint, which summarizes the temporal behavior over the trace-cycle. In this thesis, Timeprints generation will be explained in detail, together with the process of retrieving back the aggregated details of execution from the logged timeprints, which we call *Reconstruction*. Practical case-studies are also provided, to show the efficiency and explain usage and evidence of Timeprints.

1.2.1 Defining the Problem Dimensions

In a recent report, published by the royal academy of engineering [117], where the requirements for safer complex systems were highlighted, the roles of governance and regulatory oversight over operation time were strongly emphasized. All engineering efforts are still focused on designing safe systems in the first place; which is not bad at all; it is rather necessary. However it is not enough, and failures still happen, and accountability has to be backed up with evidence that enables assigning liability. Until today, while systems are in deployment, their execution is not independently traced. Rather, the systems report by themselves about their own operation; if they do at all. This is unacceptable from liability perspective. Given the recent diesel scandal and the latest Boeing's airplanes crashes (as mentioned in the introduction), the many fatal problems in automotive domain [52, 72, 127, 139], and in other domains [110], such dependence on the self reporting cannot be the best solution; a temporal execution evidence logged independently is needed.

For example, the autonomous driving society is still working on a first set of reporting-requirements. They proposed the so called "Molly Problem" [121], to get public awareness to minimum level of reporting requirements. In the article [165], authors discussed the point of civil liability and insurance, highlighting the need for a liability regime. They said "We have preventive instruments and liability to ratify potential harms. It is more difficult to identify rules to assess who is responsible for damage. This is important for the users as it is important for the insurer".

It is required to define the proper amount of data to be logged, not so little that important details of execution are missed, and not too much to be prohibitive to-log, or reveal private information. What is required, as in [92], is an "Evidence that is convincing to the public and sufficiently nonsensitive to be disclosed widely is the

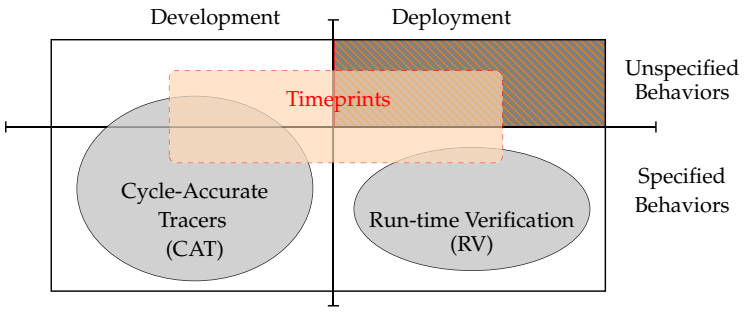


FIGURE 1.1: Cycle-Accurate Tracing Taxonomy

most effective and desirable for ensuring accountability." Trials to provide evidence, on both product and/or process levels, as in [141, 142], were proposed at the very high level of communicating systems reporting about their communication, which makes them far from being accurate or independent, and more of being a new form of diagnostics related to connected systems.

On the cycle-accurate level, variant forms of *Runtime Verification* (RV) [21] and monitoring [20], can be utilized to check a subset of the **specified behaviors**. Specified behaviors are those defined at design-time by the system specifications. Unfortunately, in practice, there are also **unspecified behaviors**, as the specifications are not always complete. Additionally, there are specified behaviors that cannot be formalized and synthesized as on-chip monitors (either due to limited expressiveness of monitoring logic or limited on-chip area). This is illustrated on the right side of Figure 1.1: during deployment, RV can only cover a subset of the specified behaviors and none of the unspecified behaviors.

During the **development** phase, *cycle-accurate tracing* (CAT) is heavily applied in Real-Time (RT) embedded software domain using dedicated debuggers/tracers, such as [17, 161]. This category of tracers focuses on software operating on a processor or a micro-controller, where tracing depends on the existence of instruction set and program counters that carry-on the execution. For arbitrary on-chip signals, other solutions like logic and protocol analyzers and/or scan-chain-based methods for FPGA/ASIC Systems on Chip (SoC) such as [45, 120] exist. These provide different forms of functional and sometimes cycle-accurate tracing. Unspecified behaviors might be captured during *development* by such existing CAT (meaning the circle of CAT might reach to that partition), as shown on the left of Figure 1.1. However, for any generic on-chip signal (systems without embedded software) this is limited to **development time**. During **deployment**, none of these CAT methods can provide continuous cycle-accurate tracing in-field, due to limited trace-buffer's area and/or notorious logging and storage requirements [174].

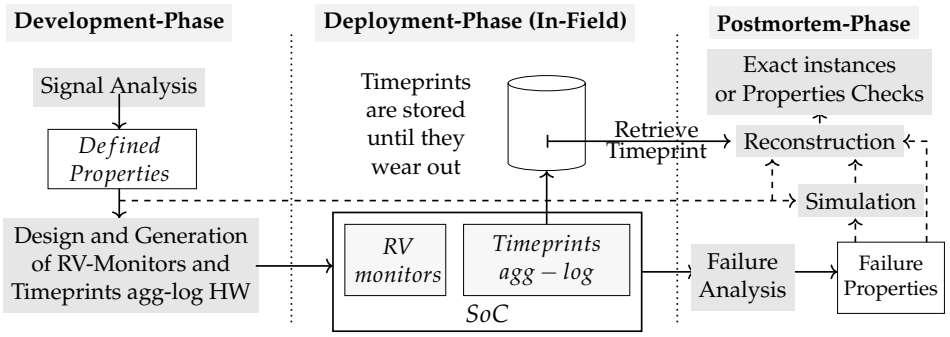


FIGURE 1.2: Timeprints Design, Generation and Reconstruction

As existing approaches can not provide continuous cycle-accurate tracing that cover **unspecified behaviors** during **deployment**, this is where the proposed method comes to serve. *Timeprints* are designed to keep an independent, compressed, and cycle-accurate temporal record/trace of the on-chip traced-signals behavior. This is vital given our increasing dependence on digital systems, of which the executions are conducted on-chip without leaving a trace. Timeprints' very low bit-rate suffices for efficient logging, storage and processing during deployment. Hence, they have the potential to change how we think about in-field cycle-accurate tracing.

1.2.2 Contribution and Proposed Methodology

To enable efficient yet cycle-accurate tracing, the key is bringing *timing* to the front as a main traced artifact. The proposed methodology starts by dividing the continuous signal trace into trace-cycles; where the first trace-cycle would start at reset or at a defined-check point. The decision about the trace-cycle characteristics happens –during development time– at the the signal analysis phase, as at the top left of Figure 1.2. Second, each clock-cycle in a trace-cycle gets assigned an encoded timestamp. A Timeprint is an aggregation of timeencodings that summarizes the temporal behavior. During deployment, a change in the signal values triggers the corresponding timecodes aggregation into the Timeprint (will be shown later in Chapters 3,4 and 5). Finally, at the end of each trace-cycle, a fixed size timeprint is logged. This keeps the amount of logging small and constant over time. To retrieve the accurate timing, (at postmortem) we retrieve exact instances of events from the Timeprint via a *Reconstruction*, as in Figure 1.2. To retrieve the accurate timing, we reconstruct the exact instances and/or check their properties from Timeprints via a SAT query (a call to a SATisfiability solver).

Timeprints are also lossy abstractions. However, the details lost by a Timeprint's abstraction are mostly retrievable. Timeprints do not compromise temporal accuracy

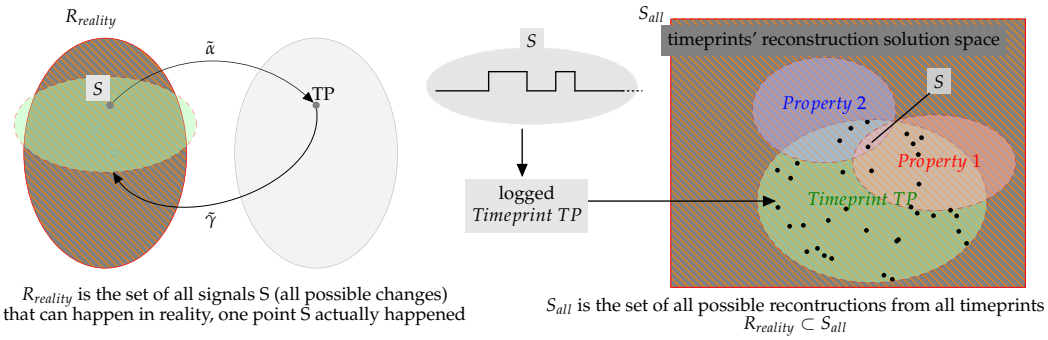


FIGURE 1.3: Timeprints as Abstractions

during the aggregation process, as most traditional abstractions. Rather, Timeprints overlook random data while utilizing the already known (verified) data in the reconstruction; to retrieve temporally accurate details. For example, the details that can be generated by simulation are not considered lost. Both simulation's input and design artifacts can be used in the reconstruction, as in the left of Figure 1.2.

Figure 1.3 shows the idea of retrieving the accurate timing via reconstruction. Each point in the Timeprints' reconstruction space (on the right) corresponds to a set of possible accurate timings that could have lead to the Timeprint at hand. The figure also shows how the ambiguity (many possible accurate timings that could have lead to the same timeprints) is mitigated. The exclusion of non-real solutions by properties' sets as in the right of the Figure 1.3 corresponds to pruning the search space in the Timeprints reconstruction space. The number of solutions can be really huge for large trace-cycle sizes, if properties are not used [44]. Hence, the properties usage, as proposed in [112, 114], is essential to render the whole method acceptable. Ideally, as in Figure 1.3, the reconstruction ends up with a unique signal/timing (the intersection of the 3 sets). This of course is not always guaranteed, as it depends on the coverage of Timeprints as shall be seen late in the thesis.

In the thesis, theoretical formulation of the Timeprints generation and reconstruction, as well as their practical implementation are presented. We contribute:

- an Evidence-Oriented Tracing and Verification methodology, which uses Timeprints as evidence that proves properties about behaviors that have taken place in reality. Unlike RV techniques (which are limited to design-time specifications), Our methodology extends post-mortem checks to the unexpected behaviors domain.

- At the heart of the proposed methodology are the new form of evident, lightweight, accurate and transparent logs: *Timeprints*. Timeprints can be saved and

later retrieved for inspection and analysis, when unexpected failure happens. We explain and formulate various Timeprints generation mechanisms, time-encoding techniques and traces-reconstruction algorithms.

- We made it possible for the first time to log (efficiently *and* accurately) detailed yet abstract independent execution traces, from actual physical systems in-field. While traditional accurate and continuous traces of the CPS in these case-studies, would typically incur several Giga bytes per seconds (GBps), for each traced signal, Timeprints reduce that by orders of magnitude, e.g. in the case studies: Mega bits per second (Mbps), while still supporting the Giga Herz clock-cycle level accuracy of the checks.
- Realistic case-studies which illustrate how Timeprints can act as a formal proof-/evidence of what has taken place inside CPS. Evidence formal proofs from Timeprints for temporal properties are provided as well.

1.2.3 Expected Impact and Applications

Ideally, one would wish for accurate independent traces that are logged all over the execution of every safety critical systems around us, from which any anomaly or un-expected behavior can be analyzed and checked. Such traces-based verification is useful –not only– to extend accurate tracing into deployment phase, but also for capturing and understanding the hard-to-detect bugs in complex systems, either before, during or after deployment.

The transparency and cycle-accuracy of Timeprints makes them a great candidate to act as evidence. Timeprints can provide an expressive trace to analyze evidently the root-cause(s) of failures. This is expected not only to help identifying what happened on the cycle-accurate level; but can also have deeper long term effects. Deploying Timeprints means that tracing the failure root-causes to the accurate level is possible. Hence, such continuously logged evidence is expected to encourage more careful development. Their efficiency makes them useful not only for in-field execution tracing, but also for hard debugging tasks. For example, they can provide clues about what might have went wrong during post-silicon verification.

The methodology proposed was not only theoretically presented, but also the aggregation and logging hardware, as well as the software that reconstruct the accurate timings from timeprints, were all developed [114]. Several timecodes generation algorithms [116] were presented, and many of them were used for different tracing applications: software execution checking [113], bus temperature effects detection and CAN-bus messages accurate timing checking [114]. The applications are not

limited to these, but these are the ones carried on within this research as illustration of what could be done.

1.2.4 Publications

Up to the moment, here is the list of published papers, which resulted during working on this thesis. In the early papers, in 2017 and 2018 for example; the name *Timeprints* was not yet coined. The concept though was starting to develop since the first FORMATS 2017's paper. There, we used the name *Footprints* instead; and the functional aspect was considered beside the temporal aspect as well. Then, by focusing more on the temporal aspects, *Timeprints* as a term was born; and interesting results were easier to highlight.

- R. Massoud, J. Stoppe, D. Grosse and R. Drechsler, "Semi-Formal Cycle-Accurate Temporal Execution Traces Reconstruction", Proceedings of the 15th International Conference on Formal Modelling and Analysis of Timed Systems, Pages 335-351, (FORMATS) 2017, Berlin, Germany.

- R. Massoud, J. Stoppe, K. Madihunta and R. Drechsler, "Time-stamps for Hardware Simulation Models Time Back Annotation", DUHDe Workshop in DATE, 2018, Dresden, Germany.

- R. Schmidt, R. Massoud, J. Raik, A. Garcia-Ortiz, R. Drechsler, "Reliability Improvements for Multiprocessor Systems by Health-Aware Task Scheduling", Proceedings of the 24th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), Pages 247-250, 2018, Costa Brava, Spain.

- P. Chini, R. Massoud, R. Meyer and P. Saivasan, "Fast Witness Counting", arXiv preprint, 2018

- R. Massoud, H. M. Le, P. Chini, P. Saivasan, R. Meyer and R. Drechsler, "Temporal Tracing of On-Chip Signals using Timeprints," 56th ACM/IEEE Design Automation Conference (DAC), pp. 1-6, 2019, Las Vegas, NV, USA.

- R. Massoud, H. M. Le, R. Drechsler, "Property-driven Timestamps Encoding for Timeprints-based Tracing and Monitoring", Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems, Pages 41-58, (FORMATS) 2019, Amsterdam, Netherlands.

The thesis contains many materials, examples and explanations which are not yet included in any other published work. This includes the Evidence-related content,

mostly in Chapter 7, and experiments in Chapter 9. These are corner stones for the evidence-oriented verification methodology presented in the thesis. Next, we give a brief overview of the thesis structure.

1.3 Thesis Structure

The thesis is divided into three major parts. The first part is an introduction and a summary of the work done. This part includes three chapters: this chapter, the background, where the main tracing problems are identified and the related literature is surveyed, and lastly, an overview of the work contributed to solve these problems. This part is enough read to get an overview of the work done without getting into details.

Then the second part of the thesis consists of the next four chapters. These represent different foundational aspects behind the timeprints, and evidence-oriented tracing. The third and last part contains the last three chapters; which contain the more practical aspects: connecting pieces together, application, conclusion and outlook.

Here follows a brief view of the chapters.

1.3.1 Part 1: Overview

Chapter 1 This introduction.

Chapter 2 This chapter presents the background. It gives a look at existing temporal verification challenges, the theoretical reasons behind the limitations, and the existing methodologies and techniques that have been proposed as solutions. We then emphasize the still standing gap. This chapter classifies the major obstacles under three main problems: accuracy, describing temporal properties, and physical tracing problems. It discusses briefly each, with a quick overview of the related literature.

Chapter 3 This is an overview of the main highlights of the proposed *Timeprints-Based Verification*. The main ideas of abstracting the temporal execution and mapping them into Timeprints, summarizing the temporal behavior are first presented. Both the process of generating and logging the timeprints from the execution, and their respective reconstruction of the temporal characteristics are also presented. A summary of the theoretical aspects and the practical applications are also given.

1.3.2 Part 2: Timeprints Foundations

This part covers in some level of detail, the four basic concepts –or *foundations*– of evidence-oriented tracing and Timeprints. These are mainly: 1) abstractions, 2) time-encoding, 3) reconstruction and 4) evidence. The part starts by explaining Timeprints as abstractions, and the first chapter in this part puts Timeprints in the context of existing abstraction-related literature. Then, encoding time, which is a fundamental part of generating the Timeprints, and the main component which allows the compression of detailed timing information is explained. Reconstructing the original accurate timing data from the Timeprints is presented in Chapter 6. Then, using Timeprints as evidence which can prove or refute temporal properties is discussed in the last chapter in this part.

Chapter 4: Timeprints as Abstractions Timeprints abstraction of the traced signal is described here. In this chapter, definitions and formulations of Timeprints as abstractions are presented. The chapter also contains a colloquial translation from some traditional temporal logic to the Timeprints domain. Which properties are preserved, i.e. can be described and/or checked by Timeprints is also discussed; although the full extent of Timeprints' capability comes later in Chapters 6 and 7.

Chapter 5: Time Encoding Here is where the Time-codes, by which signal's changes are triggered into Timeprints, are explained. The basic degrees of Time-codes' linear independence is first discussed. Then, the problem of Time-codes generation is formally described. Different Time-Encoding generating algorithms are presented in the chapter. We also present property oriented time-codes generation and describe the concept of encoding-generating functions.

Chapter 6: Traces Reconstruction In this chapter, the reconstruction of change-traces is explained. Here, we focus on the trace-cycle reconstruction, where each Timeprint, summarizes the temporal behavior over a trace-cycle, and all candidate change-signals are obtained. Different reconstruction techniques are discussed. Also adding properties to the reconstruction-problem is described as a variant of the trace-cycle reconstruction problem. Checking the temporal properties over multiple trace-cycles, (inter trace-cycle) is also formulated; although it is discussed later when addressing the Timeprints' evidence. The size of the reconstruction set (number of solutions to the SAT problem) is discussed at the end of the chapter.

Chapter 7: Evidence of Timeprints This is last chapter in the theoretical foundation part. It describes the formal framework of using Timeprints as evidence. It starts by an overview of theory of evidence, then puts it in the context of tracing signals with abstractions such as Timeprints. Then the necessary and sufficient conditions, needed for Timeprints to act as evidence are stated. After that, a typical example is given: using Timeprints as evidence of delays.

1.3.3 Part 3: Application and Conclusion

In this part, the previously detailed foundations are put together into action. First, pieces from theory are put together, where also extensions and limitations are discussed. Then, experiments and case studies come in the Applications chapter. Finally, the thesis is concluded and future directions are highlighted.

Chapter 8: Putting Pieces Together In this chapter, we connect together in one context the four foundations chapters. We show the big picture and how Timeprints and evidence oriented verification are used in practice. Tracing multiple signals with Timeprints is also discussed, as well as combining different Timeprints, to check a system's property. Projecting global properties of trace-cycles and some practical aspects for using Timeprints are discussed. The chapter ends with a discussion of the generalization and the limitations of Timeprints.

Chapter 9: Application This chapter describes many experiments and case-studies which were conducted and carried on to illustrate the usage and efficiency of Timeprints. First, Timeprints are used to trace "address signals" on an AHB bus. This experiment shows the ability to detect the memory refresh instances; as they cause one cycle delay on the bus. Then, using Timeprints in evidence based tracing of the delays on the CAN bus is described. Evidence, coverage, statistical and formal aspects are also discussed. Another case-study of ultra-sonic sensor signals and brake signals of a donkey car is presented. Proving the ability to cover delays in the ultra-sonic sensor signal is also shown.

Chapter 10: Conclusion and Outlook Here we provide the conclusion remarks. First, the main contribution is summarized and assessed.The constraints and considerations to use Timeprints are also discussed from futuristic perspective. For example, current and expected implications on safety, security, compilers, and embedded systems development and production life cycles are discussed. In this chapter also,

the currently pursued directions are described; for example: using theorem proving for asserting the ability of detecting some properties under certain assumptions, bus protocols and periodic signals signatures. , and Timeprints hardware realization considerations.

1.4 Reader-Guide

For a sufficiently informative overview of Timeprints and evidence-oriented tracing, the reader can jump to Chapter 3, which (besides the general overview) gives pointers to where details of foundational and practical related work are mentioned in the thesis. Chapter 2 provides a background of the problem which the Timeprints are trying to solve. It also discusses related work. So a reader who still needs more about the back-ground and motivation are welcome to read this chapter. The chapters from 4 to 7, dive deeply in the four main theoretical aspects of the work. Readers can go directly to the chapters where they want to dig deeper. Chapter 8 connects the dots and discusses associated limitations and extensions, and can be read by the non deep-diving reader after chapter 3. Then, chapter 9 provides case-studies where Timeprints and evidence-oriented tracing were used. Finally, chapter 10 concludes and gives hints to future direction.

Chapter 2

Background: Tracing Problems

"Traces" as a term is used differently in different domains. Briefly, in the modeling domain, a trace expresses state-changes along one execution path of some system model or automaton. In the context of simulation, a trace results from logging information about one exact simulated-run of a model (representing a system or a behavior). The simulator can be configured to save and/or log the encountered values of some specified signals in a *trace*. Formal verification of systems aims at mathematically verifying that all possible executions (traces) of a system's model comply with some formal specifications, either by enumerating and checking these possibilities explicitly, or by inference and reasoning about/from axioms and premises describing the system implementation and its specifications. On the hardware (physical domain), tracing signals, such as inputs/outputs and instructions executions, have been extensively used in debugging manufactured chips and embedded software. In this thesis and in this chapter we consider traces as in the later domain, namely: traces obtained from actual physical execution; while extending their usage –beyond debugging– to the in-field post-mortem checks. Moreover, traces from the modeling domain are sometimes used for comparison with traces obtained from in-field execution.

In Chapter 1, we showed how systems' growing complexity, and shortage of verification coverage, have lead to critical fatalities. In-field accurate and transparent tracing, even if it could not help in avoiding the fatality, can at least give detailed evidence about what happened. In its turn, this evidence will help in learning from failures, to avoid the problem in future, and can be used in assigning liability for the fatalities; which can hence drive more careful and responsible development.

In this Chapter we give an overview of the different facets of the tracing-problem. By "tracing-problems" we mean challenges and barriers facing *accurately tracing signals in-field*. After a high level description of the problems, we explain each aspect with a glimpse on the existing literature and technologies related to it.

2.1 Why tracing signals in-field?

Many safety critical systems, like airplanes, depend heavily on several Cyber-Physical Systems (CPS) in many stages of their design, manufacturing and operation, let alone that they are themselves CPS. Such systems are developed by many cooperating companies, corporations and an extended chain of suppliers. Many standards evolved (concerning both the product and process) to maintain the correctness and reliability of such systems. Lately, the trust in technical capabilities was severely hit by different accidents [52, 65, 72, 82, 110, 126, 127, 139], and the situation seems like the existing standards, regulations and tools are not anymore sufficient [1, 117].

Despite the many existing practices, tools and standards, not only errors do still happen, but they lead to failures and fatalities. If we consider the automotive-industry, safety standards that considers software and embedded systems (that contains both software operating on hardware) started appearing only in the last 10 years [146][1]. The automotive industry has been mechanical for many decades, and started to rely on software and CPS only in the last 30 years for critical tasks (like power train) [88]. Since then, there have been already many recalls, accidents and even deaths [72, 88] reported that happened potentially because of problems in the development, implementation and/or design of CPS.

The state today is that all such systems report about their own behavior by themselves. The reports are very coarse and designers give such reporting tasks very low priority (which is justified as they are not doing a critical function, however, this makes them usually not temporally accurate). Reporting tasks are also often intrusive, i.e. taking from system's execution time. This reporting is also usually done on a very high level and is often very coarse in terms of the timing and the detail of reporting. Moreover, as a result of being intrusive and of low priority, the reporting time becomes not accurate enough for checking the real-time interaction with the environment from one side, and between the internal system's modules on the other side. Because of the huge size and complexity of the systems and signals involved, tracing the operation of such complex systems *accurately and independently* is inherently a very challenging task.

The problem with today's existing solutions which try to address these challenges, is many-fold. One side is addressing the trade-off between the required high accuracy and the hardships coming with obtaining huge amounts of data corresponding to it. Many solutions tried to solve this dilemma, some from the accuracy perspective like those explained later, in section 2.3, and others from handing the

[1] The first edition (ISO 26262:2011), was published on 11 November 2011.

huge trace-size perspective, as shall be shown in section 2.5. But neither the ones which reduced the accuracy were able to provide efficient tracing without losing important timing details, nor the ones which tried to reduce the size of traces were able to reach meaningful reductions, that can enable deployment phase tracing. Section 2.4 explores literature which address the temporal tracing-accuracy from the description perspective.

In the verification domain, formal methods are providing promising results for verifying correctness of designs [50, 181] and coverage of tests [144]. Although design-verification verify models and abstractions of the systems, not the actual manufactured artifact itself, we can still learn from the way they describe traces and temporal properties. For example, test-coverage (test-completeness) related methods keep missing bugs that occur only at the field after fabrication, as in [58] or at deployment [29]. This was a major driver behind extending formal-methods to run-time monitoring domain [83], using both software and hardware methods [145]. These methods successfully addressed checking some predefined problems in run-time. However, they are still strongly tied to those a-priori defined specifications to be monitored. They can not provide generic traces from which one can check something that is not covered by the pre-defined specs. The later is needed when unexpected failures happen.

The upcoming section introduces the main gap and its different facets which reveal themselves in various applications. Then, each section would start addressing separately an aspect, first from the theoretical point of view and second by covering the main contributions in the existing body of knowledge which tried to address it.

2.2 Tracing for Verification from Designs to Implementations

The hardness of verifying the correct functionality of Cyber-Physical Systems (CPS) does not only stem from the increasing design sizes, speeds nor complexity. But also from the endless possible factors that can affect the CPS performance during its interaction with the real physical world; which cannot be covered with mathematical abstractions. As in [38], the authors acknowledge these limitations and their relevance to autonomous systems safety.

Designs are abstractions. The more abstract a design, the more computational approachable its verification is. For this reason, rigorous verification methodologies start from higher levels of abstractions. Then, they automate –at a refined abstraction level– the generation of realizations while verifying the refinement generation process at each step [8]. Unfortunately, even with such rigorous methods, the final

manufactured physical artifact cannot be mathematically verified. Rather, it goes through exhaustive testing to verify the absence of defined bugs and to validate fulfillment of requirements. Mathematical models of the environment and other real-world interaction are not perfect as well.

If something went wrong after production (post-silicon) or even after the system is deployed; i.e. while operating in field, there is no standard method to trace and detect what went wrong. The fast speeds of the systems on-chip executions make their accurate tracing a hard quest. Post-silicon bugs caused more than 50% of designs in a 2020-study to require two or more spins, timing related flaws (Timing-paths fast/slow and/or clocking) contributed to more than 75% of the bug escapes in FPGA's and 40% of the bugs in ASIC's[163].

Run-time Verification (RV) [20] and Run-time Enforcement [133] have emerged with the aim of verifying and correcting the implementations executions, while they are running in field. This shift the focus from modeling the endless possibilities of the physical world, to actually realize the system's specifications into physically implemented checkers, that reside with the system and monitor its execution compliance against these checks. A major challenge for RV and execution enforcement is being restricted by the specifiable and realizable checkers, which in turn have limited coverage and accuracy because of the available on-chip resources. They cannot cover all failure root-causes because of the inherent incompleteness of the specifications. As there is no complete specifications, unspecified factors and behaviors can still result in faults during in-field operation. If an unexpected failure happens, where none of the RV-checks was fired, the trace available can rule-out the checks as sources of failures, but cannot help with verifying the further analysis results.

Problems related to the representation ot temporal properties are covered briefly in section 2.4. The practical aspects of the problem of having accurate yet evident traces, logged continuously during the operation, are addressed in section 2.5. We first start by the necessary background on accuracy, in 2.3.

2.3 The Accuracy Problem

The accuracy problem is natural. Actually, it is not *accurate* to call it a problem. Rather, accuracy is a feature highly desired. But here exactly lies the problem; striving for absolute tracing accuracy is not only exhaustive, but sometimes a mission impossible. For example, when tracing analogue continuous signals, sampling the signal means only *samples* are going to be measured. Besides the lost values in between the samples, quantization causes loss as well. Even the analog sample itself

is actually part of the signal being measured, and hence affects the measured value itself. Although tracing digital signals is not so intrusive (in terms of changing the measured value) as their analogue counterparts, still absolute tracing temporal accuracy is absent in terms of the exact timing (value given within a clock-period). Clock skews, jitters and path delays, are all factors affecting the temporal accuracy.

A trace is a representation of a reality, and it is highly required that it is accurate. A trace obtained from measuring or observing a signal or phenomena, would never be equivalent to what exactly was traced. But what is needed from a trace, is that it acts as a documentation, which in turn helps later to recall and understand what happened as needed. When it comes to obtaining traces of signals, the trace itself shall also be another signal. A major difference, between a signal and its trace-signal, is their roles. While the traced signal carries on a functionality, its trace usually does not; and is only stored for inspection. The trace is a trail, left behind or generated by the signal leaving or generating it.

For each signal's recorded value, or for each measurement, the question is always: how accurate the trace should be? There are two aspects of accuracy. First, how many bits (or digits) are needed to represent it with enough accuracy? This corresponds to what we call spatial accuracy. Second, how many points, within a fixed period of time, the signal's value shall be recorded? We call this temporal accuracy. The more details required to be recorded (traced) about a signal, the more resources (for logging, storage and processing) are needed. These can even be more than resources which produced the signals. Whether this is acceptable depends on the application. For example, signal receivers are known to be more complex than transmitters. But when it comes to tracing systems execution, just to check the traces back when something goes wrong, while running in-field, huge logs are not anymore acceptable. They have to be very light and coarse. Otherwise, any higher accuracy has to be accompanied with a strong justification.

In the traditional definition of accuracy on chemistry or physics, measurement accuracy expresses how close is that measurement from the real value. There also appears the "probe effect", where measuring affects the signal itself. In digital signals, where a signal takes one of two value, zero or one, the value is usually kept by the circuit, and can be maintained up to a certain level, even if read by multiple consumers/readers. The maximum number of signal's readers, so that the value itself is not affected, is determined by the gate's fan-out. Hence, theoretically one can measure and trace with full accuracy, a digital signal.

When we assume tracing the exact values (level: 0 or 1) of a digital signal can

be done correctly, i.e. accurately and fully matching the reality, what remains problematic is: how frequently are measurements going to be made? If measurements are obtained every clock cycle, with the same edge on which the signal changes, one can obtain a theoretical perfect copy of the traced signal. Such a copy, albeit being accurate, is a huge burden to record, save or log and process. On the contrary, the signal itself is consumed in computation and is not stored. But if we wan to have a trace to check, we have to decide about which values/states of the signals we want to keep, and/or when the measurement/probe is going to be carried on.

The expression "temporal tracing" is used for referring to obtaining timed traces of the traced signal behavior. Timing details can be defined in terms of 1) resolution, by which the traced signal time is described, plus 2) the instances at which measurements/tracing are/is taking place (e.g. event triggered, withing constant period/-clock, ...etc). For example, if the sampling rate is fixed, a free running clock can be used, with a wide enough counter, and measured values with their corresponding counter value can be recorded when some events happen. Hence, temporal accuracy then is greatly determined by the clock; used for either sampling or counting the current number of lapsed clock-cycles. In the following subsections, we explore in more detail the accuracy of such timed measurements, their clocks, and the effect of real/integer numbers interplay.

2.3.1 Tracing Accuracy Measure

Accuracy here refers to how close are the recorded/traced values in a logged signal trace to the actual signal-under-trace values. And temporal accuracy means not having the timing aligned with some universal external time, but rather having the measured values timing as close as possible to the traced signal values' timing.

On the temporal tracing-accuracy, we differentiate between continuous and discrete signals; and also between actual in-field physical traces, and those generated from models. While time is inherently continuous, most of complex systems humans develop today are digital. Models can have both continuous and discrete variables, and in-field measurements/traces are limited to the capabilities available to capture them. For example, measured signal values are obtained in samples. Continuous signals are usually sampled at a constant rate, with some sampling-clock edge.

To determine the sampling-rate, Nyquist criteria defined the minimum frequency, which is needed to sample a continuous signal, without loosing any of its frequency components. This criteria stems from Fourier transform, and demands the frequency domain representation of the traced signal, and is heavily used in electrical signals

transmission. In communications theory, receiving a signal can be seen as a kind of tracing. In this set-up, the timing of every sample is known implicitly, from the sampling clock and the sample order. Spatial trace-accuracy here means how the logged digital representation of the measured samples values, compare to the real signal values.

For digital circuit, where signals change with respect to a clock, taking that clock as a sampling rate looks optimal. This works perfectly well only in ideal world, where neither path delays, nor synchronization problem exist. Path delays might even occur on the same chip, and have to be taken into account when synthesizing the circuit, and deciding about the maximum clock by which it can operate. If we assume all what is needed are the digital signal values according to the system's local clock, the problem which remains is the huge size of the traces, resulting from such clock-based tracing.

In conclusion, for digital signals, logged as value-timestamp pair, the temporal accuracy becomes the main concern. After this little introduction, we look at how the existing literature approached the temporal measurement's accuracy problem.

2.3.2 Literature Exploring Temporal Accuracy

The accuracy problem as defined above has been explored in many fields within the literature. Here, an overview of some important aspects related to accuracy of tracing is briefly presented. The definition of time-series in [173], where traces, signals and time series are treated equivalently, represents a good bases for judging accuracy of timed series and signals. In [173], the authors brought the concept of Hausdorff's distance [76] (which lately became popularly used in image and pattern recognition) as a measure of error/accuracy of *learned* timed signals values.

Accuracy and Real Numbers For any traced signal, both time and value are actually real numbers, because they represent natural quantities (e.g. time-lapse in seconds [2] and voltage of the signal in volts). Sampling time, when constant, shall be triggered by crystal oscillators, which might themselves have drifts and skews. The measurement of traced signal values, are usually taken with an edge of clock. Measured values are then quantized, because numbers cannot represent actual real numbers perfectly. This means we have multi-dimensional inaccuracies.

Even for digital signals, the time between samples/measurements recorded in the trace is a real number. However, this number is going to be calculated by an

[2] Time lapse can be considered scalar measured from a reference point. In general time flow can be defined in relation to other events.

integer counter, whose value represents the number of clock-cycles. Even for signals triggered by clock-edges, they are still spread over an area, and encounter propagation delays. To count for these effects, a mapping is needed, between the actual real-number representing real time, and the integer number counting the clock-edges of a digital clock.

Deadlines and other timed specifications are usually stated in terms of absolute time; not in terms of clock-cycles passed. However, systems are fed with digital heart-beats coming as clocks from crystal oscillators. This digital-tools versus analog-requirements raises many concerns. Some of these were addressed in the area of hybrid systems [14, 48]. Almost all theoretical limits of representation of real numbers land themselves in this context. Other issues related to clocks skews and jitters, were also addressed in the context of robustness as in [2].

Real-Time (RT) interaction with external physical environment makes the verification of safety critical Cyber Physical Systems (CPS) exceptionally hard, because their temporal behavior is specified, with respect to the environment's time and state; rather than the system's internal state. To this end, local internal clocks (based on physical crystal oscillators for example) can serve as a reference, to which the external world time can be measured as perceived by the internal clock. In this work, the system's basic clock is used to obtain efficient temporal abstraction of the traced signals.

For a trace of one digital signal w.r.t. its finest clock, time intervals' real values are not reflected, (as they can be expressed as integer number of clock-cycles). However, when it come to tracing multiple signals, real values shall appear as we start considering the skews and jitters, even when the same clock is used for all the traced signals. Moreover, when different clocks are used, the regions between them shall start appearing as real-numbers as well.

Clocks Digital systems almost always operate with respect to a clock. Although many of the internal computations and operations are done asynchronous to the clock, and these computations would definitely take time, there always comes a point at which events are synced with the operating clock. Time, however, advances in continuous manner, rather than the idealized integer steps of a clock. In Time-Automata based tools like Uppaal [28] clocks play a central role. Theoretical mapping of the model-clocks to continuous time is addressed by many methods: Among these, the most famous is the clock-regions described first in [60], which prevailed and dominated the approaches used in temporal formal model-checking (known as difference bound matrix DBM). This time abstraction itself may introduce errors

on the process of implementing it. For example, a bug which caused the liveness checker to produce wrong results in some cases, related to the difference bound matrix implementation of the clocks, was found in Uppaal [98], the famous Timed Automata Model Checker.

The approach of this thesis strives to utilize the most accurate clock available for tracing each signal. Hence, it has better chance facing the temporal accuracy problems, than hard-coding a timestamp-value pair in a log (which in its best accuracy-case can use a counter which uses the fastest system clock, but will then suffer from the processing delay and larger log-size to denote the counter-time). On the other hand, choosing to log traces with less than the fastest system's clock causes temporal measurement error to grow. In this regard, Timeprints perform better because their abstractions keep clock-cycle level details, as explained later in Chapter 3.

2.4 Temporal Properties Description Problems

Temporal logic was historically developed to describe models or programs' properties; rather than an actual temporal behavior of signals on hardware [91, 135]. In this section, go through a brief description of that history, and highlight the most relevant land-marks to the accuracy of temporal tracing.

2.4.1 Theoretical Background

Expressing temporal behavior came relatively late, compared to other systems functional aspects descriptions. For example, timing was considered as "non-functional" aspect in many software development life cycles [3]. Systems interacting with physical world need to take timing more seriously. Hence, many methodologies and models emerged. Good surveys of the different temporal logics, which addressed timing properties, and their usage in monitoring and checking properties can be found in [57, 107]. Most of these focus on how to take order and time into account, within the existing design models. Models can have both discrete and continuous parameters, so continuous time was affordable in many models, for example [168]. However, most of designed systems are digital/discrete. To accommodate both in the systems analysis; Hybrid-systems verification emerged [14] as a result.

Verified engineering-design-models are usually used to generate the designed system's physical realization. These manufactured artifacts can be verified via testing and/or other post-silicon verification techniques. Finding and fixing problems

[3] Is the case in most non-embedded software development life-cycles till very recent.

encountered at these stages also use formal methods, as in [24, 85, 120, 131, 174, 179, 183, 184]. Mainly, these methods are used to amend the limited visibility and trace-ability on the manufactured hardware.

Metric Temporal Logic (MTL) was the first logic suitable for hardware monitoring; [20, 80, 90] give overviews of it. Its decidability was addressed in [130]. Timed Automata was introduced in the early 1990's to include the time to systems models. Model-checking timed automata is surveyed in [34, 35, 185]. Their interaction with clocks raised the notion of clock-zones, which was explain in [14, 99, 168].

Temporal Logic development has been mainly concerned of how to efficiently represent temporal events. After first starting from philosophy,[129], getting it down to systems checking [36], the quest of letting that representation accommodate and be more sensitive to time passage, as in [87, 176], hasn't stopped. From run-time verification [145], to systems input/output timing [40], Time Automata, Hybrid Systems [5, 48], and Signal Temporal Logic [61], all these efforts stemmed from extending the classical philosophical views to modern verification. That view always kept the focus on events first, their sequence second, and then their exact timing third. Apparently, this worked very well especially in terms of reasoning efficiency. It enabled focusing about the systems attributes and their verification versus specifications. But since the quest of extending these concepts to monitoring and run-time verification started, the syntax and semantics grew more counter-intuitive and complicated.

Temporal Logic Temporal-logic can be used to "describe" and/or to "check" properties. Temporal logic has emerged to check the compliance of designs, (or system abstractions) with respect to some time (or order)-related specifications, in an efficient way. Origin of Propositional Linear Temporal Logic, known as LTL is in most literature traced back to the 70's, and specifically to Amir Pnuelli's paper [135]. The major contribution there was the two notions of "eventuality" and "invariance" (as the paper calls it or "universality" as it is mostly called now) used in describing what should eventually happen ot what should always hold as invariant. Pnuelli himself built his descriptions on Saul Kripke's temporal madalities describing possibilities of what might or shall happen in the future, proposed at 1963 in [91]. Kripke's work on its turn is built on Brouwersche's axiom, which defined the universal modality with regards to members of a set [180].

For example, if a system is to perform a certain function, where the outputs has to appear in certain order, or the order of specified values behaviors at the inputs, requires that the system produces different outputs. For this type of systems, it suffices

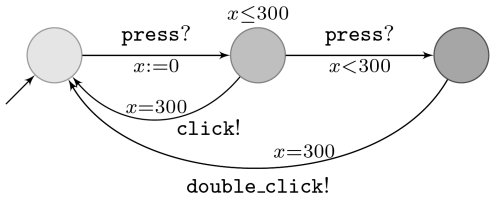


FIGURE 2.1: Double-Click Automaton from [34]

to use the temporal logic LTL, CTL or CTL* to describe these behaviors. To consider when the values of time lapses between events matter, Timed Automata [12] was introduced in the early 90's. The focus on formulating the systems interaction with their environment was further emphasized in [136], in terms of reactive and concurrent systems logic. Later, other aditions appeared, such as Metric Temporal Logic or MTL [80, 109], to represent the measured (Metric) time into the temporal logic expressions.

Timed Automata is great in modeling timed systems, but when it comes to hardware monitoring, it cannot help –as it is– in checking temporal properties satisfaction of execution traces. To explain this, consider the simple double-click automaton in Figure 2.1, taken from [34]. The automaton shown expresses the condition to consider two consecutive clicks as a double-click, describing how a mouse's double click behavior should be.

The traces (which are going to be examined to see whether they comply with the behavior described by this automaton) are not going to directly map to states and transitions. Remember that traces are lists of consecutive traced signal values. These values are obtained from some real physical execution, and are saved on trace-buffers or logged off-chips where execution is taking place. In our double click example, our trace might look as follows: two traces of timed values, one for all the mouse clicks and when they happened, and another trace stating when the system decided that there were double-clicks. We want here to check that always –in that trace– whenever two clicks happen within 300 ms, they are considered a double click. Furthermore, the automaton also indicates that when a double click takes place, the 300 ms timer is reset and counting the clicks starts over as well. It also implies that no double click action should happen in any other case.

Checking that this is satisfied means that we have to check all over the time period, for which the trace is given, that the properties described by the automaton are satisfied. Finding out whether certain trace satisfies a behavior modeled by some timed automaton, like this one, cannot be done without several intermediate steps. In our example, for each pair of entries in the clicks list, all the properties mentioned

has to be checked explicitly and separately. Notice that each of these checks spans various states and the transition conditions. More about the how hard is that check can be found in [128], where the authors provide translations from Metric Interval Timed Logic (MITL) and Metric Temporal Logic (MTL) to deterministic automata. MTL,MITL and other logics were proposed for the purpose of monitoring. Propositional linear temporal logics with metric time measures is the closest to our needs when we are focused, as here in this thesis, on one trace of actual execution. Metric temporal logic (MTL) can be used to describe the logic entailed by Timed Automata. Although the point-wise MTL is less expressive, see [62], it is still the one which actually represent logged traces of samples taken with a clock-edge.

Traces, in that context, correspond to the whole system states' possible changes. If some system's execution includes conditional branching, and hence different execution paths which depends on some variables, the temporal logic becomes a branching-logic, otherwise it is considered *linear*. This view is focused on checking the design abstraction, rather than the trace resulting from one execution.

Quantified Temporal Logic When talking about a single trace of execution, the quantification is mainly existential. Universal quantification can still be used; but not to express safety properties in the classical sense. Rather, in the context of checking one trace, if something is required to hold over the period, we can use universal quantification over this period, to describe the property did hold over it, during this one trace.

Simulation and Bi-simulation Traces, resembling one system model execution are considered simulation traces. Simulation and Bi-simulation [57] were mainly used to analyze systems behaviors and their equivalence. They can still be used to help analyzing a logged trace, when the data logged are not enough to reconstruct what happened exactly. A matching and check-points can help aligning logged traces with simulation traces. Including time in simulation and bi-simulation was addressed in [41].

Finiteness vs Infiniteness of Traces Infiniteness is considered as a generalization to be able to check properties that are expected to hold *always*. Finiteness however is more natural when we are speaking here about one particular system execution trace, which typically starts at reset (or a periodic checkpoint for example) and ends where the systems fails or needs to be inspected in general. Finite traces are not

problematic, and can usually be theoretically extended to fit for consideration by an infinite check. Propositional LTL on finite execution traces was proposed in [77].

2.4.2 Traces and Verification

Verification can be done in many ways, model-based (formal model-checking or execution via simulation), or test-based, where the system itself is excited by various inputs and its behavior is compared to a golden reference [4]. Verification can be formal, where mathematical proofs are provided, either via model-checking or theorem proving. It can also be statistical (as in simulation or testing and then calculate percentage of coverage). A simulation or an execution is going to result in a behavior, which can be expressed in terms of signal values over time. These timed signals draw a trace of the execution. Model-checking can be done by checking every possible trace of execution. Simulation can also be configured to produce a detailed trace of execution to be checked; e.g. for debugging. Testing might include just checking some final output or a series of signals measurements over time, (execution-trace).

Using models, simulation-runs can be represented more efficiently; enabling model-checking without inspecting separately each single trace e.g. using symbolic execution [31]. Counter-example guided abstraction refinement [48], using behavioral equivalence [153] and the use of fix-points as in [105], are also other examples. On the contrary to models, when monitoring the real system behavior, traces are measurements of picked signals from a single execution. Hence, these traces cannot benefit from such efficient methodologies; at least not directly.

Whether the trace is obtained from simulation, or from real execution (run-time), huge data volumes need to be stored. Alternatively, sometimes only the trace-check results are saved and reported as a trace. It is still very hard to inspect raw data for debugging or to do further checks/investigation when the data size is huge. Some methods trace only very high level software and I/O, like in [106], to avoid complexity and use formal methods. Traces are at the center of our verification methodology. We keep as much accurate, transparent and efficient traces as possible, to base our verification upon.

we consider a discrete time domain. Moreover, when real time information are concerned over a trace, usually this is conveyed in one of two ways, as described in [140]. Either *point-wise*, where each event takes a timestamp or *continuously*, as in [22] where intervals associated with certain status of satisfaction or violation of a

[4] A golden reference behavior –or trace– is the reference-design's expected behavior, and which results in the golden reference traces; expressing the designed behavior over time.

property or a proposition. In this thesis, we actually use a mix of both. At the beginning, we focus on one finite trace. Declaring constraints using LTL started in [132] to described business processes and then generalized in [55] and expressed using propositional dynamic logic [67], temporal logic over finite traces was addressed. In [43], the authors stated clearly that first order logic over finite traces cannot be axiomatized recursively. However, they also emphasized that when the underlying sequence of time-points is finite, a formula is defined to be *k-valid*; they showed this validity to be sound and complete. Hence, after focusing on one fixed-size finite trace, we then extend that and concatenate several consecutive traces, to form a window of interest over which the *k-validity* holds.

2.5 Practical Temporal Tracing and Verification Problems

Practical tracing problems can be summarized under the following points: 1) selecting which signals to trace and with which accuracy and/or frequency, 2) size of the generated trace and its generation rate variability over time, and 3) storing, logging and retrieval formats of the resulting trace. Which signals to trace, and with which accuracy or frequency the need to be traced, depends heavily on the application. Whatever were the signals chosen to trace, with today's on-chip operating frequencies, the size of the generated trace is a huge burden. So different methods use their own compression techniques, to reduce the size of generated traces, whether these are stored on-chip or logged. Logging traces to external device or host is used widely, due to limitations of on-chip storage. But even then, the problem is shifted to the required logging facilities: log-bandwidth, composed mainly of pins and available operating frequencies.

To not miss tracing important relevant events, and at the same time avoid logging huge traces, Lamport has proposed in the 70's to mark the relevant events with Timestamps [97] , which are only incremented when temporal distinction is needed between ordered events. This concept of Lamport's Timestamps was highly useful in distributed system, as it freed the tracing from a lot of burdens. It was not only a way to avoid massive unbearable logs, but also the clocks synchronization problems were also avoided. This efficient abstraction lead to the emergence of a whole new branch of "Concurrency" research, which focus on verifying (within the existence of simplified high-level representation of time) that all possible interleaving of events happening at the same Timestamp fulfill given high level specs [96]. Short history of *"Concurrency"* ca be found in [95].

Concurrency smartly dealt with multi-threaded programs, as well as parallel and distributed executions. It focused in the first place on the order of traced events/signals, rather than how much time lapses between them. It also used more coarse clocks for tracing distributed executions. With these techniques, researchers are able to verify much wider range of complex applications. However, when it comes to systems that interact in real-time with their environment, not only the order of events is relevant, but also the exact timing of these events measured according to some absolute time/clock. It becomes strongly required to measure this interval which expresses the environment reaction time.

2.5.1 Efficient Tracing

Traces size became a problem as systems complexity and speed grow. This lead to many advances in various directions. On the signal selection, many solutions, like [25, 75, 152] addressed automating signal selection for better visibility and observability. Others addressed traces compression, like [137, 183]. Addressing the limited on-chip storage, selecting which data are critical to store was also addressed, examples can be found in [6, 24, 152]. Compressing the tracing logs, either for simulation traces, as in [119]; or for on-chip embedded software executions as in [118] or [170]. Program trace compression –mentioned before– focused on branching and addresses/way-points, as the main base for compression. Besides these, pure compression techniques, like Lempel-Ziv (LZ)-based, have also been done in hardware in [84]; on the top of branching and address-encoding based compression.

Embedded software tracing had its own developments, ranging from using executable meta data to reduce the logged trace size. Processing cores specific meta data, in addition to the branching-based methodology, were used to reduce log sizes; as in ARM's Core-Sight [17]. However, the traces size remains far beyond being acceptable if we talk about deployment time. Hence, more efficiency is needed.

Trace storage on and/or off-chip, logging bandwidth and pins, off-line traces processing and/or reconstruction, are all issues which need to be addressed before tracing can be incorporated in a deployment time activity.

Novel tracing solutions, like using automata processors to enable efficient runtime tracing [150] is great in capturing problems, but requires a whole system to do the job.

Non-Intrusiveness It has been accepted in the industry that tracing execution will incur an internal overhead, a one which utilizes the processor resources, whether CPU or memory. This means tracing –slightly– changes the system behavior. The

change happens at both: post-silicon validation [100, 125, 177, 182] and embedded software debugging [17, 178]. While during development these practices are widely accepted, during deployment if the tracing facilities were removed, the system behavior might vary from the one having them.

Transparency By transparency we mean two things: 1) it is known and clear how the traces are generated, and 2) it is done by an independent module; not the system itself. The first means when some event happens, it is clear how the trace expressing this happening is created. For example, a system sending a messages can generate a trace (log) saying it sent the message with some specific content, at a specific time. Clarity requires that the source code generating this message is available, verified to be the one which generated the log, and the time exactly this code was run, with all the corresponding data which appears in the log are also available and can be verified by non-designers. The second, namely *independence*, means that traces informing about performance for example are not run by the system of which the performance is reported. Another example of dependent-trace, a task reporting it has finished in certain time. If the task itself is still reporting, how then it is done? An independent reporting has to be carried on by external independent observer; which can witness –unbiased– that the task is done. This is even more important in autonomous systems [164], where the details of performance should be reported.

These two requirements go very well, hand in hand with non-intrusiveness. A real independent observer will be non-intrusive. And clarity and openness can be used as a witness of non-intrusiveness; as well as the reliability of the trace in reflecting what happened accurately.

2.5.2 Existing Tracers and Debuggers

We give an overview of tracing in embedded software debugging and in general chips post-silicon validation activities. This is not meant to cover all possible literature or solutions; but rather it gives a representative example from each area.

Software Tracing

Examples of existing software tracing methodologies. Overview of features and drawbacks of selected trace modules. We try to cover the ones widely used. Here, a light differentiation is highlighted between embedded and general purpose processors. Examples of embedded systems processors tracing units are surveyed first,

because of their relevance to the timing accuracy of the tracing. They have more facilities to provide more accurate timing information about the execution of software on them. However, even within embedded cores, a major difference can be seen. Simple debug units as that of LEON3 processor for example, are just logging/storing blindly the executed instructions with their timestamps. While on the other hand, debug cores for more advanced processor families, like ARM and RISC-V, are focusing their traces around the branching behavior of the executed instructions.

Simple Debug Support Units The example we use here to illustrate this category is LEON3 DSU [69]. [5] This unit enables hardware breakpoints, by which one can control where to start/stop recording a trace of instructions executed. Traces are stored in an on-chip trace-buffer, which contains the executed instruction, its address, and a timestamp. The trace-buffer is cyclic; i.e. when it is full, writing new entries override the oldest ones.

As the name implies these are units dedicated to debugging, and they incur huge size of traces saved on-chip in trace-buffers; and the traces are logged when needed on a best effort bases. The LEON3 DSU example is open source.

Sophisticated Debug and Trace Units Like the simple debug support units, these are also traces for embedded software. But they provide much more efficient tracing capabilities. The optimizations are based on considering meta data while compiling the programs; and using these meta-data by the tracing units during execution, to generate small and efficient traces; logged to a host, which can recover the original executed instructions using dedicated software. Examples are ARM Core-Sight, MIPS , Lauterbach and Greenhill's debuggers/tracers [17, 56, 161, 162]. We use ARM Core-Sight to illustrate this category. Some work, as in [53], has built on these traces to automate checking temporal properties.

To use the most accurate and detailed tracing capabilities, one needs to compile the embedded software with compilation options which enable such type of tracing. The Core-Sight module then incurs very small size log (which can be as small as one bit per the processor's clock). This log is received by a host, which has a software installed and ready to receive the transmitted packets containing that log. The host software can decode the logged trace to extract the details of the program execution; up to clock-cycle-accurate level.

[5] LEON3 is also picked here because it is used in many of the experiments, conducted to apply the results from this thesis.

Although this solution seems efficient from log-size point of view, still one bit per second means as many bits as the processor's clock; which can still be a lot to log during deployment. Timeprints for example can incur many orders of magnitudes less bits than the processor's clock. Another point to consider is that this solution is not independent, it depends on ARM's proprietary hardware and software. These cannot provide the level of transparency needed for evidence, as they are not open sourced, nor freely available.

Nexus 5001 Standard for a Global Embedded Processor Debug Interface This is a partially open [157] standard for tracing embedded software; which was first introduced under the IEEE Standard 5001, as a Real Time Debug Instrumentation Architecture and methodology standard. It defines packet-based messaging between the traced processor and a host. Also a processor independent signaling is defined for accessing traced data by the host. It specifies details of using 1149 TAP -2 or 4 wire protocol, and JTAG. The specifications though still results in huge amounts of data; which makes it impractical to use during deployment.

RISC-V Trace Specifications RISC-V is a, relatively recent architecture (started in 2010 and first fixed release in 2016). It is mainly an open, extendable and configurable instruction set architecture. RISC-V also defines a tracing specifications, which were first released in March 2020, and the latest version can be found in [143]. The trace specifications are centered around what should be logged by each *hart*, the RISC-V hardware thread. Instead of logging all incidents of address discontinuities, the tracer is required only to log the address if it is non-inferrable Jump. It is required also to log the type of the address discontinuity, as well as whether a branch was taken or not. Traps and interrupt causes are also required to be logged. This makes the size of traces much bigger than Core-Sight's smallest trace. In the first version of the trace-specifications, it is allowed that the tracing can be configured to stall the execution if trace-logging asks for it. This particular configuration allows for intrusiveness, and when it is not used some of the trace data might be missed; as the case was with LEON3's trace-buffers.

However, there is a great potential for more efficient traces here, as RISC-V is an open ISA and most of its implementations are also so. Efficient traces, as Timeprints, can be added to many variants of RISC-V cores, transparently and independently.

General Signals Tracing

All the previously mentioned tracing tools and technologies are focused on tracing embedded software. To trace any arbitrary signal on-chip; exploiting the specifics of software execution (an incremented program counter, branching, ...etc) is not going to be possible. We survey here briefly techniques used in different areas, for tracing on-chip signals.

ASIC and FPGA Signals Tracing The various FPGA manufacturers provide their own solutions for tracing user-selected signals. Usually hardware module templates are provided and auto-generated, which reside with the design-hardware and support the signals capturing and logging. Examples are Xilinx Chip-scope pro [45] and Altera's SignalTap II Logic Analyzer [89].

For ASIC, Multi-input Shift Registers (MISR) are inserted on chips, and used for debugging them after production (post-silicon) [120, 154]. They contain data which can be inspected, to check correctness of implementations; and to trace the error root-cause when there are problems. They cannot, however, cover errors related to small path delays, and accumulated timing errors.

Other techniques like Test-Points, scan-chains, ..etc are also used [120]. But these are very tied to manufacturing time, as they require sophisticated hardware –used to inspect many chips– and visibility into the chip itself. Hence, they cannot be used in-field during deployment.

Test-Equipment for Protocols Testing For testing specific systems and communication protocols, dedicated devices, usually called *test equipment* are developed and used. They check streams of data versus protocol or test specifications. By being very specific; they can analyze huge amounts of data for compliance. They are usually huge in size and specialized in specific areas, systems and/or protocols. Such devices cannot by defaults be used –as they are– in-field; unless they are already designed for such purpose, for example: in-field protocol sniffers. These later devices are not meant for tracing compliance or providing evidence; but have other purposes.

2.6 Chapter Summary

In this chapter, we summarized the tracing problems and classified them. Problems related to accuracy, and those related to temporal properties description, and lastly practical problems, were presented. A representative sample of the existing tracing

methods, were presented and assessed, from the perspectives of: efficiency, intrusiveness, transparency and independence. This overview shows how there are still strong need and room, for better tracing which can lay itself as in-field transparent and independent evidence.

This serves as preparation to start introducing our contribution in the next chapter. In Chapter 3, an overview of our Timeprints and evidence-based tracing methodology is presented briefly as a guide to the rest of the detailed chapters.

Chapter 3

Timeprints Overview

At the heart of our proposed methodology of evidence-oriented tracing, lies the *Timeprints*. The target is tracing, evidently, cycle-accurately and efficiently, the temporal behavior of signals. Unlike most tracing methods which are used during the development life-cycle, especially in debugging and testing, *Timeprints* are mainly designed with in-field operation in mind. Current cycle-accurate tracing schemes incur unacceptable amounts of data for logging, storage and processing, or use proprietary hardware and software which makes them not transparent nor independent, or they do require huge and sophisticated machinery, that are only suitable for use in manufacturing or testing facilities.

The key idea, by which Timeprints enable efficient yet cycle-accurate tracing, is bringing time to the front as a main traced artifact. This means, instead of tracing values and associating a meta data describing time with it, we trace the timing at which values of the traced signal change. Our Timeprints-based approach depends on transforming the continuous observation and measurement task, into periodic logs summarizing the temporal behavior over continuous periods; which we call trace-cycles. Signal tracing is split into consecutive (back-to-back) finite trace-cycles. Within a trace-cycle, a signal's value-change instance gets assigned an encoded timestamp. At the end of each trace-cycle, these encoded timestamps are aggregated into a logged timeprint, which summarizes the temporal behavior over the trace-cycle. To retrieve the accurate timing, reconstruction is needed to deduce the exact instances from a timeprint. This reconstruction is a formal operation, that takes the timestamps encoding and system plus environment properties as input, and produces the exact timing or certificates about temporal behavior, as output.

This chapter gives a high level insight into the proposed Timeprints. It explains –with adequate level of detail– what they are, how they get generated, how traced signals can be reconstructed from them, and how properties can be formally checked based on them. In this sense, the chapter functions as a core of the thesis. But

still, it does not delve into theoretical details (which are the focus of the theoretical-foundation's next four chapters), nor it discusses application aspects (which is the focus of the last two chapters). Rather, the chapter acts as an efficient summary of the main contribution, so that a reader can be fully aware of what Timeprints are after reading it, and is capable on consulting the respective other chapters or parts of thesis as needed. Hence, the chapter is also a key and manual for the rest of chapters.

The chapter begins with explaining the key, *bringing time to the front*. Then the basic elements of Timeprints, their generation, reconstruction, and usage principles are given. Finally, the thesis contribution is highlighted, publications made to date are listed, and new doors opened are knocked. By the end of reading this chapter, the reader should be able to state what Timeprints are, and can start using them.

3.1 Bringing Time to the Front

Traditionally, when a signal is traced, the focus is all centered on the signal itself. So, an observer monitors, measures, probes or samples the Signal under Tracing (SuT). This means capturing several values of SuT, separated by some time lapse t. In this traditional tracing task, the main focus is the value of the traced signal. And the time at which any traced signal value took place, is either implicitly defined by the sampling rate, if t was fixed, or explicitly attached to the sample as a timestamp, when sampling is is done based on non- fixed t criteria.

When fixed sampling rate is used for tracing, the resulting data are consecutive measured values, known to be taken within constant period t_s, known as the sampling rate. These values then are either recorded locally and/or logged to a host for external storage. When pairs of measurements and their timestamps are used, such pairs are what gets recorded and stored. In the former case, the required size of basic data generated during tracing is constant ($t_s \times samplesize$), over the passage of time. In the later case, where the pairs of measurements and their timestamps are generated from tracing, the size of data depends on how frequent the signal changes. In both cases, the main recorded value, is the value of the signal SuT itself, beside the implicit or explicit timing at which the recorded value has taken place, as an attribute.

Bringing time to the front, means doing the opposite: i.e. instead of focusing on the measured value as the main traced artifact, we shift the focus to tracing the *timing*, at which these values takes place.

Timeprints enable efficient tracing by trying **not** to log the information –properties– we already know about the signal. What gets logged is something that can enable

us to check *accurately* what happened when doubt is raised. So, we use the *properties* already known to be holding, to check the properties we are suspecting, or even to restore the exact occurrences when we have no clue at all. Figure 3.1 presents a clarifying classification of signal's properties from cycle-accurate perspective.

After the signal to be traced is identified, properties of that signal are used in the development phase, to decide about which monitors/checks shall be implemented. *Signal-Analysis* is done at the *design-phase*, to explore different types of properties. This would result in a subset of the describable (definable) properties, which can be formally defined down to the cycle-accurate level, as illustrated by the subsets in Figure 3.1. Among the defined-properties, some can be checked using hardware synthesized monitors, constituting another subset of the defined properties. Such monitors verify in run-time that certain properties hold about the traced signal [20], or reason about combinations/history of them as in [123]. They can also be used in WCET analysis as in [7], and can even correct/enforce the behavior if the property is violated during run-time [133]. Due to on-chip space limitation, only a subset of the synthesizable monitors gets implemented. The darkest area in the *Timeprints* set in Figure 3.1, represents the definable properties which were **undefined at design-time** on the cycle-accurate level, and timeprints can enable tracing them.

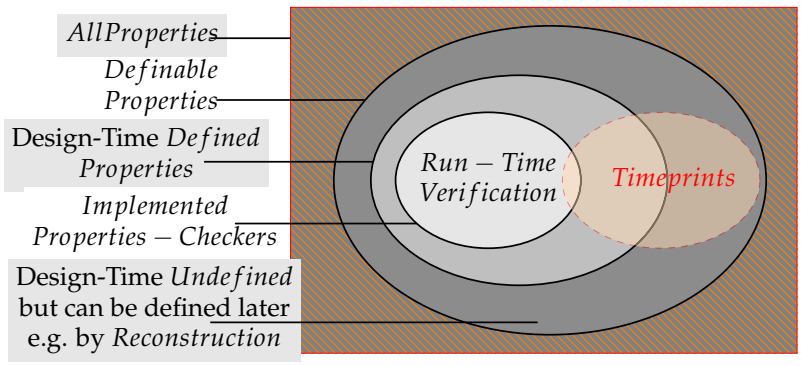


FIGURE 3.1: Cycle-Accurate Properties Classification

In our methodology flow, the *defined properties*, resulting from the signal-analysis step do not only result in the run-time verification monitors, but are also used to decide the size and encoding of the timestamps, as shall be seen later in Section 3.3. Verified properties and other assumed properties can be used to retrieve/deduce/prove properties which were not defined at design-, nor run-time.

3.2 Timeprints Generation

The following example is going to be used to explain how timeprints are first generated, then used to extract information. Consider the scenario of autonomous car crash; where the obstacle was identified, the car slowed down but still hit the obstacle. We assume that every module exactly behaved according to the logged diagnostic information, which narrowed the suspicions down to the exact timing of sending signal S_t by chip C_1 to chip C_2. Having the infinite discrete signal S_t to trace at hand, we split it into *trace-cycles* that contain $m = 16$ clock-cycles each. In Figure 3.2, on the left-hand side, a trace-cycle j is depicted, where four changes in S_t's value $V_1, \ldots, V_4$ took place. If S_t was sent by C_1 *before* a specified deadline, that lies at the d^{th} clock-cycle in the trace-cycle j, then C_2 is responsible for the overall system's delayed response. But if S_t was sent *after* the deadline, C_1 is responsible.

A common way to record the behavior of a signal like S_t, is to log the precise timings. This means we log a number with $\log(m)$ bits each time the value of S_t changes. In Figure 3.2, we have 4 changes. Hence, we log $4 \cdot 4 = 16$ bits. In general, when the number of changes is denoted by $k \in \mathbb{N}$, we need to log $k \cdot \log(m)$ bits each trace-cycle. Thus, the amount of logged information depends linearly on k. As k varies from one trace-cycle to another, the amount of logged bits also does. This makes processing or searching through logged information difficult. Using one pin for logging, during a trace-cycle of length m, the maximum number of bits that can be logged each trace-cycle is m. If we need $\log(m)$ bits for each change, we can record at most $m/\log(m)$ of them.

For the method, we identify clock-cycles by encoded timestamps. These are bitvectors of a fixed dimension (or size) b, where $m \geq b \geq \log(m)$. In Figure 3.2, we identify the i-th clock-cycle by a timestamp $TE(i)$. We use $b = 8$ bits for the 16 timestamps.

Initially, we use a b-bit hold vector, with all bits set to 0. When a change in S_t occurs, a corresponding *change-signal* $S(i)$ goes to 1, causing the corresponding timestamp to be aggregated –here added– to the currently hold vector. At the end of the trace-cycle, this results in a b-bit vector TP, called the timeprint. Since the vector is a summary of the traced signal's behavior, our method logs TP. In Figure 3.2, we aggregate/add the timestamps $TE(4), TE(5), TE(10)$, and $TE(11)$. The resulting timeprint TP is the sum of these timestamps: $TP = (0,0,0,0,0,0,0,1)$. TP is shown in the middle of Figure 3.2, as an 8-bit vector. Notice that modulo 2 addition is just bitwise XOR. Generating the Timeprints is a kind of abstraction; and it is discussed in detail in Chapter 4.

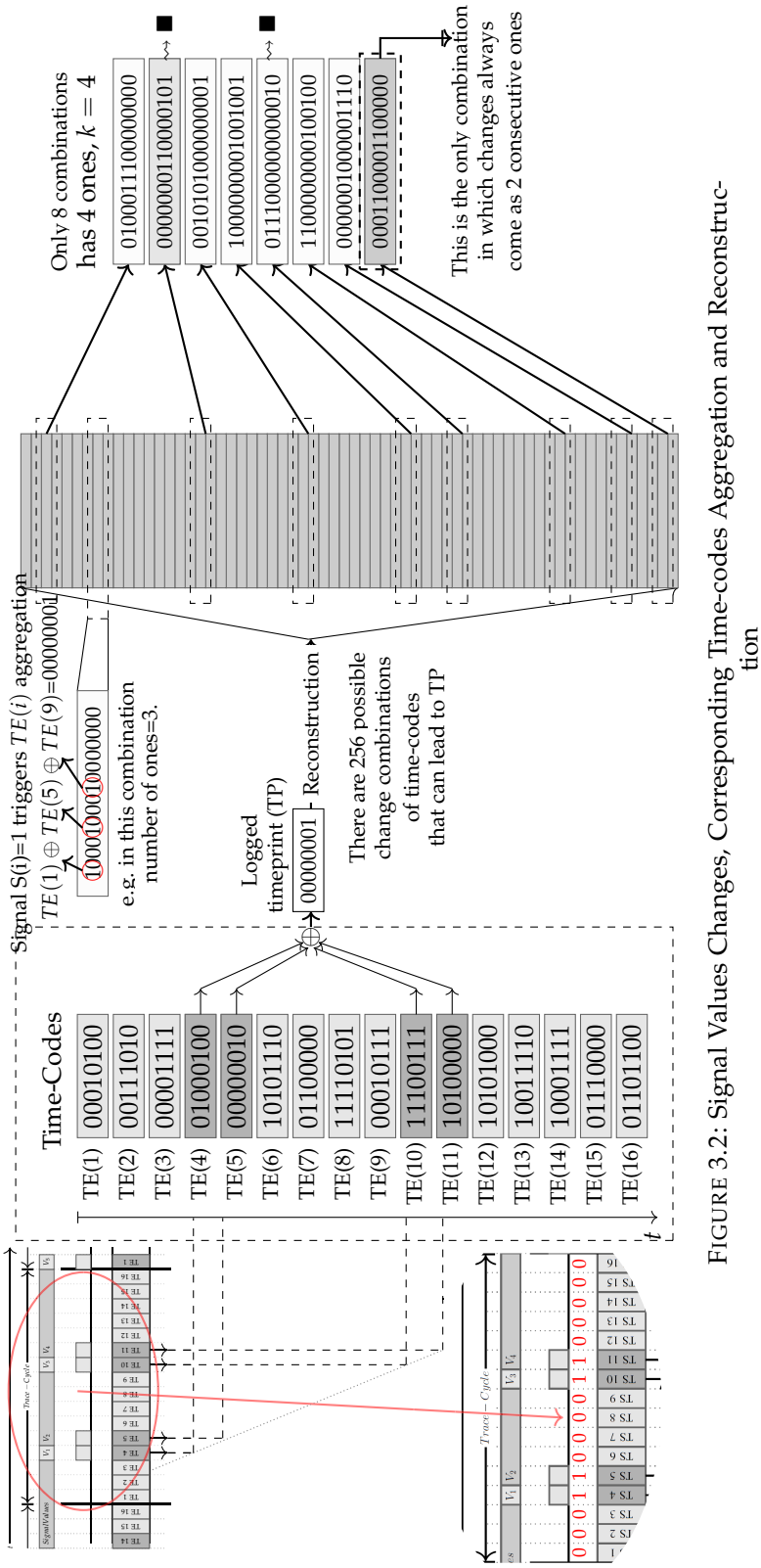


FIGURE 3.2: Signal Values Changes, Corresponding Time-codes Aggregation and Reconstruction

Bitwise XOR function can be implemented easily on hardware and is bit-width preserving. When we use the XOR as an aggregation function, the number of changes is lost. So, we record the precise number of changes that happened during a trace-cycle in a counter k that is increased each time a change shows up. Since there are at most m changes, we can encode k into $\log(m)$ bits. We show later, that k plays a central role in the reconstruction problem. During a trace-cycle, our method always logs $b + \log(m)$ bits in total: b for the timeprint TP, $\log(m)$ for the counter k. This enforces a constant number of logged bits each trace-cycle, irrespective of k.

In a trace-cycle, the only information logged by our method are: the timeprint and the number of changes in the traced signal. This may create ambiguity: there might be different change-signals leading to the same timeprint. Finding these is called *reconstruction problem*.

For the aggregated timeprint $(0, 0, 0, 0, 0, 0, 0, 1)$ in Figure 3.2, there are more solutions besides the actual *change-signal* (we'll call it for simplicity *signal*): $TE(4) + TE(5) + TE(10) + TE(11)$, for example, $TE(1) + TE(5) + TE(9)$. In total, there are 256 signals whose timestamps sum up to TP. The number of possible reconstructions increases with the decrease in the timeprint size. Until now we have not yet taken into account all of the logged data. Since we also know that the actual signal has exactly k changes, we can exclude those violating this requirement. This leads to a formal problem description:
"Given a timeprint TP and a number k of changes, find all signals with k changes, whose timestamps add up to TP."

For our example, limiting the number of changes to 4 decreases the number of candidate signals from 256 to only 8. A list of these is shown on the right-hand side of Figure 3.2. It is worth noting that the reconstruction problem crucially depends on the chosen timestamps. Linear independent timestamps would cause no ambiguity but the timeprint width would then be m-bit which is too big. On the otherhand, too compressed timestamps would increase the degree of ambiguity.

To isolate the actual signal in the reconstruction problem, we utilize information obtained from the *defined and verified* properties.

Back to our original scenario, where the traced signal S_t was sent from chip C_1 to chip C_2, we know that these are timings of writes of new S_t values; so if we know that such writes always last for one cycle (by some verified specifications) after which S_t goes to zero and remains there until a new value of S_t is written, we can conclude that the changes in the signal (represented by ones), has to occur in two consecutive clock cycles; and hence the last shaded row is actually the one that happened and lead to our footprint. In our example, it was not an accident that we had at

the end one possibility and we became sure of what accurately happened. We utilized the property (of 2 consecutive value's changes), to choose a set of timestamps that would, in most cases, result in a unique reconstruction for each couple of pairs. However, sometimes we cannot exclude all the non-actual traces. For example, consider the case in which the property of 2 consecutive value's changes, is not run-time verified. Rather, either the two changes would happen consecutively or separated by at most one clock-cycle. In this case, the second and fifth possibilities (2^{nd} and 5^{th} rows, marked with ■), in Figure 3.2, would still be candidates.

Often, we do not need to find the actual signal, rather it is enough to check whether all signals that might lead to the logged timeprint, satisfies or breaks a certain safety property. Consider the case of meeting a deadline. Assume that the deadline is given by $i = 8$. In Figure 3.2, we can see that all 8 possible reconstructed signals have a 1-bit already before the 8-th position. This means that all traces involving 4 changes that aggregate to TP meet the deadline, no matter which one actually took place. So, it may happen that the information given by the temporal property does not suffice to isolate a single signal reconstruction (i.e. a single cycle-accurate instances). But often, we only want to know whether there is a trace that satisfies or breaks a certain temporal property.

3.2.1 Formal Definition

Before we delve into the formulation, we clarify some notation used all over the thesis. We use TP to refer to any arbitrary Timeprint, while we use tp to denote a specific Timeprints trace (logged), and use tp_i to denote the i^{th} Timeprint in a trace $tp = tp_1.tp_2....tp_i.....$ We use both exchange-ably when it is clear from the context.

We formalize the logging procedure, prove it to be a sound abstraction, and present a solution for the reconstruction problem. The underlying domain is $\mathbf{F}$, the field consisting of $0, 1$ with addition and multiplication modulo 2. Let $n \in \mathbb{N}$. We write $\mathbf{F}^n$ for the n-dimensional vector space over $\mathbf{F}$. This is the set of n-bit vectors where addition is bitwise XOR.

We trace signals over *trace-cycles* of *length* m, with $m \in \mathbb{N}$. These are periods containing m clock-cycles. Let such a trace-cycle be fixed. Formally, a *signal* is a map $S : [1..m] \to \{0, 1\}$, where $S(i) = 1$ indicates a change of the traced-signal in the i-th clock-cycle. By *Sig*, we denote the set of all such signals.

Fix $b \in \mathbb{N}$ with $m \geq b \geq \log(m)$. We define the logging relative to an *encoding*. An encoding is an injective map $TE : [1..m] \to \mathbf{F}^b$, which assigns each clock-cycle a unique *time-code*. The logging procedure abstracts a signal S to a pair (TP, k), where

$TP \in \mathbf{F}^b$ is called *timeprint* and k is the number of changes in S. Such a pair is called a *log entry*. The set of all possible log entries is $Log = \mathbf{F}^b \times [1..m]$.

Formally, for a fixed encoding TE, the logging procedure implements the function $\tilde{\alpha}_{TE} : Sig \to Log$. Given a signal S, the function returns a log entry $\tilde{\alpha}_{TE}(S) = (TP, k)$ with $TP = \sum_{i:S(i)=1} TE(i)$ and $k = |\{i \mid S(i) = 1\}|$. Note that the timeprint TP is the sum over those time-codes where the traced signal changes. Moreover, there are exactly k changes in S. Notice that we use the terms *Log* and *trace* exchange-ably when clear from the context.

Soundness

We show that the logging procedure constitutes a sound abstraction of signals. To this end, we establish a Galois insertion between the domain of signals and the domain of log entries.

Formally, the domain we abstract from is the powerset lattice $\mathscr{P}(Sig)$. The abstract domain is the powerset lattice over log entries, $\mathscr{P}(Log)$. Fix an encoding $TE : [1..m] \to \mathbf{F}^b$. For the abstraction function, we apply the logging procedure to sets of signals.
Let $\alpha_{TE} : \mathscr{P}(Sig) \to \mathscr{P}(Log)$ be the lifting of $\tilde{\alpha}_{TE}$, defined by $\alpha_{TE}(F) = \bigcup_{S \in F} \{\tilde{\alpha}_{TE}(S)\}$, where $F \in \mathscr{P}(Sig)$.

For the concretization $\gamma_{TE} : \mathscr{P}(Log) \to \mathscr{P}(Sig)$, we first define an auxiliary function $\tilde{\gamma}_{TE} : Log \to \mathscr{P}(Sig)$. It maps a log entry (TP, k) to its preimage under $\tilde{\alpha}_{TE}$, the set $\{S \in Sig \mid \tilde{\alpha}_{TE}(S) = (TP, k)\}$. Then γ_{TE} is the lifting of $\tilde{\gamma}_{TE}$ to sets of log entries, defined as above.

Both functions, α_{TE} and γ_{TE} are monotonic by definition: We have that $\alpha_{TE}(F) \subseteq \alpha_{TE}(G)$ for $F \subseteq G$ sets of signals, and similar for γ_{TE}. Moreover, the functions establish a Galois insertion as shown in the next lemma.

Lemma 1 *Let TE be an encoding. For each $F \in \mathscr{P}(Sig)$ we have $F \subseteq \gamma_{TE}(\alpha_{TE}(F))$. Moreover, for each $V \in \mathscr{P}(Log)$ we have the equality $V = \alpha_{TE}(\gamma_{TE}(V))$.*

The proof follows directly from the above definitions.

The Timeprints as abstractions are covered in more detail in Chapter 4.

3.2.2 Trace Reconstruction

Since the logging procedure is an abstraction, there might be different signals that result in the same log entry. Finding all signals that get abstracted to a particular

entry is what we refer to as the signal reconstruction problem. We give a formal definition of the problem and an idea of an efficient SAT-based solution.

Let $(TP, k) \in Log$ be the output of the logging procedure. Reconstructing all signals S that get abstracted to (TP, k) is the task of computing the preimage of (TP, k) under the map $\tilde{\alpha}_{TE}$. We define the *Signal Reconstruction* problem as follows:

Signal Reconstruction ($\mathscr{R}(S)$)	
Input:	Encoding $TE : [1..m] \to \mathbf{F}^b$, timeprint $TP \in \mathbf{F}^b$, $k \in \mathbb{N}$.
Task:	Find all signals S with $\tilde{\alpha}_{TE}(S) = (TP, k)$.

We can state an equivalent form of $\mathscr{R}(S)$ in terms of linear algebra. Let $A = [TE(1) \mid \cdots \mid TE(m)] \in \mathbf{F}^{b \times m}$ be the matrix consisting of all timestamps. $\mathscr{R}(S)$ is equivalent to finding all solutions $x \in \mathbf{F}^m$ of the system $Ax = TP$, where x has exactly k entries set to 1. Each solution x represents a signal S with $\tilde{\alpha}_{TE}(S) = (TP, k)$ and vice versa.

Variants of this problem are well-known in coding theory [172]. Moreover, $\mathscr{R}(S)$ is known to be NP-hard [30] and it is therefore unlikely that it can be solved in polynomial time.

To solve $\mathscr{R}(S)$, we use satisfiability SAT solvers; and encode our problem as input to them. Usually, the input to a SAT solver is a formula in CNF. However, we use an extension which also allows clauses of XORed variables. Those are particularly helpful when encoding linear equations. A solver supporting this input format is *Cryptominisat* [158]; the solver which we used and slightly modified to solve $\mathscr{R}(S)$.

For the encoding, we introduce m variables $x_1, \ldots, x_m$. These are the bits of a solution x in the system $Ax = TP$. Intuitively, setting x_i to 1 amounts to a signal that has a change in the i-th clock cycle. This means that timestamp $TE(i)$ is *chosen* and summed up to arrive at TP. The linear equations $Ax = TP$ are modeled by b clauses $C_1, \ldots, C_b$. A variable x_i occurs in C_j if the j-th bit of $TE(i)$ is 1. All the variables in C_j are XORed. If the j-th bit of TP is 0, the clause C_j gets negated, a feature supported by *Cryptominisat*. Hence, the clause represents exactly the j-th equation of $Ax = TP$.

What is left to encode is the cardinality constraint over the variables. We have to choose exactly k out of the m variables and set them to 1. A naive encoding would use $\binom{m}{k+1} + \binom{m}{m-k+1}$ clauses, resulting in an intractable SAT query. We use the efficient and compact cardinality encoding proposed in [155], which introduces $\mathcal{O}(m \cdot k)$ additional variables and needs only $\mathcal{O}(m \cdot k)$ clauses to express the constraint.

Chapter 6 addresses reconstruction, in detail, both theoretically and with some practical aspects.

3.2.3 Time Encoding

The choice of the time-codes has influence on the ambiguity occurring within the logging procedure and thus on the time needed to solve $\mathscr{R}(S)$. Intuitively, a sparse choice of time-codes allows only for few possibilities to sum up to the timeprint. It decreases the number of solutions of $Ax = TP$, making it easier to find all of them. However, we can only allow sparsity up to a certain extend as the number of logged bits would grow.

Ideally, we would choose a time encoding that avoids ambiguity at all. This can be achieved by constructing an encoding $TE : [1..m] \to \mathbf{F}^b$, where $TE(1), \ldots, TE(m)$ are linearly independent vectors. Then, the system $Ax = TP$ has a unique solution and $\mathscr{R}(S)$ can be solved quite fast. For example, a *one-hot encoding* would be of this type. However, choosing m linearly independent vectors requires that the dimension of $\mathbf{F}^b$ is m, hence $b = m$. But then, the number of bits we need to log depends linearly on m, contradicting our goal to establish a space-efficient logging procedure.

We can achieve a trade-off in the choice of time-encoding by requiring linear independence only up to a depth d. That means for each unique subset of time-codes $T \subseteq TE([1..m])$ of size d, the timeprint is unique. As d grows, the number of solutions to $\mathscr{R}(S)$ decreases, but the number of logged bits b increases. Currently, we fix $d = 4$ and approximate TE and b using a practical heuristic. Computing an encoding with the smallest possible b is an open problem for future research.

Chapter 5 discusses Time-encoding in more detail.

3.3 Timeprint Design Parameters

By carefully choosing trace-cycle length and time-encoding, one can steer the number of solutions and the amount of computations needed to reach them. We discuss here briefly some of these parameters and how would their choice affect the whole tracing functionality.

3.3.1 Trace-cycle length m

The choice of m affects directly the amount of logging. When b is the bit-width of the timeprint, the bit rate required for logging is: $(b + log(m))/m$ multiplied by the maximum clock-rate. Increasing m would decrease the log-rate, but the average

m,k	b	c-SAT.1	c-SAT.10	c+P2.1	c+P2.10	c+Dk.1	c+Dk.10	c+Dk+P2.1	c+Dk+P2.10	R
64/3	b=13	0m0.010s	0m0.085s	0m0.026s	0m0.092s	0m0.023s	0m0.027s	00m0.049s	0m0.022s	20.97 MHz
64/4		0m0.048s	0m0.149s	0m0.014s	0m0.192s	0m0.023s	0m0.044s	0m0.032s	0m0.036s	
64/8		0m0.019s	0m0.101s	0m0.021s	0m0.175s	0m0.029s	0m0.222s	0m0.045s	0m0.221s	
64/32		0m0.032s	0m0.023s	0m0.013s	0m0.058s	0m0.013s	0m0.024s	0m0.041s	0m0.030s	
128/3	b=16	0m0.124s	0m0.709s	0m0.127s	0m0.817s	0m0.108s	0m0.170s	0m0.125s	0m0.131s	12.5 MHz
128/4		0m0.116s	0m0.610s	0m0.274s	0m0.928s	0m0.085s	0m0.223s	0m0.157s	0m0.313s	
128/8		0m0.118s	0m0.774s	0m1.070s	0m2.122s	0m0.156s	0m0.460s	0m0.259s	0m0.948s	
128/16		0m0.087s	0m0.149s	0m0.035s	0m0.228s	0m0.036s	0m0.261s	0m0.102s	0m0.174s	
512/3	b=22	0m1.714s	1m50.343s	0m29.912s	0m33.847s	0m1.479s	0m1.514s	0m0.427s	0m0.393s	4.3 MHz
512/4		0m42.638s	1m50.312s	1m3.351s	2m2.762s	0m1.563s	0m1.551s	0m1.901s	0m1.892s	
512/8		0m44.025s	1m57.296s	0m52.319s	2m46.793s	0m4.226s	0m14.590s	0m8.770s	0m21.824s	
1024/3	b=24	1m36.234s	18m29.931s	4m7.147s	4m5.050s	0m2.747s	0m2.624s	0m1.380s	0m1.687s	2.3 MHz
1024/4		3m42.684s	18m0.121s	10m28.096s	24m48.517s	0m4.567s	0m5.443s	0m23.424s	0m21.586s	
1024/8		3m33.685s	15m14.368s	7m37.488s	22m10.561s	0m8.891s	1m23.997s	0m4.949s	2m3.061s	

TABLE 3.1: Effect of choice of m,k on reconstruction time.

number of changes k, would increase, increasing the number of solutions, and the reconstruction time. [1].

Unlike the illustration example which used small m, practical values of m are 512, 1024, or even more. For such m, the reduction in log-rate gets 2 to 3 orders of magnitude less than logging one bit per clock-cycle (which take the system's clock-cycle rate, and requires no reconstruction). For example, in trace-cycle of length 512 and $b = 22$, we only $\log 22 + log512 = 31$ bits. For $m = 1024$, we $\log 24 + log1024 = 34$ bits.

Each row, in Table 3.1, represents the reconstruction time for certain trace-cycle length m and number of changes k. At the end of each row, the log-rate R required for a signal of 100 MHz is given. The first column gives time needed until reaching the first candidate change-marker CM, with encoded k cardinality; so we name the column n-SAT.1, for n constrained SAT solving to the first solution. The second column, n-SAT.10 gives the time until the 10^{th} satisfying solution, or until no more solutions are found, if there already less than 10 solutions.

3.3.2 Time Encoding Bit-width and Algorithms

In Random-constrained time-code generation: time-codes are generated randomly, while checking for the LI-4 condition. Another encoding mechanism, starts by smallest possible time-codes that fulfills linear independence of degree 2; then we increment and check that linear independence still holds. We call this incremental time-codes encoding; from which we obtained b, shown in Table 3.2's b columns. This encoding leads to smaller b, and less average reconstruction times than the random-constrained encoding, see Table 3.2. Scalability of encoding and the role of choosing

[1]The example log-rate at the last column of Table 3.1 is assuming system's clk is B bits per second, where $B = 100MHz$.

the time-codes width b on both the codes generation and reconstruction is discussed in details in Chapter 5.

Along the thesis and within the experiments, we used many generic heuristics, which starts by a first guess or smallest possible time-code, and then look for the next code that fulfills, for example, LI-4; then we proceed and check that the condition still holds. Comparison of the SAT based reconstruction times and the SMT reconstruction algorithm's run time, for the same set of time-codes, was given in [113].

In the table, we also show the run-time of solving the SAT problems which includes the properties $P2$ and PDk encodings, which are explained next.

3.3.3 Encoding Temporal Properties

As mentioned earlier, the additional encoding of temporal properties of a signal into $\mathscr{R}(S)$ helps pruning the search space and reduces the SAT solving time. When the property is a violation (satisfaction) of some safety property, then we only require one SAT-solution (UNSAT) to prove (disprove) that a violation has (not) happened. For illustration we consider two simple properties, $P2$: 2 consecutive time-codes would appear at least once, and Dk: at least k changes happen before a deadline D. These are the properties for which the reconstruction time was given in Table 3.1.

Columns marked with c-SAT, in Tables 3.1 and 3.2, correspond to solving time of $\mathscr{R}(S)$ (i.e. only using cardinality constraints c=k). The columns marked with $P2$ and Dk, in Table 3.1, show the solving time under additional constraints imposed by $P2$ and Dk of $k = 3, D = 32$, respectively. For each columns pair, the first gives time until reaching the first candidate, while the second (c-SAT.10) gives time until the 10^{th} satisfying solution, or until no more solutions is found, if there are already less than 10. As can be seen, $P2$ is less efficient than Dk, because it is also a weaker property. Using $P2$ and Dk, together, reduces the solving time even more than Dk alone (column $c + Dk + P2$). Notice that, in Tables 1 and 2, we just used some specific timeprints, and that the time is affected by the timeprint itself, and not only the properties[2]. We can model properties defined in [102]. Details of the temporal properties encoding will be discussed later in Chapter 6, section 6.2.

3.3.4 Checking Temporal Properties

Now that the properties are encoded, as just explained in the previous section, we look here at how temporal properties can be checked within one trace-cycle. (Notice that global properties can also be projected down over trace-cycles of interest). In the

[2]Results were obtained on an Intel i7-7500U CPU @ 2.70GHz x 4 with 15.6 GiB memory.

Time encoding:		Random-Constrained Time-codes			
m/k	b	c-SAT	c+P2	c+Dk	c+Dk+P2
512/3	b=31	2m57s	0m0.340s	4m39.727s	0m0.270s
512/4		33m17	13m33.423s	3m26.192s	1m51.532s
1024/3		11m39s	0m1.063s	22m26.209s	0m0.987s
Time encoding:		Incremental Time-codes			
m/k	b	c-SAT	c+P2	c+Dk	c+Dk+P2
512/3	22	0m1.714s	0m29.912s	0m1.479s	0m0.427s
512/4	22	0m42.638s	1m3.351s	0m1.563s	0m1.901s
1024/3	24	1m36.234s	4m7.147s	0m2.747s	0m1.380s

TABLE 3.2: Different time encoding schemes effect on reconstruction time

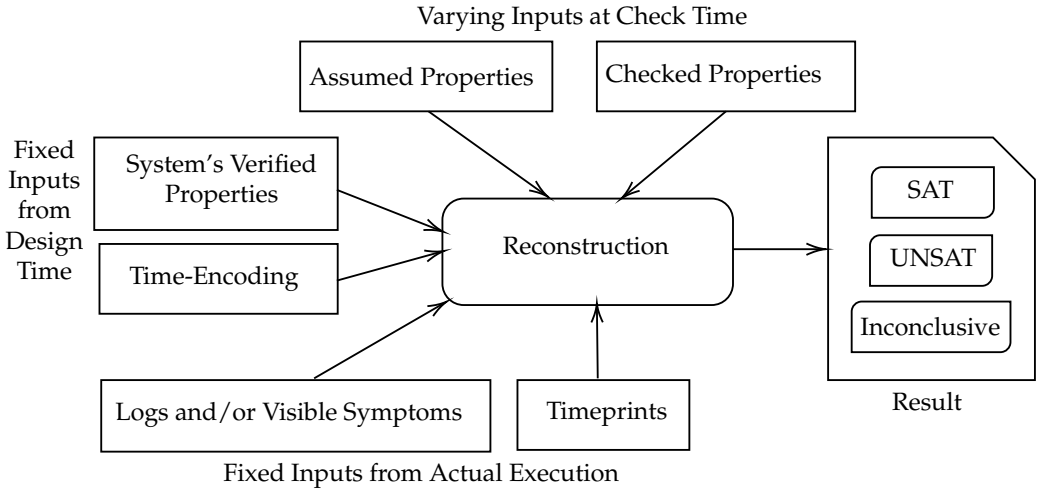


FIGURE 3.3: Checking Properties using Timeprints via Reconstruction

previous subsection, their main role was to prune the SAT problem search space to get faster and more precise reconstruction. Another main usage, for which we also encode properties, is to check these properties. To explain, let's consider the following example: we want to check that some n^{th} change in a trace-cycle has happened before its deadline; which lies in the same trace-cycle. We can encode a property, let's call it $p1$ which says n changes happened before the deadline, and check its satisfiablity.

In this example, if the reconstruction with this property is SAT, it means: it is possible that the n^{th} changes happened before the deadline; but not sure. To ensure that we have to also encode the property that the n^{th} change happened after the deadline; if this one is UNSAT, then we are sure that *given the logged Timeprint* the n^{th} change happened before the deadline. If this check gave SAT (i.e. it could be that the n^{th} change happened after the deadline), then the logged Timeprint is inconclusive about the property; as its reconstruction contains solutions, in which some have the n^{th} change before and some have it after the deadline.

Figure 3.3 shows the various inputs that can be used to do the reconstruction. It is not only the Time-Encoding and the system's verified properties that get encoded into reconstruction problem. These are the parts of the reconstruction-problem encoding that are known since the design time. At the execution time, the timeprints which were logged over the time-window for which we want to check what happened are also encoded into the reconstruction problem instance. Similar to Timeprints (in the sense that they come from the actual execution), are visible symptoms, and trusted logs from the execution itself. Moreover, when tracing sporadic and unexpected behaviors for which the reconstruction-space still needs to be pruned, various assumptions can be used (assumed properties in Figure 3.3). Similar to assumptions, properties that we may check can be defined at the time we are doing the check; hence, at post-mortem, we can check properties we have not think about during design, nor run-time.

In Chapter 7 we define when a Timeprint can be used to check a property, which maps to the Timeprints evidence; hence the chapter name: *Evidence of Timeprints*.

3.4 Main Contribution

The thesis contributes a novel evidence-oriented *Timeprints*-based tracing methodology. Timeprints provide cycle-accurate temporal traces of signals execution, which enable checking temporal properties. The amount of logging required to have *timeprints* traces, makes it acceptable for deployment. Being of extreme light-weight, *Timeprints* open the door for the first time, to a wide range of deployment-phase accurate timing-properties verification. The reconstruction method provided is generic, and the computational effort can be enhanced by coordination with simulation-guided and smart search methods.

Our approach to tracing is unique in its evidence orientation. Unlike most existing software run-time verification and hardware monitoring techniques, Timeprints are design independent and completely transparent. We got there because we wanted to find evidence of what have taken place; in well-designed and verified systems. This can help greatly in capturing sporadic faults, undefined-specifications, and unexpected breaches. With Timeprints, any misuse of a system can be captured; and early detection of problems can be better enabled.

3.4.1 Theoretical

From the theoretical perspective, although we didn't contribute any new mathematical or scientific theory, we did use existing formulations and theories with a completely new perspective, to solve the long standing in-field tracing problem; efficiently and with promising success. In Chapter 4, we reuse some of the existing metric temporal logic formulations to describe the evidence oriented tracing. We also contributed Time-Encoding (Chapter 5), in the sense of Linearly independent codes of instance within a trace-cycle; which is a corner stone in our Timeprints-based tracing methodology. The reconstruction problem, and how it can be tackled effectively is the topic of Chapter 6.

Checking traces via reconstruction has the potential to serve as evidence; even in courts. This is the topic discussed in Chapter 7. Although all the formulations and theorems provided here are stated in terms of tracing one signal, in Chapter 8, we show how they can be extended to multiple signals and system level tracing. The presented methodology can be extended in many theoretical directions. These are highlighted in the last chapter, where we conclude the thesis.

3.4.2 Practical

We applied our method to practical realistic case-studies. In Chapter 9, we apply it to tracing AHB bus address signal change, by which we captured memory-refresh instances which depends of temperature. We also traced messages exchanged over the automotive CAN bus. Our Timeprints can prove delays taking place over the bus, which makes it possible to be used as evidence if needed; which determine the module responsible for a system's delayed action.

Properties usage in reconstruction algorithms, as shown in Chapter 6 is illustrated by many practical experiments in Chapter 9. Practical Time-Encoding generation algorithms are presented in Chapter 5, together with the possibility of using a signal-oriented choice of time-codes. The usage of properties is also illustrated by an ultrasonic sonic sensor experiment, also in Chapter 9, in which properties are used to provide a proof of the ability of some Time-encoding to capture all delays in the signal.

3.4.3 Outlook

The next natural step is to automate timeprint-based properties checking, in coordination with system level timing constraints. We believe the method is very efficient in detecting properties that were not identified during design phase, which

makes it particularly interesting to use in exploratory missions. Being consistent and cycle-accurate, enables them to act as reliable and transparent evidence for checking in-field behavior. Because of the methods handiness, it can be extended to provide more efficient tracing of analog signals. Using the time-encoding, we propose here, can enable obtaining unprecedented accuracy in capturing details of analog signals values change over time. The method can be extended to provide systems execution signatures, which can in turn reveal anomalies, capture and report non-authorized intrusions of devices to users and authorities.

3.5 Chapter Summary

This Chapter gave an overview of Timeprints, their generation, and a quick view of how the traced signal can be retrieved form them via reconstruction. The efficient tracing uses formal verification by utilizing system's verified properties to reduce the log, to minimal size, and to later reconstruct the signal from these light weight logs using properties. Noticing that tracing digital binary signals is equivalent to tracing the signal's instances of change. To enable the minimal log-size, we take several steps: 1) the tracing task is divided into consecutive trace-cycles, then 2) the internal clocked passage of time is encoded, i.e. each clock within a specific trace-cycle gets a code, and 3) the change instances of the traced binary signal triggers the aggregation of corresponding time-codes. Then 4) only that aggregation of codes is logged at the end of each trace-cycle. 5) A *Reconstruction* is used to obtain the traced signal details, and/or the reach conclusion about what happened via some behavioral check.

We gave also, all over the chapter, where more details about each part of the contribution can be found. This chapter ends the first part of the thesis, and the next one is the beginning of the second part: the theoretical foundation.

The next Chapter is the first in the series of theoretical foundation chapters, which one by one discusses in depth the aspects of Timeprints. The four aspects are: abstraction, time-encoding, reconstruction and evidence. We start by abstraction, because Timeprints are, as discussed here, are abstractions in the first place.

Chapter 4

Foundation 1: Timeprints as Abstractions

This chapter explains the details of temporal abstraction described briefly in Chapter 3. We start by defining the term *Temporal Abstraction* then in light of this definitions, we present our definition of Timeprints as abstractions.

Abstractions are used everywhere to express, calculate, design and implement. Languages –either gestures, movements, words or even mathematics– are all abstractions. They are not the same thing as what they express, but they are needed for referring to anything in any context. The abstractions we use to express entities, are different from nature's auto-generated abstractions. A tree's reflection on the water is an abstraction. Our vision is an abstraction generated in our eyes and minds by reflections of light from objects. Traces left behind someone wearing muddy boots on the ground are auto-generated abstractions, of their walking action. They hint to the fact that someone wearing muddy shoes or boots have passed. Reality as it happens over time leaves its own traces. A walker on the sand or on the snow will leave a trail which might last longer than the action itself, long after the action is over. Time as well leaves its trace on everything which passes through it. We age, things change, and the universe expands.

Inspired by this, if we want to abstract some behavior that happened over time, why not copying the nature's mechanism of auto-generating trails/abstractions? We can let the electronic signal execution leave a trail, and call that trail *a Timeprint*. This is not what have been the case in the existing literature. There, trails or traces of signals looked more like someone leaving intentionally signs or labels as he/she walks. This made sense as the methods were centered initially around the traces of conceptual models and their descriptions, rather than the traces of the actual execution itself. Hence, when it comes to run-time and automated abstraction of reality, why not using auto-generated labels instead?

In this Chapter, we explain how these key ideas lead to the abstraction-aspects of Timeprints. We also explain how they are (auto-)generated and used in expressing temporal behavior. Lastly, the chapter ends with a short summary.

4.1 Signal Behavior Abstraction

When it comes to defining requirements and specifications in terms of time, a deadline for example is not going to be stated in terms of the number of clock-cycles of the system, but rather in absolute interval (milli- or micro-seconds for example) starting from (or relative to) some event occurrence instance. Sometimes, these intervals are calculated using intrusive software APIs (that use the CPU time) to check the fulfillment of such deadlines. The result is an always approximated calculations. Timing information abstraction gave up –for practical reasons– on accuracy. These reasons can be summarized as: it is not possible to have perfect clocks (with absolutely constant frequency and zero jitters) anyway, let alone synchronized ones between distributed systems. Hence, from pure pragmatic perspective, the huge amounts of data required for higher accuracy are not worth it. Absolute accuracy do not exist anyway. Hence, when systems are seen as non-timed state-machines, only the order of occurrence of events matters. Accuracy that enables checking the correctness of that order is then enough. This manifested itself, as mentioned before in Chapter 2, in Lamport's partial and total order [97].

Temporal logic and timing analysis are usually done using models. In systems where timing has to be measured explicitly, as in transmitters/receivers testing, various synchronization and locking mechanisms are used to let the focus on order be possible again. At the RF front-ends, for example, what matters is storing all the data, leaving it to later conditioning and processing stages to extract the data which was originally sent timely.

Temporal logic is used to describe and analyze models behavior and to check the absence of problems. For this purpose, models are considered as best-possible representations of the systems and environment. The more accurate the model, the computationally harder verifying it becomes. Trying to afford both accuracy and computabity lead to the development of methods like Counter-Example Guided Refinement (CEGAR) [49], where the verification is first applied to the highest level of abstraction. If the system is safe at that level (verification succeeded), no further action is needed. Otherwise, refinements (more accuracy) are needed when counter-examples are found, to verify that the counter examples are real. In [48], the authors extended CEGAR to hybrid systems.

However, not only must care be taken when choosing the higher abstraction level to start with (to still be reflective and able to reveal underlying problems) but also it is not always possible nor straight forward to find/create such models. Another problem with this approach is that it starts from the models, rather than from reality. The models are idealizations. They are perfect at reflecting our designs and specifications, but approximate when it comes to expressing environmental factors and physical manufactured products.

Abstraction in the sense of tracing is an automated measurement or triggered signal, generated by an execution (a change in the traced signal's value) and logged separately. A tracer is an abstractor, which conveys some information about the traced signal. This is different from the traditional view of systems abstractions as design artifacts. These stem from the designer's understanding of the system and environment. The abstractor here is the designer, or the verification engineer.

Despite these differences between traces of models execution and those logged from actual physical system execution, we can still make use of the rich literature in temporal logics and the associated verification methodologies. In the following, we start from existing work and explain the difference to our usage.

A classical problem of tracing, is how to define the proper level of tracing-accuracy, which can suffice to convey relevant information, without incurring too much data for logging, storing and processing. The tracing-accuracy abstraction can be considered on two dimensions: 1) temporal, i.e. how frequent should the traced signal(s) be measured; and 2) spatial, expressing which signals to trace and how many digits/bits should be used to represent the measured value at each sample/measurement.

Spatial Abstraction

By *spatial* we mean choosing which signals to trace; as in reality different signals are traced from different points in space (chip area). Abstraction here means choosing which ones to leave (abstracted away). Inputs and outputs of systems and their sub-modules are usually the choice for black-box tracing.

To illustrate how spatial abstractions are done, consider the following **example**: an AND gate, as in Figure 4.1, with two inputs a and b, and one output o. The two inputs, a and b indicate, respectively, the "readiness" and "need" to override the nominal output. First input a indicates the health of the system's sensors, and the other b indicates the urgent need (based on another analysis not shown here) to override the system's output. To explain criticality, the output o is a critical signal, which leads to

overriding the user's commands by another already fixed algorithm which is highly reliable and proven to lead to a fail safe situation. But for the override to be correct, it assumes both perfect sensors (represented by a) *and* real urgent need (represented by b). If any of the sensors are not healthy and the override still took place, this can lead to a disaster. If the decision on need is wrong, but the sensors were all OK, triggering the fail-safe is not going to be as disastrous. It will however cause delays and other material loss, for example: park a moving car at the first available road side; or land the airplane at the closest safe spot. It means the drive or the flight will not reach their destination; but if there is a justification (e.g. the driver fainted or the plane is hijacked), this is exactly the fail safe mechanism to be applied [1] .

In the figure, input a is actually the output from an *NOR* gate, which has two inverted inputs i_1 and i_2. These two represent the health status of the needed sensors. If all sensors are OK, the signals i_1 and i_2 are set to one; if any input indicates a problem in the sensor, the input is asserted to zero, and, consequently, the *NOR* gate output a is going to be zero if any sensor fails. This makes the override possible when needed, i.e. if b turned to 1. Hence, when both the need exist and the sensors are all OK, the AND gate output o, which represents the override signal, is also going to be one.

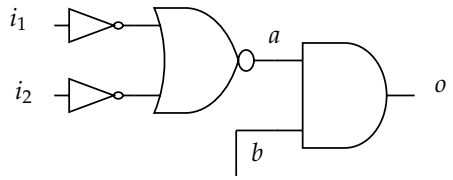


FIGURE 4.1: Circuit for the example illustrating spatial abstraction

We want to log a trace to which we can return in the case of mishap to understand what went wrong. We can simply choose to log all the three signals a, b, o all the time, either with every clock edge, or when (any or some) of the value(s) of a, b or o change(s). The logged trace shall be consulted if something unexpected happen. Now consider the following situation:

1 there was an urgent need for override **and** all the sensors were OK,

2 all inverters are RV verified and operated correctly,

3 but the override did not take place.

[1] Notice that if any of the sensors are not OK and there is an urgent need to override user's control input, other safety mechanisms have to be implemented which do not depend on the sensors. We need here to keep the example simple to explain abstractions related to the described scenario.

We therefor consult our log to check what is the root cause of the problem. Each entry of the log contains three values, one for each of the three signals a, b, o.

A log containing the values: $1, 1, 0$ shall indicate that what went wrong is the AND gate. A log containing the values: $0, 1, 0$ shall indicate that what went wrong is the NOR gate. A log containing the values: $1, 0, 0$ shall indicate that what went wrong is the module which should have driven the b value to 1 to indicate the urgency. A log containing the values: $1, 1, 1$ indicates that the signal o still was asserted, and that the problem which caused the override not to happen lies somewhere after this point; at the module which should have been triggered by the override signal.

We have listed here some of the possible log entries which indicates single points of failure. If many parts failed at the same time, we could have other entries. For example $0, 0, 1$ indicates that all the NOR, AND, module indicating the need and module which should have carried the override; all behaved in the wrong way.

We can reduce the log size under justifiable assumptions. For example, we can choose to log only a and b (abstracting o away), justified by this assumptions: 1) the AND gate is reliable (formally proved and/or run-time verified). For a two input AND gate, one can always deduce the output knowing its two inputs. Another possible abstraction would be to log only a and o (abstracting b away) under the assumption: a is more critical than b (revise the consequences explained above). This stems from that if b was fired it's critical that a is one so that the override happen. If an urgency of override exists and a was one, and still override o was zero, if the AND gate is reliable, then it is clear that the reason is that the b signal is zero. In this particular case, the values of o and a suffice to deduce the value of b.

In conclusion, choosing which signals to log, and which to abstract away depend on the system analysis. In the literature, signal selection for enhanced observability, in the area of post-silicon validations and chip-testing coverage metrics, as in [24, 152], have been a subject of many efforts. Here however, although many concepts from these areas can be borrowed in some cases, the main motivation behind the spacial abstraction in our context is to have smallest log size, while keeping the ability to cover failure root-causes.

From formal verification perspective, some signals' traces are not enough to verify the system's correctness. Verifying a safety property is done over the circuit description itself (model), not any single trace of it. And one way of verifying the model's correctness in all cases, is to explore all its possible finite and infinite traces. This is the trivial way of conducting *model-checking* [50]. But after deployment, if a failure happens, the main concern is find the root cause of that failure. Considering all possible executions is needed before deployment, to ensure that the system is

going to be safe, before it goes to operate in-field. During deployment however, detailed traces should focus more on gathering evidence about what happened; rather than covering all possible cases.

Temporal Abstraction

An accurate trace w.r.t. time results in huge amounts of data. By *temporal abstraction*, we mean reducing the timing accuracy. For example, taking samples over longer periods of time. Or logging only the order in which the different signals change (e.g. Lamport's timestamps). Timeprints, explained next, is another special type of temporal abstractions. To explain traditional temporal abstraction schemes, consider the example in Figure 4.2.

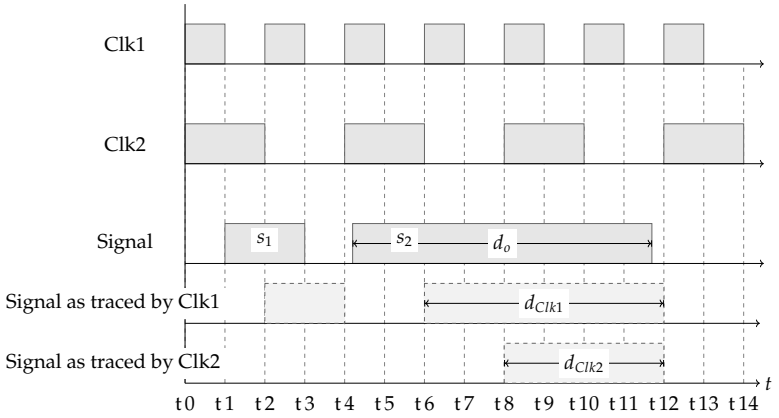


FIGURE 4.2: Measuring Time with Different Resolutions

If the clock is not aligned, at least a clock of double the traced signal frequency, is needed for adequate tracing; this is a natural implication of Nyquist criteria. In Figure 4.2, we assume clock's rising-edge based measurement. For example, s_1 in the figure cannot be detected by $clk2$; because it was high for only half the duration of $clk2$. Even for signals with lower frequency (higher duration) than $clk1$, The accuracy of indicating which time the signal changed can be affected; as seen in the different duration-lengths d_o, d_{clk1}, and d_{clk2}, measured by different tracing clocks.

Deadlines are usually given in the specifications in absolute time. For example, a specification might look like: "If condition c holds, signal a has to go from zero to one, in less than 5 ms, from when signal b went from zero to one". These 10 ms can express the time in which a planning module (which evaluates c) **must react** (raise signal a from zero to one), measured from the instance and input (signal b) has changed; which can be coming from a perception module for example. To trace the satisfaction of such a specification, one needs to count the clock-edge deviation tracing error, and

project that into the internal module requirements. Also the duration on which the levels of the signals remain at zero and one, to ensure they are captured, has to also be specified.

Let's assume that the second *Signal*, s_2 drawn in Figure 4.2, is a value of signal a, which is required to meet the 5 ms deadline. The 5 ms shall be measured from b. We assume that b is also traced with the same clock as a. Both signals are shown in Figure 4.3.

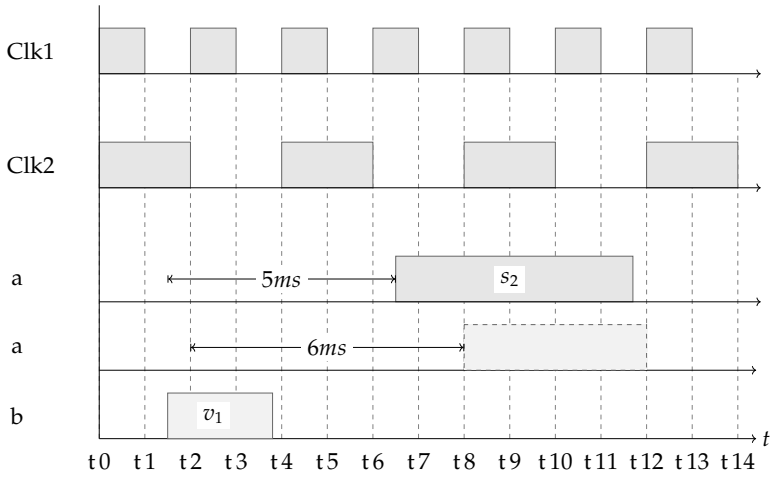


FIGURE 4.3: Measuring Time Between a and b

A natural problem is how to determine the start point of measuring the 5 ms. When the change in b, which indicate the beginning of the 5 ms, happen before the clock-edge, it will not be actually traced except when the clock-edge comes. Hence, we have to count for the worst case. Namely, that b has changed from zero to one right after the rising-edge has passed of the previous clock-cycle. So technically, nearly one clock-cycle could have passed unmeasured. Notice that a's required response can –in the worst case, i.e. largest delay given the measured values– b has just occurred at the clock-edge in which the measurement is taken. The 5 ms duration defining the deadline has to be adjusted to: duration= $5ms - (clock - cycle - period)$. Then, this period must be mapped to a number of points, corresponding to the sampling instances. If we assume the clock-cycle-period of $Clk1 = 2$ ms, then, a duration of 3 ms is left after counting for the worst-case as above. Actually if such a requirement exist, it means the tracing clock has to be small enough to trace it. The last clock-edge, which coincides with the 3 ms interval, is the last chance for the raise of a to appear.

We call the set of clock-edges, which happen within the duration in which a meets the deadline τ. If, in our example, τ happened to end exactly at t_7. In this

example, although the signal s_2 did not miss the deadline, its traced version did. The opposite situation could have also happened. If b's v_1 started earlier, i.e. before t_1, and s_2 started exactly at t_6, here s_2 has actually missed the deadline, but the tracing of both a and b would announce it didn't.

In the light of this example, one may conclude that tracing with the smallest system clock is the best affordable solution. But the problem is the huge amounts of traces this shall incurs. It is common to use wider clocks (lower frequency) to avoid huge logs. And have systems report internally with their local time about the important events; for which it is required to meat certain deadlines for example. On one hand, this is problematic from evidence perspective because of the self-reporting element. On the other hand, wider clocks lead to non-determinism regarding whether a deadline was met or not; as shown by the example. It shall be optimal if we can use the smallest possible clock-period but without incurring huge logs.

To reduce the log size, we inspect how can the time be represented in efficient way. We start step by step in the coming sections.

4.1.1 Representation of Time in Logged Traces

In a signal trace, which is physically logged from a system, the timing information can be represented in several ways. Logging traced signal can be done by logging every sample, which means the n^{th} entry in the log corresponds to the signal value captured at the n^{th} tracing clock edge.

In some cases, traces contain only newly values, un-timed. This is the case for system-model traces, for which functional relations between values need to be checked, and their timing is irrelevant. In these systems usually the whole state of the system is considered, and need to be checked versus not reaching a bad state (safety property checks), or possibility of reaching good state (liveness property). In these cases no timed interaction, with the physical external environment or other systems, is considered. But in systems where this interaction is inherent as in CPS, the temporal aspect becomes unavoidable. Hence, when timing is relevant it is hard to follow this approach, of only recording new values, un-timed, in a trace.

Representing the timing information, with more than just the implicit order of new values' appearance, is done in several ways. These ways would differ if we are talking about a physically created trace from a real execution which took place, or we are talking about theoretical system-model trace of possible execution. On the physical logs and traces, each new value in the trace can be accompanied with

a timestamp. The Timestamps can be either the clock-orders, on which values appeared, obtained by a counter, or by converting the count into absolute time, measured from reset or recording start time. On the system-model traces, considering time in the state and transition descriptions lead to the development of Timed Automata [12], which added the notion of time to the Automata-based system models. This then enabled modeling timed-transitions and checking conditions like meeting deadlines.

Temporally limited accuracy, i.e. with clock and counter or Lamport's timestamps can keep the order, but will miss checking extended temporal properties, that are expressed using the absolute time. When we talk about one logged trace of a specific signal, the order is already kept between the elements of this trace. And when the clock-cycle width is known and there is a log every clock-cycle, then the timing of each trace-entry is defined implicitly, with the clock accuracy. When relation to other physical traces is needed, the timing, either with based on the order and clock value, or an explicitly logged timestamp, will be important.

4.1.2 Abstracting Temporal Accuracy

In some applications, many samples are taken from the signal, to ensure its stability at the measured value. There, usually a small processing unit or micro-controller is responsible for signal conditioning, and for generating then a proper timestamp and attach it to the logged value. In this case, also, the same problem is still there. The logged timestamp expresses one point in time, around which or before which the event happened; but not the exact point.

There is always a range of time values, which can be expressed by a time-interval; within which the event have happened. And reducing the tracing clock period can not entirely eliminate the problem. But as mentioned in 4.1, can only make it better to judge based on clearer defined acceptable terms. For our above example, to accept that we shall judge a raise in signal a coming in 3 ms as late, (although the deadline is at 5 ms) is not easy to accept nor to accomplish. When projecting deadlines to components and sub-modules, (where each probably use different clock) expressing delays in terms of the smallest clock becomes almost unavoidable; otherwise the specifications analysis has to account for different regions, where specs apply and where they do not. Besides relieving the burden from systems specifications perspective, accurate measurements can also provide more insight that can enable detection of hidden problems.

Today, projecting a specified deadline into testing its satisfaction is not yet standardized, and is left to designers and testers. In embedded software, it is a common practice to call a software time measurement function at the two points, then a subtraction process checks whether the lapsed time is less than the deadline. Using software code to do this is inherently error prone and should be taken with a grain of salt. Such method is intrusive (time and processing resources taken to calculate the time), non-transparent (where the code exactly was put and how this relates to the actual time between the 2 specified events) and suffers from all errors arising from calculating time periods, in terms of clocks. Here, even worse, as it depends on calculating the absolute time out of these, instead of the more accurate number of clocks. Taking tracing clocks into consideration as we suggest here is also not standardized.

Real-Time (RT) interaction with external physical environment makes safety critical Cyber Physical Systems (CPS) verification exceptionally hard, because their temporal behavior is specified, with respect to the environment's time and state; rather than the system's internal state. To this end, local internal clocks (based on physical crystal oscillators for example) can serve as a reference, to which the external world time can be measured as perceived by the internal clock. In this work, the system's basic clock is used to obtain efficient temporal abstraction of the traced signals. Notice that this is not necessarily the smallest system clock, because when we do white-box tracing (for open source hardware for example) a signal which change according to certain clock edge is better traced with that clock.

In the next chapter, we present a different abstraction of time. It tries to summarize the details, without fully overlooking them. This means instead of just having longer tracing clock's duration (to make logs light) we log at the end of a long trace cycle, a summary of the details which happened with a very small clock-cycle.

4.2 Timeprints as Temporal Abstraction

Timeprints, introduced in Chapter 3 are temporal abstractions of digital binary signals. But unlike previously discussed temporal abstractions, they do not just abstract accuracy. Rather, they focus on gathering data about the timing, ans summarizing it. Is there are more than one digital signal to be traced, spatial abstractions are applied to decide which signals shall be traced. If one digital signal has more than one bit, it can be abstracted into as one change-signal and only focus on tracing the time in which it changes.

In the coming subsections, we explain the three different levels of Temporal abstractions, undertaken by Timeprints. First, change-instance-triggered abstraction, sacrificing spatial accuracy. Second, trace-cycle-based abstraction, counting the number of changes, sacrificing temporal accuracy. Third, aggregation of time-codes over a trace-cycle, which is also an abstraction, but one which enables directed concretization of temporal accuracy through reconstruction, to amend the accuracy lost by logging only the counted number of changes over the trace-cycle.

As mentioned in Section 4.1, traces of a signal which results from physically tracing it, are automatically generated abstractions of the signal. Instead of classically: using design models, or manually abstracting existing devices, of which traces represent one simulated execution, we let the real in-field execution automatically generate its behavior abstraction. We make this possible by giving the execution signals a way to trigger generating the abstractions. The trigger of this generation is automatically done by measuring the change in the signal; as explained next.

4.2.1 Change-triggered Abstraction

Let's focus first on tracing one digital binary signal. Assuming the smallest system clock *clk* is used for tracing, a full trace of the signal (a sample every clock cycle) would be a sequence of zeros and ones. Each zero or one, expresses the signal value at the clock-cycle corresponding to its order. Repeated zeros and/ones are going to be written several times. i.g. a trace can look like:
0110010010110000100111011010100111101000110....

When tracing more than one bit, it could be more efficient to record only every new value, paired with its time (to avoid repeating writing the same value several times). Value-time pairs appear in a trace like: $(213, 0.001), (17, 0.030), (728, 0.037), (394, 0.207), ...$ Even if one bit is being traced, it is still reasonable to follow this approach if the rate of change is orders of magnitude less than the clock-rate. This is useful when the resulting log size is less than logging one bit per trace cycle. But it suffers from inconsistency of the log-size (dependent on rate of change), and requires extra hardware and memory to track the current time and attached it to the logged entry. When it comes to retrieving particular entry happening at a specified time window (i.e. not just retrieving some n^{th} occurrence), the search operation is cumbersome as it is not easy to spot where the entry would be in the trace.

Tracing accurately only the changes in a binary signal can be very practical; especially in monitoring systems which are already formally verified. Formal verification works on abstract models, which means the values are mainly verified, but not the

timing in which these values take place. That is the reason it makes sense to trace the timing of change. Moreover, if the initial value is known, an original binary signal can be restored from its changes trace.

For non-binary digital signals, we can assume that the traced signal's values are available (known, logged, or can be inferred) by other means. For example, tracing a fixed software code execution (with no interrupts). There, given non-self modifying code, we can trace only the changes in the program counter value, and whether a branch is taken or not. With such a trace, and full knowledge about the software image, one can reconstruct a trace of any execution. This shall be discussed in Chapter 9. In these cases, logging a stream of bits expressing whether a change happened or not, can help in fully reconstructing the abstracted values.

For any traced signal S (either input, output or even any chosen internal signal to be traced), to know the exact value of the signal, at any clock-cycle i, the number of changes which happened before i is enough to know S_i's value v_j. Notice that in general it could be that $j \neq i$, because there can be clock-cycles where the signal does not take new value. We use s_i to describe the value of S at the i^{th} clock-cycle.

To express whether a change happened or not requires a binary signal, which takes the value 1 when a change happens. This signal shall abstract S into the exact timing of when exact different values $v_1, v_2, ..., v_e$, took place, where v_e is the last value of the signal S within some fixed number of clock-cycles in the trace.

Example 2 *A trace of length l clock-cycles of some signal S can be written as:*
$S = s_1, s_2, s_3, ..., s_l$, *where*
$s_1 = v_1$, $s_2 = v_1$, $s_3 = v_2$, $s_4 = v_3$, ... $s_{l-1} = v_e$ *and* $s_l = v_e$

Detection of change between signals can be implemented in hardware. In this set-up, the actual signal trace is abstracted into change instances.

A change-instances abstraction C can be seen as a discrete difference function, over the samples of a signal S, taken every clock-cycle. This expresses the exact time over which the corresponding change has taken place.

Physically, a change can be detected by comparing an old to a new value of the traced signal; for example s_i and s_{i+1}. A difference between digital traced signal's value in one clock-cycle, can be easily compared to latched value from previous clock. If there is a change, the change-instance abstraction function output is one, otherwise it is zero.

$$\forall i, 0 < i \leq l, \quad C_i(S) = \begin{cases} 1, & s_i \neq s_{i-1} \\ 0, & s_i = s_{i-1} \end{cases} \tag{4.1}$$

where s_0 is the initial state of the traced signal, and l is the length of trace in clock-cycles. Notice that C can be written as $c_1, c_2, ..., c_m$, where $c_i = 1$ when there is a change at the i^{th} clock-cycle, and zero otherwise.

It is straight forward to implement compactors of digital signals. For analog signals, which change qualifies as a trigger has to be defined. More about analog signals tracing considerations is mentioned in 9.3.2. Notice also that this abstraction will miss any change happening within less than the clock-cycle. To be able to capture less than that, methods discussed in 8.5 can be applied.

4.2.2 Trace-cycle-based Abstraction: Counting

Temporal accuracy abstraction was explained in 4.1. There, we showed how logging data with different clocks shall affect the accuracy of time measurements. Higher frequency clock provides more accurate traces, but they result in unacceptable logs size. On the other side, wider sampling clock (lower tracing frequency), by which we measure the traced signal value, leads to worse temporal accuracy of measurements.

As a way to balance between these two extremes, we suggest using the higher accuracy of the fastest clock in tracing, but at the same time we do not log the measurement (nor the change) traced every clock-cycle. Instead, we log only a summary at the end of a relatively long [2] tracing period. This period is what we call a *trace-cycle*, and use TC as a short cut of it. Hence, a signal trace S, of length l clock-cycles, can be represented by a trace t of consecutive l/m trace-cycles (TC's); where m is the trace-cycle length.

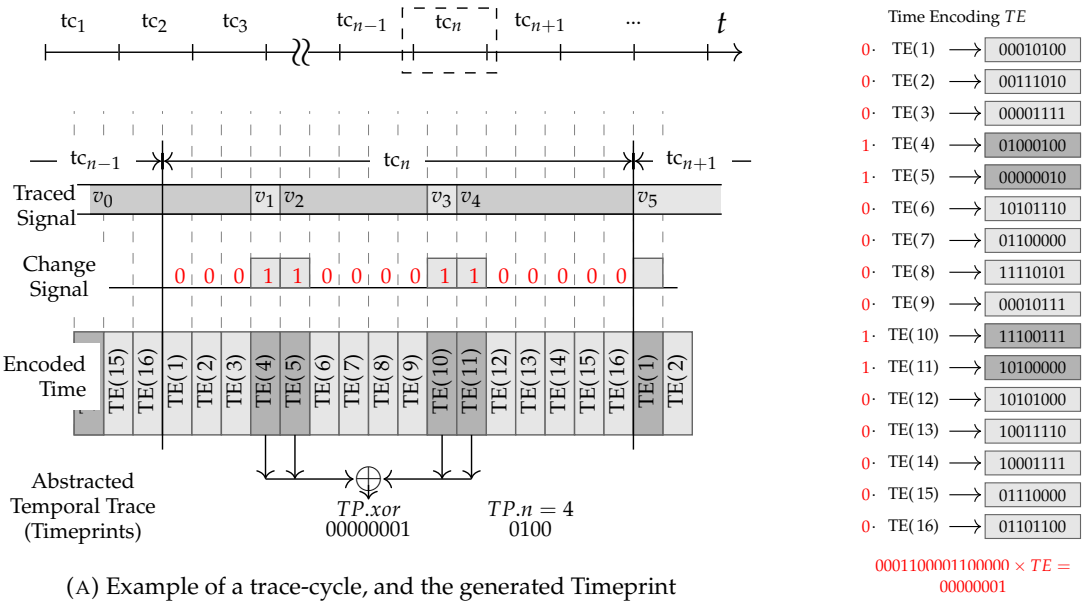
One form of summarizing the temporal info, over a trace-cycle, is to count the changes taking place. Then we log at the end of this period the counted number of changes. Formally, this is a temporal abstraction, which abstracts the changes which happens within a trace-cycle, into a single number, representing how many changes took place within the trace-cycle.

This might seem in the first sight as worse than tracing with a lower period clock-cycle; but this is not true. Because when we trace with the fastest clock, but only count the change, rather than logging it, we are sure not to miss any change which happens within the fastest clock. Notice that there will be ambiguity in the exact moments where the counted changes happened. To reduce this ambiguity, without needing to log a bit every clock-cycle, we introduced *time-encoding*, explained next.

[2] Relative to the clock-cycle period, the tracing cycle period needs to be at least three orders of magnitude, to reduce the log size to acceptable range.

4.2.3 Trace-cycle-based Abstraction: Time-Codes Aggregation

As mentioned before, the number of changes is logged at the end of each *Trace-Cycle*. This is implemented by first dividing the tracing task into trace-cycles, as in the top of Fig. 4.4. Besides counting the changes in every trace-cycle, each clock-cycle within the trace-cycle gets a coded bit-vector distinguishing it, which we call this Time-Encoding TE. Fig. 4.4 shows an example of such encoding at the right.



(A) Example of a trace-cycle, and the generated Timeprint

FIGURE 4.4: Timeprints Generation as Abstraction

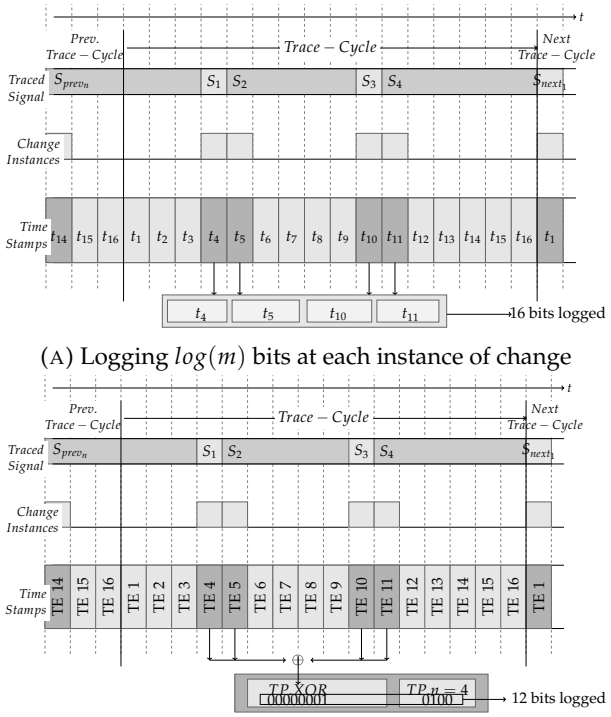
A time-encoding, TE, is usually fixed before system deployment, hence, has to be as much transparent and independent as possible, to raise its ability of capturing unexpected behavioral details. TE can be viewed as an $b \times m$ matrix, where each column is a time-code. Then the Timeprint TP XOR part is generated by multiplying the change vector x by this matrix as: $TP.xor = C(S) \times TE$. This Timeprints generation mechanism, besides its small incurred logs, enables fixing the logging rate, irrespective of how frequent the traced signal change.

At the end of each trace-cycle, an aggregation of the exact timings' codes is logged. As shown in the figure, the aggregation function is simple an exclusive OR (XOR), of the codes' bit-vectors. This aggregation $TP.xor$, together with the number of aggregated codes $TP.n$, are logged at the end of each trace-cycle, constituting the trace-cycle's Timeprint.

This way, Timeprints abstract the temporal behavior in a way that do not cause loss of timing details, in the same way caused by the coarse reporting of time.

At the left of Fig.4.4, a signal time-line, divided into trace-cycles is shown, where one trace-cycle is magnified below. At the instances of value change, for example from v_1 to v_2, a change-signal is set to one, marking occurrences of change, otherwise, it is zero as long as the traced signal value did not change. The time-codes corresponding to when a new value occurred are aggregated into the Timeprint TP.

To retrieve the original traced data from the logged Timeprint, a *Reconstruction* is needed, which shall be explained in detail in Chapter 6. The example shown in Fig. 4.4, shows aggregated time-codes into one logged Timeprint. The first formulation and the associated problems were first mentioned in Chapter 3.



(A) Logging $log(m)$ bits at each instance of change

(B) Logging aggregated timestamps

FIGURE 4.5: Different logging schemes

Timeprints also follow the approach of consistent logging, which logs the same amount of data over time. To clarify this, take the example of a trace-cycle is shown in Figure 4.5a. It contains 16 clock-cycles. A timeprint (TP) is shown in Figure 4.5b, it is an 8-bit vector. In general, when the number of changes is a natural number $n \in \mathbb{N}$, we need to log $n \cdot \log(m)$ bits each trace-cycle. Hence, the amount of logged information would depend linearly on n. This has two major disadvantages:

(1) Since the number of changes in a signal varies from one trace-cycle to another, the amount of logged information also does. This leads to inconsistencies in the logging pattern. Hence, processing the logged information or searching through it

is not straightforward.

(2) If, during a trace-cycle, the number of changes n is too high, it is not possible to log the timings of changes anymore. In a trace-cycle of length m, the maximal number of bits that can be logged is m: One bit is logged each clock-cycle. If we need $\log(m)$ bits for each change, we can record at most $m/\log(m)$ many changes. In Figure 4.5a, we log 16 bits. This is the maximum number of bits. Hence, we cannot detect more than four changes in a signal.

A better way to logging is to summarize the temporal behavior of a signal at the end of each trace-cycle. This keeps the amount of logged bits constant and hence resolves (1) and (2).

The only information logged by our method during a trace-cycle is the timeprint, which is composed of two parts: 1) an aggregation of the time-codes and 2) the counted number of changes. Yet, still this aggregation may create ambiguity: There might be different signals leading to the same timeprint. Finding these different signals is called *reconstruction problem*, and shall be explained in detail in Chapter 6.

4.3 Mapping Temporal Behavior to Timeprints

After defining Timeprints as abstractions, the relation between such logged abstractions and the temporal behavior, (which is usually described in traditional temporal logics) is clarified. One major obstacle is that Timeprints are defined to reflect a summary of exactly one trace-cycle, in one actual trace. By contrast, temporal behavior is defined over all executions of the system, and deadline constraints are described as intervals whose length uses an absolute time domain. This means classical properties cannot be checked using only Timeprints, logged at the end of trace-cycles.

Traditionally, temporal behavior can be described in various ways. Not only because of the variety of existing description methods, languages and logics, but also because this depends highly at the nature of the problem at hand. Focusing on CPS help us narrow this down in two different ways: 1) we can focus on (input and output) signals abstractions and their relation over time and 2) as the hardware is fixed, the software execution is what is left to be monitored. Input/output signals can be seen as continuous streams of digital data, (for analogue signals we'd assume for the scope of this thesis that they can be captured at the point after an Analogue to Digital Converter ADC).

Software execution itself leaves naturally a digital trace: for example: executed instructions and processor status register, or program counter of the execute stage in a pipeline. As these constitute a digital stream as well, we can assume that all what

we have are streams of digital signals. Without our Timeprints, temporal logical relations between streams of digital signals has been described in various ways (refer to [23, 80, 90] for an overview). Here, we pick First-Order Metric Temporal Logic (FO-MTL). In the following, we give a simple example of how a typical FO-MTL formula looks like, when mapped to the Timeprints domain. Our example describe that access to some resource (e.g. input port on a device) must be granted, within $10ms$ (from the moment the access is requested, and only if the requested-access is allowed) to an application which is requesting it.

Consider the following FO-MTL formula:

$$(access\_request \wedge access\_allowed) \implies \diamond_{[0,10ms]} access\_granted \tag{4.2}$$

where the symbol $\diamond_{[0,10ms]}$ denotes that *access_grant* shall be given eventually within $10ms$ from the time in which the premise $(access\_request \wedge access\_allowed)$ holds.

To map such properties to the Timeprints domain we first have to understand that while the MTL formula is meant to hold all the time, the Timeprints domain addresses only whether the property hold or not over trace-cycles for which we have Timeprints at hand. Once this is understood, we apply two steps: 1) Define Timeprints, which includes defining the the change vectors of the 3 signals *access_request*, *access_allowed* and *access_granted* –let's denote them as C_1, C_2, C_3–. These signals are obtained from changes in signals as in section 4.2.1. We leave assume we are interested in whether the property hold within one trace-cycle here for simplicity. Mapping properties over multiple trace-cycles, determining the trace-cycle length and the issue of "some part of the trace signal is happening in one trace-cycle and the rest happening in the next" are explained later in Chapters 6 and 8. Here, let's assume all the three values of the traced signals above are happening at the same trace-cycle. In this simplified set-up, we have 3 Timeprints: TP_1, TP_2 and TP_3. Where for each, $TP_i = TE_i x C_i$. 2) re-write the MTL formula in terms of the new variables. The condition in equation 4.2, can be described over the variables of the change vectors C_1, C_2, C_3. Notice that each change vector can be written as: $C_i = c_{i1}, c_{i2}, ..., c_{im}$ where m indicates the number of clock-cycles within the trace-cycle. The condition in equation 4.2 can be re-written over our one trace-cycle as:

$$(c_{1j} \wedge c_{2k}) \implies \diamond_{[0,10ms]} c_{3l} \text{ where } j \leq k \text{ and } k \leq l \tag{4.3}$$

where C_1 is the *access_allowed*, C_2 is the *access_request* and C_3 is the *access_granted*. Also the instance in which the access request occurred c_{2k} is the k^{th} in the trace-cycle.

Notice that if many changes were happening to any of this signals within the trace-cycle, one can point to the order of the particular signal occurrence by means of cardinality constraints as shall be seen later in chapter 8.

Notice that here the precedence relation between the access request and the *access_allowed* became relevant; this resulted from the fact that the classical way to check FO-MTL is triggered by the moment the implication premise is satisfied and the monitor looks during the indicated period (here 10ms) for the implication to hold to consider the property satisfied.

4.4 Timeprints Generating Function

We have so far described a concrete way of generating Timeprints. However, Timeprints are in essence a more general concept. They can be viewed as a transformation of consecutive traced signal chunks into a sequence of Timeprints. This can be expressed by a mapping or generating function: $\mathscr{F}_{TP}^{G}$, which takes itself a map (from time domain to the domain of the Timeprints) expressing a signal chunk (e.g. the change signal over a trace-cycle described above) denoted by $\mathscr{C}_S$, and produces a Timeprint TP as a projection. This can be expressed as:

$$TP = \mathscr{F}_{TP}^{G}(\mathscr{C}_S) \tag{4.4}$$

As mentioned before Timeprints abstract out some details, as it reaches efficiency by the lossy XOR aggregation. This means that in many cases Timeprints of different signals shall be projected to the same TP. The main idea is to make what gets abstracted out coincide (as much as possible) with what we actually know and can recover about the signal. This should enable us to differentiate between the signals which are mapped to the same TP. This way, logging the Timeprints will not incur much data, and only what we do not know can remain there un-abstracted. Ideally, Timeprints would abstract away *all* what we are sure about (e.g. checked with Run-Time Verification) and keep the rest. However, this implies that huge RV checks are implemented, which would result in huge area overhead. Another problem is the complexity of generating the codes which can check every other possible property is prohibitive. For this reason, completely design-independent time-encoding abstractions, like those described next in [112], and in the next chapter are used.

4.5 Chapter Summary

Abstraction is a key corner stone in constructing Timeprints. In this chapter, signal abstraction in general was first discussed, before presenting the three levels of abstraction used in generating Timeprints. These levels are namely: signal change abstraction, counting changes over a fixed period (trace-cycle), and aggregating the exact instances of change over this fixed period by XORing the corresponding time-codes. These three levels result in a train of Timeprints for a traced signals, where each Timeprint is an abstraction of the temporal behavior over the trace-cycle. A generalization of this abstraction was discussed briefly. Timeprints can be viewed as projections of the traced signal in another space, where signals can be transformed into there via a Timeprints transform. The transform admits an abstraction function (Timeprints generation) to project traces into the Timeprints domain, and a concretization function (Timeprint Reconstruction) as the inverse transform to retrieve the original signal. The next two chapters address these two functionalities, the generation and reconstruction, consecutively.

Chapter 5

Foundation 2: Encoding Time

Time encoding is the process of marking each time-instance, clock-cycle edge for example, with a unique code, that shall be triggered if a change happened in this time-instance. Choosing which and how time-instances are encoded is the topic of this Chapter. Timeprints are continuous periodic logging of summaries of temporal behavior. One Timeprint is generated every trace-cycle, and the whole tracing is happening over consecutive back-to-back trace-cycles.

The aggregation of time-codes, generated by algorithms presented in this chapter, is used to express the temporal behavior over the trace-cycle. As explained in Chapter 4, the tracing change-abstraction is used to generate a change-signal. When the value of this change-signal is one, the time-code corresponding to it is triggered.

Triggering the code means involving it in the process of generating the Timeprint, which is logged at end of the trace-cycle including this instance. An example of how an encoding might look like is given at the right of Figure 5.1. The encoding in general is going to lie between two extreme ends: 1) the smallest unique codes are the indexes of time instances, (and will be of bit-width $logm$) and 2) the largest codes, which are hot-bit encoding (and will be of width m). We explain these

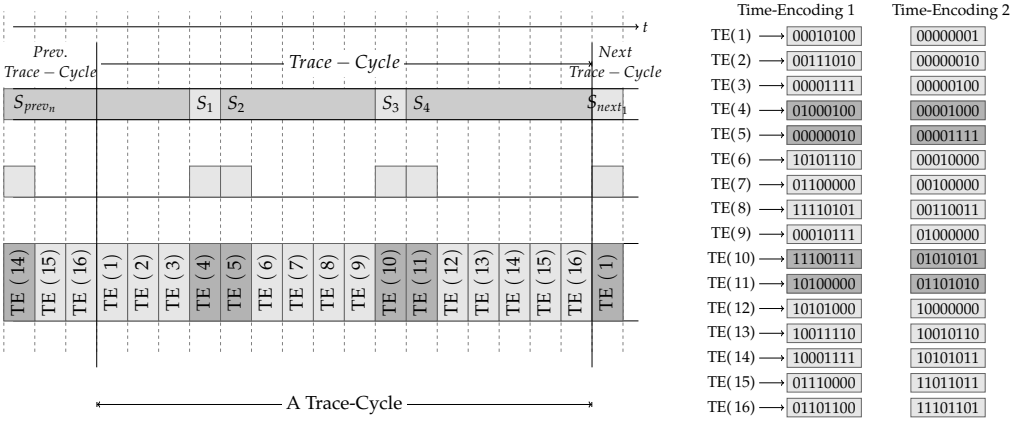


FIGURE 5.1: A trace-cycle, with Time-Encoding Example

two extreme cases in the following section, as they are very important in the evaluation of any encoding that might be used.

After that, in this chapter, the basic concept we use for encoding time is explained; the so called *Linear Independence of Degree N*, $LI - N$. This concept is the major enabler of an efficient yet design-independent tracing. We then explain different time-encoding generation algorithms; which have been developed and used in case-studies. Another variant of time-encoding is the properties dependent encoding. Although it might seem contradicting to the design-independence requirement, considering some properties in choosing the encoding can have benefits; when used carefully.

The chapter presents a brief complexity analysis of the encoding generation. Although the cost is very high, the encoding is required to be done, in most of the cases, only once before deployment. Many of the completely independent encoding algorithms discussed here, need not to be run again; and the resulting codes have already been used in many applications without any need to recalculate them. The Chapter ends with defining the notion "encoding-generation functions", which enables hardware efficient implementations of the time-codes.

5.1 Encoding Time Passage

We use in today's systems development two separate notions of time: clocks, which are mostly used for hardware, and absolute intervals between timestamps, usually reported by software. While clocks are used by digital systems designers to derive sequential behavior, absolute time is the form used by high level requirements and specifications. To keep track of the absolute time after deployment, usually clocks lapsing between two events are counted. To determine whether a specified absolute time of deadline was met, if the number of clock-cycles lapsed –between two events– is less than the deadline, it is met; otherwise, the deadline is missed. This results in an approximation of the absolute relative time.

However, this approach is limited, on one hand by the number of counters and their counting limits. on the other hand by the clock-cycle width as accuracy limit. When deadlines are large enough compared to the clock-cycle width, considering the deadline as *missed*, because it wasn't met a clock-cycle before the actual absolute time is acceptable. However, when tracing temporal behavior in general, every single clock-cycle (not just those where counters are set to test) might indicate a behavioral property that can be relevant. For example: missing deadlines, as a defined

property, would be captured by Run-time Verification checks; while smaller delays that might indicate a security threat [74], can skip such check.

Timeprints aim at cycle-accurate tracing, without generating un-acceptable amount of traces. It does this by utilizing the layers of abstraction described in Chapter 4. The last abstraction-layer utilized time-encoding; which we explain here in this chapter. The time encoding TE is done for one trace-cycle, and then reused for all trace-cycles. This is why we name each interval of time over which we summarize the temporal behavior as *trace-cycle*; because the time-codes repeat after a trace-cycle ends. We fix the XOR as a generation function; to discuss possible time-encoding strategies. Even for this one generating-function, encoding time instances within a trace-cycle, can be done in endless ways. But we shall focus here on the finite set of encoding-schemes which results in codes of sizes $log(m) \leq b \leq m$. The codes first goal is to present an efficient time representation. The codes minimum requirement is that each instance's code is different than the other; which fulfilled by the minimum $b = log(m)$.

We start by analyzing the two extreme encoding strategies (Given XOR as aggregation function): 1) the smallest codes which are still able to fully recover every temporal behavior, and 2) the absolute minimum $b = log(m)$, yet unique codes. These two strategies are used as a reference to analyze the efficiency of other encoding schemes.

The smallest codes which can recover all temporal behavior patterns are hot-encoding. Every instance's m-bits long code is one at the instance, and zero otherwise. The result of the XOR function is then straight forward; as the instances where changes happen results in a single bit being one in the resulting Timeprint; and this one bit is never going to be changes by any previous or next time-instance code. This is equivalent to using m bits for each trace-cycle, and the bit is set to one when there is a change and zero otherwise. This is just using the signal-change vector as it is. The bit-rate required to log such encoding is one bit per second. In systems with Giga-Herz clocks, this means for one traced signal, one second results in one Gig-bits log.

On the other extreme, the smallest time codes which can be used, $b = log_2(m)$, can be simply mapped to the indexes of the time-instances. Within a trace-cycle of length m clock-cycles, we have m time instances. We can encode these instances, each by an index, i.e. $1, 2, 3, ..., m$. The index 0 is not used, because it is transparent to the aggregation function XOR. i.e. it will not change TA when triggered in the Timeprint calculation.

5.2 Defining Degrees of Linear Independence

Linear independence of vectors, which have the same bit-width, is well-known in mathematics. Here we define another type of linear-independence. The focus here is concerned with a specific fixed set vectors, and our linear independence is required to keep sub-set abstractions of these vectors distinct. We call this here: *linear-independence of degree* N, where $N/2$ is the number of sub-set members (vectors), which is required to distinguish between two different subsets, each of $N/2$ distinctive members (vectors). Notice that the whole $N/2$ vectors are distinct, but on the level of individual vectors, there could be repetitions; but not all of them.

5.3 Time-codes Generation Problem

The required time encoding, TE, is an ordered finite set of b-wide distinctive bitvectors, of m elements. It is ordered because each element corresponds to a predefined clock-cycle within the trace-cycle. The distinctiveness of the vectors is essential; hence the minimum $b = log(m)$. We denote the set of such distinctive time-codes by TE. An element of this set can be accessed by an index i, as $TE(i)$, where $TE(i)$ is the i^{th} bitvector, and it represents the code corresponding to the i^{th} clock-cycle inside the trace-cycle.

Changes happening to the traced signal, over *trace-cycles* of *length* m, with $m \in \mathbb{N}$, resembles the change-abstraction we described in Chapter 4. Although it describes the change in the traced-signal, the change is itself also a signal. We consider the *change-signal* as a map, $C : [1..m] \rightarrow \{0, 1\}$, where $C(i) = 1$ when a change takes place in the i-th clock-cycle, and $C(i) = 0$ otherwise. We use $C_x(i)$ to denote the i^{th} clock-cycle change status, of a specific fixed change vector (change-signal) C_x. We enumerate all change-signals, i.e. from C_1 to C_{2^m}, representing all possible changes, which can happen in m-long trace-cycle. We denote the set containing all such change-signals by ς_m; hence $\forall i, 0 < i < 2^m, C_i \in \varsigma_m$.

Within one trace-cycle: A logged timeprint TP, is the returned log entry (TA, k), where TA is generated by the time aggregation function TA_f, where $TA_f(C) = \sum_{i:C(i)=1} TE(i)$, and k is generated by the counting function $k_f(C) = |\{i \mid C(i) = 1\}|$, representing the number of changes in the signal.

The first variant of the time encoding generation problem aims at generating a time-encoding set TE, where its codes can enable complete retrieval of any change signal C_i. This variant can be described formally as follows.

Problem: Time-Codes Generation 1 (TCG1)
Input: TA_f,k_f, trace-cycle length m, bit width $b \in N$ and $log[m] < b < m$
Task: Find TE, such that: $\forall k, 0 < k < m, \forall i, j, 0 < i, j < 2^m \wedge j \neq i$
it is always the case:
$k_f(C_i) = k_f(C_j) \implies TA_f(C_j) \neq TA_f(C_i)$.

This version of the time-encoding generation problem, seeks finding a perfect encoding. It is perfect in the sense it enables recovering any change-signal C_i perfectly for all k, just from the logged Timeprint $TP = (TA, k)$. In other words, when this encoding is used, the reconstruction as mentioned in 3.2.2 shall always result in a unique solution. However, this is not only very hard problem to solve, but also the resulting time-encoding is expected to require a relatively large b, so that a time-encoding TE can be found. Finding the smallest b, which makes TCG1 solvable, for some k, fulfilling linear independence of degree $N = 2k$, is a very interesting, but also extremely hard problem. It is still easier to find whether there exist such a TE of fixed b. A fixed bit width of the time-encoding is also required to be fixed, to generate a fixed width Timeprint. It is also desirable to fix it from practical hardware realization perspective. And the limitation on b-size can be partially derived by area and implementation considerations.

As we actually want to use as small b as possible, to make the tracing more efficient, we fix b as input and loosen the requirement on all k to fulfill the condition $k_f(C_i) = k_f(C_j) \implies TA_f(C_j) \neq TA_f(C_i)$. We call this condition, namely:

$$k_f(C_i) = k_f(C_j) \implies TA_f(C_j) \neq TA_f(C_i) \tag{5.1}$$

the *fixed-cardinality distinct-reconstruction* condition. In words, this condition means: when the number of changes k_f is the same for two different change-signal vectors $C_i \neq C_j$, their resulting time-codes aggregation TA have to be distinct. This is the necessary condition to obtaining a time-encoding which fulfills $LI - (2k)$.

Another variant of the time-codes generation is TCG2 below. It expresses limiting the satisfaction of *distinct-reconstruction* condition to $k \leq 2$; or –in other words– for fulfilling LI-4, as described in 5.2.

Problem: Time-Codes Generation 2 (TCG2)

Input: TA_f,k_f, trace-cycle length m, bit width $b \in N$ and $log[m] < b < m$

Task: Find TE, such that: $\forall k, k \leq 2, \forall i, j, 0 < i, j < 2^m \wedge j \neq i$ it is always the case: $k_f(C_i) = k_f(C_j) \implies TA_f(C_j) \neq TA_f(C_i)$.

Another way to express LI-4, is to consider the subset size as a property, which is equivalent to m, the trace-cycle length, and then ask the generation to result in unique reconstruction, for the vectors/subsets which fulfills the property.

Problem: Time-Codes Generation 3 (TCG3)

Input: A time-encoding set TE, trace-cycle length m, bit width $b \in N$, $log[m] < b < m$, and a property P.

Task: Find $TE' \subseteq TE$, such that: $\forall$ signals $C_i \in \sigma_m$ where $C_i \models P$, $\forall C_{j(j \neq i)} \models P$, $TP(C_j) \neq TP(C_i)$, where $TP(C_k) = (TA_f(C_k), k_f(C_k))$.

Where a property P is a temporal property defined over C. For example, let's express the precondition for LI-4, which is to have two changes in the change-signal vector. We can name this property P_2. And this can be expressed as $C \models P_2 \equiv k_f(C) = 2$. Notice that TCG2 is equivalent TCG3 with $P = P_2$.

Another variant of the TCG problem is the one that is limited in the choices of the possible time-codes to a predefined TE. This variant simplifies the property-based generation of time-codes. It starts from an existing encoding TE_1, which satisfy a simple property[1], for example LI-4. Then, from TE_1, we remove time-codes which do not satisfy distinct-reconstructions 5.1, given the property we need. For example, if we want a distinct reconstruction for every $C \models P$, a TE which fulfills LI-4, can only guarantee a distinct reconstruction when $k_f(C) \leq 2$. In cases where $C \models P \wedge k_f(C) > 2$, the reconstruction-set might contain more than one solution. This variant can then scan all these cases, and remove time-codes which do not fulfill the condition.

[1]Simplicity here is relative, but we mean a one which is easy to generate a TE which fulfills it.

Problem: Time-Codes Generation 4 (TCG4)

Input: trace-cycle length m, bit width $b \in N, log[m] < b < m$, property P and input TE_{in}.

Task: Find $TE' \subseteq TE$, such that: $\forall$ signals $S_i \in \sigma_m$ where $S_i \models P$, $\forall S_{j(j \neq i)} \models P$,$TP(S_j) \neq TP(S_i)$, where $TP(S_k) = \sum_{i:S_k(i)=1} TE(i)$.

Choosing to solve which variant of these problems to generate a time-encoding depends on the application, and the available encoding/s; if any. For example, it is natural to generate time-encoding sets which fulfills our linear independence at first. We start by $LI-4$ and have sets which satisfy it. Later on, when these sets are available, we can choose to solve TCG3 or TCG4 to obtain more specialized subsets. In the next section, many algorithms are presented, which solve these variants.

Among the encoding algorithms, we first present those which results in a fixed encoding, which can be directly reused –as is– and hard-coded in a tracer as it is. Then, we describe using properties to direct the encoding generation.

5.4 Fixed Encoding Algorithms

Time encoding can be done fully a priori, i.e. running a fixed encoding algorithm, which results in codes that are going to be hard coded and used as is. They can be also generated dynamically using a pseudo-random-like generating circuit. Here we discuss the fixed algorithms.

In this section, we introduce different practical approaches, which we used to tackle the time encoding/ or time codes generation problem. We sometimes call the generated time-codes: timestamps; when clear from the context.

In the next subsections, five different generation methods are explained using:

1) an SMT solver: we describe the linear independence of degree 4 (LI-4) to the SMT solver and ask for a set of m codes of width b,

2) random generation: starting from a seed, each random integer generated is checked for LI-4 and the required encoding width is then trimmed,

3) incremental generation: similar to random generation, but starting from 1, and incrementing by one each time for a new choice that is checked then for LI4; the result is minimal in terms of vectors' size (time-stamped codes) satisfying a set of conditions,

4) greedy algorithm: here an algorithm is presented for obtaining time-codes set of fixed width b bits, satisfying LI-4, here the full length of these are obtained irrespective of m, and

5) a composed properties-Based Generation, that takes a set of time-codes as input, and produces a subset of it that fulfills certain property.

For each of these, after describing the algorithm, we present also how the properties can be used in within or at the to of it, for properties-aware time-encoding process.

5.4.1 SMT-based Time-Codes Generation

To describe the problem of TSG using an SMT solver, we used bit vector theory and array theory to describe the array of encoded timestamps. LI-4, is encoded as follows: each aggregated 2 entries corresponding to 2 different time-codes, would result in a different aggregation than that of any other 2 different array entries.

As an example of how an SMT solver can be used to generate the time-stamps, the details of the generation for $N = 2$ is illustrated in this section. The exact same criteria can be applied to higher N. Z3 [124] was used to apply the conditions:

For $N \leqslant 2$ (and using XOR gates to merge the time-stamps), the condition (besides the time-stamps' uniqueness) would be:

$$\forall i, j, k, l, [TS_i \oplus TS_j \neq TS_k \oplus TS_l] \tag{5.2}$$

$$\begin{aligned} , where \ \ & (0 < i, j, k, l \leqslant M) \wedge i \neq j \wedge k \neq l \\ & \wedge (i = k \Rightarrow j \neq l) \\ & \wedge (j = l \Rightarrow i \neq k) \\ & \wedge (i = l \Rightarrow j \neq k) \\ & \wedge (j = k \Rightarrow i \neq l) \end{aligned}$$

Similar conditions can be derived for higher N.

The resultant SMT instance is:

$$
\begin{aligned}
&(exists\ \ ((ts\_var\ \ (Array\ \ (\_BitVec3)(\_BitVec6))))\\
&(forall\ \ ((k\ \ (\_BitVec3))\\
&(l\ \ (\_BitVec3))(m\ \ (\_BitVec3))(n\ \ ({}_{B}itVec3)))\\
&(let((A1(and\quad (not(=k\quad l))\quad (not(=n\quad m))\\
&\qquad\qquad (=>(=k\quad m)(not(=l\quad n)))\\
&\qquad\qquad (=>(=l\quad n)(not(=k\quad m)))\\
&\qquad\qquad (=>(=k\quad n)(not(=l\quad m)))\\
&\qquad\qquad (=>(=l\quad m)(not(=k\quad n)))))\\
&(A2(not(=\quad (bvxor\\
&(select\ \ ts\_var\ \ k)(select\ \ ts\_var\ \ l))\\
&\qquad\qquad (bvxor\\
&(select\ \ ts\_var\ \ m)(select\quad ts\_var\quad n))))))\\
&(=>A1\quad A2))))
\end{aligned}
\tag{5.3}
$$

which reads as: first, we assume the time-stamps are contained in an array called *ts_var*, representing a variable array which the SMT solver tries to find a solution for. In this example, we generate time-stamps of width 6 (i.e. array elements are 6 bits wide bitvectors). We generate 8 time-stamps for a trace-cycle of length 8. Hence, this array has an index of length 3 to address it's elements. The statement $A1$ expresses uniqueness of the pair of indexes of each pair of time-stamps. Namely, for every two different indexes of time-stamps to be XORed (k,l), k does not equal l and to compare the result to the result of any other pair of time-stamps of indexes (m,n), where also $m \neq n$, if $k = m$, this implies that l must be$\neq n$ to make (m,n) a different pair, and similarly goes all the other implications to ensure the uniqueness of pairs of time-stamps. When this uniqueness ($A1$) is satisfied, this implies $A2$, which is that the two results of XORing those two pairs of time-stamps (indexed by (k,l) and (m,n)) are different (not equal, in the SMT formula 5.3). This implication should hold for all k,l,m,n and we assert that there is a time-stamps array *ts_var* that fulfils this condition. A solution that the SMT solver finds for this formula gives a list of 8 time-stamps that are guaranteed to give different timeprints (here results of XOR's), for any 2 different time instances.

An alternative encoding of the LI-4 condition, would be to encode all the XOR results into an array of distinct elements.

Unfortunately, this method does not scale. It becomes very expensive to use

for more than 16 clock-cycles long (array size). While it takes about 10 seconds for trace length of 16 clock-cycles, it takes around 10 hours for 32 clock-cycles. All those measurements are taken on a machine with ®Intel Core ™i7 CPU@ 2.67GHz with 8 GB memory.

5.4.2 Random-based Time-Codes Generation

Random number generators can be used to generate the time-stamps faster. Each newly generated time-stamp is checked to be fulfilling the condition in equation 5.2. If this is satisfied, the results of XORing the new time-stamp with all previously existing time-stamps is added into a List *XORList*, to check the next randomly generated time-stamp against, and this goes on. The generation is illustrated in Algorithm 1.

Algorithm 1: Random Time-stamps Generation Algorithm

```
   Data: initialize   random − seed
   Data: XORList is empty
1  TS_0 = rand()
2  for i in 1 ⟶ M − 1 do
3      TS_i = rand()
4      while IsThereCollision(TS_i) do
5          TS_i = rand()
       /* where IsThereCollision(TS_i) is shown below                    */
6  IsThereCollision(TS_i) {
7  for j in 0 ⟶ i do
8      if IsRepeated( TS_i ⊕ TS_j ) then
           /* where IsRepeated checks whether TS_i ⊕ TS_j has been obtained
              before in the XORList, and TempXORList                     */
9          BackTrack(Reset TempXORList)
10         return True
11     else
12         AddTo_TempXORList( TS_i ⊕ TS_j)
13 Confirm Adding TempXORList to XORList
14 return false
15 }
```

Notice that in line 9, backtracking is needed to the last ensured *XORList* content, when a collision is detected; not to add a non-actually-existing XOR results, from a time-stamp that has become rejected after the collision detection.

This method is much faster than using an SMT solver and the minimal time-stamps generation, mentioned in the next subsection. Time-stamps for trace-cycle's lengths of thousands of clock-cycles can be generated in seconds or few minutes at most on a machine with Intel Core i7 CPU@ 2.67GHz with 8 GiB memory. However, this method does not properly detect if there are no possible solutions to the given

constraints; the designer should thus be sure that the method should eventually terminate.

5.4.3 Incremental Time-stamps Generation

This method is very similar to the random generation, but instead of randomly generating the time-stamp, before checking them, the latest time-stamp candidate is constantly incremented (by one). Afterwards, the new time-stamp candidate is checked whether it fulfills the conditions or not (in which case the time-stamp is incremented and checked again). This method takes longer time than the random generation but remains faster than the SMT solver and can create time-stamps for trace-cycles of a thousand clock-cycles in less than a day on the same machine mentioned before.

Although this method seems to be providing the minimal size of time-stamps, it is still possible to provide the same size with random generation because we know from the number of possible permutations how many bits are needed to present them. However, non standard size random generators have to be manually developed. So this last method turned out to be the preferred solution for custom bitvector sizes that are not available in the standard C data-types.

After the above-mentioned generation, the time-stamps are utilized to mark each clock-cycle within the trace-cycle. By the time a a given signal is toggled, the corresponding time-stamp is XORed into the timeprint, and logged at the end of the trace-cycle. At a host computer that this logged timeprint is transmitted to, the exact instances of change, which triggered the corresponding time-stamps into the XOR-aggregate described earlier, then need to be recovered.

5.4.4 Greedy Algorithm

This algorithm is similar to the incremental algorithm. It starts from scratch and iterates over all possible time-codes in increasing order (i.e. treating them as integer values). Then, it greedly adds a new code to the set of selected time-codes, if doing so does not violate the property under consideration (e.g. LI-4). Due to some optimizations such as look-ahead elimination of time-codes that are guaranteed to violate the property, this algorithm is much faster than the incremental one. It also generates a maximum m time-codes that could be generated of width b, satisfying the LI-4 property. The algorithm depends on binary decision diagrams and implements a near to minimum length time-codes.

5.4.5 Algorithms Summary

In conclusion, the space of which reasonable fixed length encoding of time over a period of length m shall lie is between two ends. On one end, the hot-encoding, where for each instance, the bit corresponding to it. This is an encoding of width m. On the other end, lies the smaller size of a time-encoding, where simply the index of the time instance is used. This is an encoding of width $log(m)$. This encoding can be generated simply by increments of a seed.

The algorithms described in this chapter results in much more concise Encodings. The resulting encodings widths lie between m and $log(m)$. Theoretically, one can encode with more and/or less bit-widths, but with less bits than $log(m)$ a collision is known to happen, so unless there is a special case which justifies this, it is not advised. Using more than m bits does not have added value, first because we are already looking at compressing the traced signal, and second we are focused on the signal that shall be logged at the logical level. More than the signal length in the encoding can help later in physical transmission within noisy environment, but not when we are talking about the level of logical values compression.

5.5 Properties and Time Encoding

After obtaining a set of Timestamps by linear, incremental or greedy algorithm, a filtration of the results by removing those who do not produce a unique timeprint is possible. The resultant timestamps set would be resilient not only to this property it was filtered based on but also might perform better when the signal satisfies other related properties.

Properties Coverage of Time-Encoding Given a Timestamps encoding A_{TS} the temporal properties coverage can be classified using two different criteria. First from the perspective of reconstructions uniqueness: this means the effect of that property on the reconstruction set size. The second criterion is the evidence. This is going to be discussed in detail later in Chapter 7. Here, a brief overview is given from the perspective on reconstruction uniqueness; as it is going to be used afterwards in time-codes generation algorithms assessment.

5.5.1 From the perspective of reconstruction uniqueness

A "Distinctive-Property": is the property that whenever it holds, each trace fulfilling the it has a unique timeprint than all other traces fulfilling the same property. The

distinctive properties are defined by the encoding itself, and can be directly associated with the encoding kernel.

A "Strong-Property": is the property that whenever it holds, the reconstructed traces are bounded by, a defined and acceptable, upper limit on the number of possible reconstructions for a timeprint; for all possible Timeprints. The acceptable limit depends on the application, but generally, in total, at least $1/3$ of the search space of timeprints-reconstructions has to be pruned by it. These are the properties are highly encouraged to be used in run-time monitoring.

A "Weak-Property": is the property which is not strong, but might still be useful to apply for further refinement, after pruning the search space by a strong property.

5.5.2 From the perspective of evidence

Assistant-Properties: (also called assumed properties) are properties which are used to help pruning reconstruction space. An assumed property can be classified into strong or weak properties according to definitions above.

Objective properties are those required to be checked using some Timeprints trail, i.e. the ones which we may require evidence for. From this perspective of evidence, for any given encoding TE, all properties can be classified as follows: 1) Fully-defined Properties by the encoding: those are the properties for which any timeprint is evident in proving or refuting them. 2) Partially-defined Properties: are those properties that can sometimes be proved or refuted by certain encoding (sometimes means here that for some traces baring this property, their timeprints will be evident, while for other traces baring the same property, timeprints are not). 3) Uncovered properties: these properties cannot be proved nor refuted by certain time encoding for any possible trace.

Later, in Chapter 7, the ability of a set of time-codes to cover a property is discussed.

5.6 Time-Codes Generation Hardness Assessment

To assess the efficiency of the algorithms presented, we have first to define a criteria to evaluate the quality of an ordered set of time-codes. One parameter that can be considered a measure, for example, is the bit width b of the time-stamp. The smaller b is, the less logs are going to be incurred, and hence the better a time-stamp encoding is. When a system designer wants to add timeprints based tracing to their system, the first criteria to define is the length of a trace-cycle m (or at least

$m_{min} \leq m \leq m_{max}$), which represents the target time-codes set size. Longer trace-cycles result in less logging effort –as one fixed width timeprint is logged at the end of each trace-cycle–. However, in general, longer trace-cycles means harder (bigger) reconstruction problem size and may require bigger b to make the reconstruction process reasonable for the expected number of changes that could happen within a trace-cycle. A typical range of acceptable trace-cycle length is in the range of hundreds to thousands. If we can find better trace reconstruction algorithms than the one we have now, [114], we can further increase m.

As in the formulation of the algorithms that encode clock-cycles in a trace-cycle, the target is usually to find a maximum trace-cycle length for a fixed b for efficient logging.

When using a set of time-codes, we can assess its performance based on a number of measures. For example, the number of collisions they produce when reconstructing both generic signals; and reconstructing signals related to the properties similar to those they were generated to accommodate. The measures we shall cover here are:

- The run time of the time encoding generation algorithm. Although the algorithm is usually run once, and the result is hard encoded in the hardware. As the problem is very hard, is still important that the run time is not prohibitive.

- How good the generated encoding is able to distinguish between different signals. A perfect encoding (one-bit hot encoding) would be able to distinguish between all S. But this would lead to $b = m$, which destroys the basic idea of *timeprints: having compressed logs*. The quality of the generated encoding is measured by the number of collisions in the reconstruction made using it.

- How long the time-codes generated by an algorithm could be. Some algorithms can generated a maximum encoding (maximum m) for a given b; while others are limited by a given b and m.

- How a specific encoding affects the reconstruction time.

5.6.1 Algorithms Run-time

In order to compare algorithms' run times (in Table 5.1), we tried all the algorithms with $m = 1024$ and b given as 32 when it is passed as input to the algorithm, and when it is indicated as an output, the value is the one reached by reaching the required 1024 time-codes. Since the comparison is not fully possible due to how the

Algorithm	Alg.2	b	m_{max}	Run-Time	new m
Inc-Index	-	output	input	~ 0	-
Random	-	input	input	~ 0	-
SMT-LI4	-	input	input(limiting)	timeout	-
Inc-LI4	-	output	input(limiting)	6h51m6.037s	-
Ran-LI4	-	input	input	59m24.473s	-
Greedy-LI4	-	input	output	17m53.622s	-
LI4-to-LI6 $m_{in} = 1024$	Inc-LI4	input	output	20m51.181s	79
	Ran-LI4	input	output	19h33m53.949s	214
	Greedy-LI4	input	output	9m5.284s	71

TABLE 5.1: Comparison between different Time Encoding Algorithms, generating 1024 time-codes, or for an input set of size 1024 (in LI4-to-LI6)

different algorithms work, we can still give here a qualitative comparison of the run-time, and whether they have the b, m parameters as input or output.

In Table 5.1, first rows gives six direct algorithms run-times. **Inc-index** in the first row is just using the index i of a time-code $TE(i)$ as the code itself. These codes are not for any practical usage, but they are meant to act as a reference to judge properties-usage over a generated set of time-codes versus another. **Random** in the second row is a trivial generated codes, without any checks. It is also used a reference to compare other methods, or properties-based methods to. **SMT-LI4** is the SMT based generation of a set of time-codes that has linear independence of degree 4; which was described in section 4.1. **Inc-LI4** is the incremental generation algorithm described in section 4.3. **Random-LI4** is the algorithm described in section 4.2. **Greedy-LI4** is the algorithm described in section 4.4. The part labeled by **LI4-to-LI6** shows the run time for generating a set of linear independent time-codes of degree 6 from a set that is already generated by any of the three LI4 methods (Inc-LI4, Random-LI4, and Greedy-LI4). For these sets we have a new length m of the newly generated time-codes set. The generated sets are of smaller size because codes that do not fulfill the LI6 condition are removed. Notice that the surviving codes from the Random-LI4 are much more than those surviving from other algorithms; but they also took much longer time to be generated.

5.6.2 Encoding with Properties

The tables below shows the number of reduced time-codes in the case of applying some properties; for example the property that n consecutive changes happened; we denote them by P3, P4, ... Pn. The reduced time-codes (out of 1000) applying

Pn are shown in table (a). Table (b)[2] shows the number of remaining codes after applying the properties in each column. Notice that odd number of changes in the properties is very useful in both types of properties, as it does not cause reduction in the number of codes even for the incremental ones.

		P3	P4	P5	P6	P7	P8
Inc-Index		0	791	0	731	0	335
Random		0	0	0	0	0	0
Inc-LI4	-	2	18	0	13	0	11
Ran-LI4	-	0	0	0	0	0	0
Greedy-LI4	-	2	24	0	13	0	9
Comb-LI4-LI6	Inc-LI4	-	0	0	0	0	0
	Ran-LI4	-	0	0	0	0	0
	Greedy-LI4	-	5	0	7	0	9

TABLE 5.2: Reduction in m (the number of Time-codes remaining in the trace-cycle) for different P_n. This means the trace-cycle length in which these codes can be used is reduced.

	D1b2	D2b2	D1b3	D2b3
Inc-Ind-1	15/200	12/200	70/200	81/200
Inc-Ind-3	151/200	114/200	112/200	125/200
Inc-Ind-7	149/200	112/200	149/200	174/200
Random-8	193/200	185/200	143/200	143/200

TABLE 5.3: The number of remaining time-codes in the trace-cycle after applying properties of on or two clock cycles of "constant delays (D1,D2,D1,D2) between 2 and 3 changes (b2,b3)" consecutively

In table 5.2, it could be seen how the greedy algorithm results still contains collisions when it comes to consecutive occurrences of changes, as opposed to the random but orthogonal codes. The naive incremental codes are worse though.

5.7 Dynamic Time-codes Generation

The main way we have used the generated time-codes so far, in our experiments, are hard-coding them into hardware, after they are first generated only once. This is acceptable in applications where there is enough on-chip area for storing the codes. If this was not the case, another solution is to use a generation function; in the sense of Pseudo Random Noise (PRN) generators.

Some of the time-codes generating algorithms presented here can be implemented in hardware; as a mean to save the area needed to store the codes. However, for such implementations, the generating times of the different codes are not similar. This variability makes running them while tracing is taking place takes unpredictable time. This implies the tracing clock shall be limited by the worst case/path

[2]Inc-Ind-k: means the incremental code **Inc-Index** with increments of weight k.

time of the time-codes generation. One workaround is to run them once, as part of the system initialization after reset. This approach can be used to generate non-fixed (configurable) time-codes, as the generated codes have to be saved on some memory anyway at the end.

Using PRN generators however cannot be done exactly the way it is used to generate PRN. It is not easy to find the feedback connections which results in codes that fulfill properties like LI-4 or LI-6. This leads to formulating another set of very interesting problems. One such statement of the problem can be put as: finding the connections for a set of fixed size of feedback shift registers, which can result in time-codes, that fulfill LI-4. Notice that the size of such registers should be at least the width of Timeprints (and codes). And the resulting codes should be taken parallel from the outputs of some selected registers. Because it is not acceptable that the bits generated are taken in series; the time for generating one code shall not be more than one clock-cycle. An extra circuit, not just the shift registers and their feedback connections, shall be needed to decide which registers outputs should be taken at a specific moment.

Notice that the resulting hardware is required at least to be smaller than the time-codes hard-coding size. This can represent a limit, which can be given as input to the codes-generator generating problem. This problem is at least hard as the Timestamps generating problem. One can think of solving it as trying to find Karnaugh map for resulting in the time codes; but this is not expected to lead more efficient area utilization, than hard-coded time-codes.

5.8 Chapter Summary

This chapter started with presenting the time-codes generating problem, and articulated variants of the problems so that they can be translated into code, which can actually generate such codes. The chapter also presented some algorithms that tries to solve these variants of the problem from several perspectives. Generating codes fully capable of distinguishing between traces which fulfills certain properties was also discussed. A brief analysis of the generating problem hardness is presented. The chapter ended by discussing the variant of having a time-code generating function, which can be implemented on hardware, to replace hard-coding the time-codes.

We also presented an overview of how some simple temporal properties can be used in enhancing the generation of timestamps encoding used in the timeprints-based monitoring. Using temporal properties in the case study shows the plausibility and potential of obtaining timestamps that produces more unique results. This is

a new way to look at the time-encoding, i.e. before we only focused on linear independence, which was not easy to extend beyond the 4th degree. Now by applying properties to existing timestamps-sets, we can obtain time-codes that are more capable pf producing unique results in the cases that are known to take place. Here, we simply have made more scattering of the similar solutions that could coincide, and avoided having them mapped to the same timeprint.

Chapter 6

Foundation 3: Reconstruction

This chapter is dedicated to *Reconstruction*, a cornerstone of our evidence-oriented tracing methodology and a mean to construct proofs of properties from Timeprints. The reconstruction is what makes it possible to restore from the Timeprints, –these infinitesimal size logs– the original traces or the temporal details which are required.

The chapter starts be formulating the Reconstruction as a problem. Then we describe, in detail, the approach we took to solve it. In short, we chose to describe the problem as an input formula to Satisfiability (SAT) or Satisfiability-module-Theory (SMT) solvers. This means describing logical relations between change-variables, either in Conjunctive Normal Form (CNF) for SAT solvers, or as an SMT formula, for SMT solvers. We encode all possible traces which can lead, within a specific trace-cycle, to the Timeprint logged at the end of this trace-cycle. We also encode the properties which we know to hold, and the ones which we want to check. A solver takes the input formula, of one or more trace-cycles, and finds a satisfying assignment (or the set of those) which lead to the Timeprint(s) at hand.

In our context, we want to find all such possible assignments, which can lead to the logged Timeprint(s), given the coded properties. Hence, the SAT variant which is being solved –when we want to find all possible traces– is the all-SAT. However, as shall be seen later in this chapter; this is not always the case. Sometimes, we just need to check or prove whether something has happened or not. In such cases, we formulate the check also into the input formula; and according to the formulation, finding a single satisfying solution (or the absence thereof) –i.e. the classical SAT– is enough.

After formulating the problem with its variants, we start addressing properties formulation into the reconstruction problem. An important aspect there is projecting traditional temporal properties into their equivalent counterparts described over trace-cycles. We also briefly discuss the inter-trace cycles properties descriptions and the stepped-reconstruction.

Lastly, before the chapter summary, counting the number of solutions in the reconstruction set is discussed. This counting can help us avoid the last step (until UNSAT is reached) in the All-SAT problem variant.

6.1 Trace-Cycle Reconstruction Problem (TC-$\mathscr{R}$)

Prologue: we use the symbol $\mathscr{R}$ for describing reconstruction in general. We might use that symbol without further specifications if it is clear from the context. If not otherwise mentioned, a reconstruction $\mathscr{R}(TP)$, shall express an ordered set of sets, representing a series, where each element in the ordered set corresponds to a trace-cycle tc, and each element contains a set of possible trace-cycle candidates which can result in the corresponding Timeprints trace tp, which can lead to logged Timeprints sequence tp, where $tp = tp_1.tp_2....tp_i.....$ When the Timeprint is not specified, it means we are considering any reconstruction set $\mathscr{R}$ in general; for any Timeprint. We distinguish between a full trace reconstruction and one trace-cycle reconstruction. We call the one trace-cycle reconstruction problem, *TC-*$\mathscr{R}$, or simply R, and we focus on it first. Then, full trace reconstruction is called *GT-*$\mathscr{R}$, standing for the general trace-reconstruction problem, is discussed afterwards, after discussing needed concepts. As mentioned before, the main idea which makes Timeprints efficient is dividing the reconstruction problem, into trace-cycles. This is why one trace-cycle reconstruction *TC-*$\mathscr{R}$ is the key unit of the more general one.

One trace-cycle reconstruction is not limited as it looks like in the first sight, and can lead to sufficient results and clear judgments in many situations. We start by considering it in detail first. Then we move forward with other considerations, before we discuss the more general *GT-*$\mathscr{R}$, as mentioned above.

For *TC-*$\mathscr{R}$, or for short R, The essential inputs are:

1) the pre-specified set A_{TE} of time-codes that contains m vectors, each of dimension d over **F**, denoted by a Time-Encoding $TE_{m\times d}$,

2) a Timeprint tp_i logged from a specific trace-cycle tc,

and optionally, one can specify properties, formulated one trace-cycle. Properties are going to be discussed later in 6.2. The output resulting from solving the *TC-*$\mathscr{R}$ reconstruction problem is a set R_{tc}, which contains all possible trace-cycles changes which can lead to the input Timeprint TP. The set of time-codes TE is known before the system goes to deployment, while timeprints are logged periodically all over the execution. For any one trace-cycle tc_i, for which we want to reconstruct its details,

the Timeprint tp_i is the one generated from the signal changes during the trace-cycle tc_i. Figure 6.1, below, illustrates a logged train of Timeprints.

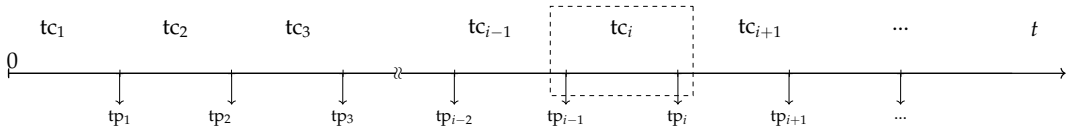


FIGURE 6.1: Trace-Cycles Over Time-line

The trace-cycle i highlighted/framed in Figure 6.1, contains a fixed number of clock-cycles, which we called the trace-cycle length m in Chapter 3. The change signal, which expresses instances of change which actually happened during this i^{th} trace-cycle, is called actual trace-cycle tc_i^a. The Timeprint tp_i results from tc_i^a triggering the d-wide time-codes in $TE_{m\times d}$ corresponding to ones in tc_i^a to be XOR'ed. When we are logging only the Timeprints, we do not know tc_i^a, but we know tp_i. From this Timeprint tp_i, we reconstruct the set of candidate trace-cycles tc_i^c, where c is an enumeration of the candidates inside the set of all possible change vectors which can lead to the logged time print. We denote this set as R_{tc_i} and define it as:

$$R_{tc_i} = \mathcal{R}(tp_i) = \bigcup_{\forall c} tc_i^c, \quad \text{where} \quad TE_{m\times d}.tc_i^c = tp_i.XOR \quad \text{and} \quad |tc_i^c| = tp_i.n \tag{6.1}$$

Notice that $|x|$, expresses the number of variables set to one in the bit-vector x [1]. Hence, R_{tc_i} the set of all trace-cycles tc_i^c which can result in tp_i. Notice that the i^{th} trace-cycle tc_i is represented by an m-long change vector $c_1, c_2, ...c_m$, where each c_j is a variable which can take either zero or one (i.e. an m-dimensional vector over **F**) expressing whether there was a change or not in the j^{th} clock-cycle within the i^{th} trace-cycle.

We know that the set R_{tc_i} contains the actual trace-cycle tc_i^a. i.e. $tc_i^a \in R_{tc_i}$, because it has –in reality– resulted in tp_i. Of course the underlying assumptions here are that the XOR hardware is correct, and the Timeprint is received correctly. These assumptions and practical, even when used in environments where errors can happen, well known error detection and correction methods can be applied, orthogonal to the suggested scheme.

6.1.1 TC-$\mathcal{R}$ Problem Formulation as Satisfiability Problem

Reconstruction function takes the logged Timeprint (the aggregated XOR + the number of changes) and any proven or run-time verified properties about the signal

[1] Recall that $|A|$, where A is a set of vectors, expresses the number of distinct vectors in the set.

change-pattern, and retrieves a set of candidate timings of change during the corresponding trace-cycle. The naive implementation of the reconstruction function blindly checks every possible change candidate tc^c_j, and if it leads to the Timeprint at hand, it is added to the reconstruction set. Instead, we used satisfiability solvers to smartly reach that set. SAT-solvers enjoy a long accumulated enhancements over more than 60 years, since Davis and Putnam's [54] paper. Even when trying to solve the all-SAT problem to find all the change trace-cycle candidates, which can lead to a Timeprint at hand, using SAT solvers is usually much faster than looping over all possible candidates to check whether they can lead to the logged Timeprint or not.

To formulate this as an $all - SAT$ problem, we use Conjunctive Normal Form (CNF). More about CNF and how SAT-solvers handle them can be found in [32]. Assignments of m-variables, expressing changes within a candidate trace-cycle, are going to be returned as a solution by the SAT-solver. Variables which are set to one in the solution returned, reflect that their corresponding time-code contribute, via XOR, to the Timeprint's XOR part $tp.XOR$. The number of changes $TP.n$ can also be encoded, limiting the number of variables allowed to be set to one in the solution to exactly $tp.n$.

This can be encoded in CNF as follows: First, a variable is assigned for each change that might occur. Then, we express the fact that if a change happened, the respective time-code gets into the XOR. We used CryptominiSAT [159] as it supports direct writing of XOR statements. In CryptominiSAT, an XOR clause contains variables that are XORed, while the XOR result is represented by at least one negated literal if the result is zero; otherwise no negated literal shall appear. For a b bits Timeprint, we write b XOR-clauses. The SAT-solver is then asked to find all possible trace-cycle candidates, that can lead when XORed to the Timeprint at hand. To do this, for each bit in the given Timeprint, a time-code which contains one at this bit position, triggers a one into the XOR function. Hence, the XOR clause corresponding to one bit in the XOR part of the Timeprint shall contain literals of the variables where there is a one in this bit within the corresponding time-code. In Cryptominisat, the default of an XOR clause is that its result is one; i.e. that it has to be satisfied. For one XOR clause, if the corresponding bit in the Timeprint is zero, it can be expressed by having any one literal negated. This corresponds to expressing that if one variable of these is not set to one, the result would have had been one as well. And only because it is negated, the original XOR result is actually zero; while the result of the written XOR clause is one. And because of the associativity property of the XOR, it does not at all matter which literal is negated.

To summarize, the XOR clauses in this form express the XOR-ing of Time-codes'

bits. If a change instance within the candidate *tc* is one, and the literal of this variable occurs in the XOR clause, it will get into the XOR. Otherwise, when the bit in a time-code is zero, nothing needs to be done even if a change happened in this instance; because the zero is transparent to the XOR, hence it will not affect the result anyway. This expresses that the variable will get into the XOR if, and only if, the variable corresponding to this instance is set to one in the candidate trace-cycle. The SAT solver, when given such XOR clauses, shall try to find a satisfying assignment of variables. The resulting satisfying assignment is just one possible solution, and an all-SAT is required to find the set of all possible assumptions. But this set can be huge if we are talking about more than few changes within thousands of clock-cycles. Hence, in the reconstruction problem we need to express also more than merely all what could have lead to the XOR result at hand. Considering that the XOR result is not the only thing we have; i.e. we still have the number of changes logged, besides our other knowledge about the system.

The number of changes, $TP.n$ is encoded as the cardinality, i.e. the number of variables which are allowed to take the value one, expressing a change has taken place. The cardinality encoding can take a lot of memory if done naively. We address this in more details next.

Cardinality Encoding

The number of changes which has taken place over a trace-cycle is expressed explicitly in part of the Timeprint, namely $TP.n$. We show here how to encode this in the CNF formula which we give an input to the SAT solver. As mentioned before, the number of changes is reflected, in the CNF formula, by the exact number of main variables which are assigned a one (or true). The notion *Main Variables* here refers to the ordered m variables, $c_1, c_2, ...c_m$ of the change-vector, corresponding to whether a change happened or not in the i^{th} clock-cycle within the trace-cycle at hand. This is as opposed to any auxiliary variables we shall use shortly for more efficient CNF formulas.

The direct way to express the cardinality (how many variables should be set to one), among the variables of a CNF formula, can be explained as follows. CNF formulas are AND's of clauses of OR statements, so we have to express that exactly this number is assigned to one using this AND of OR clauses logic. The direct way of doing this, is to have every possible combination of $m - n + 1$ variables in a clause; because this is how we express that at lease one of the $m - n + 1$ variables has to take the value one. But of course this very expensive in terms of the number of clauses;

and not practical at all when we start to talk about hundreds of clock-cycles in a trace-cycle. One text file expressing a CNF formula –with this form– for only 128 variables is already in Gigabytes.

An efficient way for representing cardinality is presented in [156]. The methods presented there use auxiliary variables. Because we don't know which variables exactly are going to be the ones assigned one, we have to use auxiliary variables, to express this possibility aspect. The methods in [156] can help describing that at most n variables are going to be assigned ones. And to express that exactly n variables takes one, we use new auxiliary variables $c_1^{'}, c_2^{'}, ...c_m^{'}$, expressing the negation of main variables. And for these, the cardinality is also expresses to be less than or equal to $(m-n)$ of the auxiliary $c_1^{'}, c_2^{'}, ...c_m^{'}$ variables are assigned one. This is in summary the way we describe cardinality of a change-vector.

Cardinality is actually a property, but because it is already explicitly logged as part of the Timeprints, it was discussed first. Next, we first briefly address adding properties to the reconstruction problem. But later in more details, another subsection is dedicated to this.

Properties Encoding

Cardinality has helped a lot in reducing the number of solutions to the all-SAT problem. But even then, the resulting number of solutions might still be very large; especially when none of the number of changes k, or no-changes $m-k$, is small relative ot the trace-cycle length. In many cases, the practicality of Timeprints depends on our ability to express the effect of system aspects, which are known and/or verified, on the change-vector candidate. As the main usage of Timeprints in the context of obtaining evidence, there is already a known context and an existing system. Both contexts and systems have many properties, of which their reflection on the change-vector can be encoded. Encoding logical relations can be always done using Tseiten transformation [134, 167], which uses auxiliary variables, to encode logical relation efficiently into CNF formula.

Next, we explore encoding the temporal properties, in bit more detail.

6.2 Encoding Temporal Properties

Here, we mean by properties: any feature which can be described over time using first order logic. The variables which are going to be used for the description are

change variables over clock-cycles. And these variables have a direct correspondence to the digital value signal, of which they express change. Relations between these main variables may require other auxiliary variables to express, more efficiently, relations between main variables.

Previously, examples of properties were given in Chapter 3, namely: the changes coming always in two consecutive trace-cycles, and that certain number of changes before some deadline. These properties for example are described as follows: within one trace-cycle of length m, we have m main variables. One variable x_i corresponds to the element of the change vector c_i. The property that all changes come in two consecutive trace-cycles can be expressed as $P_{2consec}$, as in Figure 6.2 below. This figure already shows the effect of applying this property in our running example since Chapter 3.

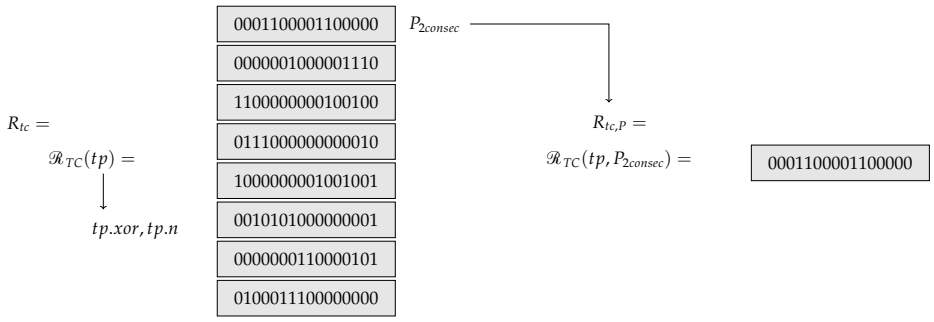


FIGURE 6.2: The reconstruction sets of the Timeprint TP only, and after using the Property $P_{2consec}$

Notice that $P_{2consec}$ may, in other cases rather in this example, have implications on two variables outside the current trace-cycle. Namely, for $i = 1$ and $i = m$, the condition $P_{2consec}$ written as:

$$(\forall(1 < i < m),\ \ (x_{i-1} = 0 \wedge x_i = 1) \implies (x_{i+1} = 1)) \wedge (\sum_{\forall(1<i<m)} (x_i) \text{ is even}) \quad (6.2)$$

shall imply that if $x_1 = 1 \wedge x_2 = 0$, the condition $P_{2consec}$ can still be satisfied if x_m of the previous trace-cycle $= 1$. We call this an implication to the previous trace-cycle, or simply *backward implication*. Similarly, the implication which results if $x_m = 1 \wedge x_{m-1} = 0$, is that x_1 of the next trace-cycle has to be $= 1$. This one is called *forward implications*. These various implications on adjacent trace-cycles are discussed in the next section.

The property that certain number of changes come before some deadline can be expressed using cardinality. The deadline d can be mapped to some i between 1 and

m in a specific trace-cycle. And the cardinality of variables $x_0...x_d$ can be encoded as above in 6.1.1.

The properties-encoding can also be extended to *spatial* properties, i.e. of various signals, as perceived by spatial abstraction in the sense described in 4.1. This leads to involving different change-vectors, corresponding to the various traced signals. More details about this comes in Section 8.4. For now we shall focus first on one signal and the temporal properties over one change-vector. Then in Chapter 8, spatial properties between different signals are discussed.

6.3 Implications on Neighbor Trace-Cycles

If no properties are used, a Timeprint (with both its parts: $.xor$ and $.n$) results in a reconstruction-set, that is independent from previous and next trace-cycles' reconstruction sets. However, if properties are used, it can happen that some of the solutions in the reconstruction set, have implications of adjacent trace-cycles. This means, some solutions, require for their validity, certain conditions to be fulfilled by adjacent trace-cycles. These implications, are tied to the exact solutions, implying them. There can be forward, backward or both side implications; depending on the properties used in the reconstruction.

To clarify the idea of implication, let's take an example from the previous Figure 6.2. The property $P_{2consec}$ can result in a reconstruction set that contain solutions, which have implications on adjacent trace-cycles as follows: if a construction-set contained the solution 1011000000000001, the two ones 1, at the beginning and end of this candidate, imply that the first bit in the previous and next change-vectors tc_{i-1} and tc_{i+1}, have to be also 1's in order for this solution to fulfill the $P_{2consec}$ property.

A composition of implications may extend to more than only the previous and next trace cycle; depending on the properties which were used to generate the reconstruction set. In general, for any length of properties, the implication chain shall end in the worst case at the reset, on the backward implication side, and at the trace-end on the forward implication side. This brings a form of verification problem similar to run-time verification problems which considers full execution trace, but with Timeprints as abstracted trails, instead of the full signal trace. For a temporal property of length l_p clock-cycles, and a trace-cycle of interest tc_n starting at t_1'th clock cycle and ending at t_2'th, the implications might span the interval from $t_1 - lp + 1$ to $t_2 + tp - 1$. When these boarders lie in the middle of some trace-cycles, the whole trace-cycle's reconstruction shall be affected. The implication itself shall be encoded

as taking place only within the interval it affects. But that encoding can be written only as part from the full trace-cycle reconstruction.

The type of implication also depends on whether this is the only solution, or there are other solutions in the reconstruction set. There are cases also where the implication itself cannot be satisfied by any of the solutions in the adjacent trace-cycles reconstruction sets. In these cases, that solution (for which implications cannot be satisfied) is removed as it is considered invalid. We shall get back to details of implications and proofs about their finiteness in section 6.5. But before that we need to classify the reconstruction targets, because they shall affect how the implications work.

6.4 Reconstruction Targets

Solving the *TC-*$\mathscr{R}$ results in a set of all possible candidate trace-cycle change-vectors, that can lead to a specific Timeprint tp at hand. Without any properties, this set can be very large, especially for non trivial numbers of changes. Getting the number of solution in a reconstruction set smaller is not the main target; although it is useful on its own. The targets from reconstruction in general can be classified into two main categories: 1) actual trace reconstruction, and 2) properties checking. The are explained with that order in the next subsections.

6.4.1 Actual Traces Reconstruction

Here the target is to reconstruct the exact actual trace which happened. As logs, we only have the trail of Timeprints logged from the execution, where from each Timeprint, we are sure that the corresponding trace-cycle belongs to the reconstruction set of that Timeprint. But we cannot know which one exactly from the Timeprint only; unless the reconstruction set contains one element only.

When the reconstruction sets contain more than one element, which is mostly the case, knowing which one exactly is the one which took place might still be required. Here, properties, which are known to hold over the trace-cycle – for example they could be run-time verified– can be use to reduce the size of the reconstruction set. Intuitively, these properties help excluding all solutions in the reconstruction set which do not fulfill them. This can be done before the reconstruction set is obtained in full for efficiency reasons. This means instead of obtaining all the solutions from a Timeprint, and then exclude some of them, while obtaining the solutions, the verified properties themselves are also encoded at the same time condition 6.1

is encoded. With assumptions, the reconstruction set $\mathcal{R}(tp, a)$, where a are the assumptions, can be rewritten similar to equation 6.1 as:

$$R_{tc_i} = \mathcal{R}(tp_i, a) = \bigcup_{\forall c} tc_i^c, \text{where } A_{TE}.tc_i^c = tp_i.XOR \ \wedge \ tc_i^c \models a \ \wedge \ |tc_i^c| = tp_i.n \tag{6.3}$$

where tc_i^c is a candidate/solution in the reconstruction set of the i^{th} trace-cycle.

Obtaining one exact solution is equivalent to that we are trying to do a full reconstruction of the signal, out of the Timeprint, using the help of properties we know they hold. This can be needed in cases where full reconstruction of exact cycle accurate details is required. While debugging for example, different properties expressing various assumptions of what might have gone wrong can be tested. Care must be taken as assumptions has to be formulated correctly, with all possibilities. Because when formulating as assumption and finding that the reconstruction set contains solutions, does not necessarily mean that this assumption holds or is satisfied. In such cases, a non-empty reconstruction set means the assumption can hold. In this case, an empty reconstruction set of the negated assumption, can only prove that the assumption holds. And only then, the reconstruction set of that assumption can be trusted to contain the actual solution.

After an assumption is proved to hold, if the reconstruction set still contains more than one solution, more assumptions can be applied until the exact one trace-cycle is obtained, or until it is enough to prove some assumption. This last case is similar to the next reconstruction target: checking properties. In general it is possible to check whether obtaining one solution is possible, given as set of run-time verification conditions, which can be implemented in the system. This can be also used to select suitable time-encoding that can provide one solution in some required cases. For example, the LI-4 encodings discussed in Chapter 5 guarantee obtaining one solution for all cases where there is only two changes in a trace-cycle. More properties known to hold, or that shall be run-time verified, can be also used similarly to produce time encoding that results in one solution for such traces. This means in cases where this actual trace reconstruction is expected to be required, defining the set of needed properties, that should be verified at run-time, to enable ending up with one solution for every reconstruction can be conducted before system deployment. This is not an easy problem, but a one worth solving for these situations.

6.4.2 Checking Properties via Reconstruction

A practical alternative to full/actual trace reconstruction is using Timeprints to only check properties. This can be done in a similar manner to checking assumptions,

mentioned in the previous subsection. Notice that also here, that a set containing some solutions might not guarantee that the property used in that reconstruction holds; a check of the negation emptiness is also mandatory. Similarly, if a property needs to be checked using Timeprints, then using an encoding which enable checking that property in all cases is required.

Checking properties, when done correctly, is more practical than reconstructing the exact trace. However, special care must be taken in the property formulation, so that the check result reflects –provably– whether the checked property holds. In the next chapter we call the ability of Timeprints to prove or refute properties *Evidence*, and we explain there in more details the needed conditions for evidence to suffice.

When the reconstruction target is to check a property, we do not care about elements/solutions of the reconstruction set. Rather, what matters here is the check itself. A property can be also formulated in CNF as mentioned in 6.2. Hence, the SAT solver' input CNF formula is going to contain the property which needs to be checked. Here, one can choose to formulate the property so that either SAT or UNSAT reflects that a property holds. For an existentially quantified property, when the result of solving this CNF is SAT, then the formulated property is satisfied. While for a universally quantified property, UNSAT should be used to express the satisfaction.

6.5 Implications Revisited

Now, after the difference between actual trace reconstruction and checking properties (as reconstruction-targets) is clear, we discuss the reconstruction implications in more detail for each of these cases.

First, we consider the case of one trace-cycle within an exact/actual trace reconstruction, as long as the assumptions used are verified, an implication-check might is needed in most of the cases. However, implications which might occur in some of the solutions, can be used to exclude these solutions, if the implication does not hold. If a reconstruction set, contains one solution, and that solution has an implication on some adjacent trace-cycle, then this implication has to be fulfilled. A check of that adjacent trace-cycle reconstruction fulfills the implication is considered as a consistency check, which is expected to hold (when it doesn't hold it means something is fundamentally wrong in the assumptions). By consistency check, we mean: a verified reconstruction set containing one solution, which has an implication, implies that at least one solution in the adjacent trace-cycle's reconstruction set fulfills

the implication, and only the solutions fulfilling this implication have to be considered. If that check did not hold, then probably it was reached by using assumptions which are not correct ot do not actually hold.

If the actual trace reconstruction already contain more than one trace-cycle, the implications can be checked with each step a reconstruction is calculated. In each step, if an implication on some adjacent trace-cycle does not hold, i.e. no solution in that adjacent trace-cycle fulfills the implication, the solution/candidate causing this implication has to be removed from the set of valid solutions in the original reconstruction set.

When talking about implications while checking properties, as reconstruction target, we have to distinguish between the assumptions, on one hand, and the checked properties themselves on the other hand. Implications associated with assumptions, as mentioned before, can be used to get rid of the implying candidates/-solutions, when the implications don't hold). But implications associated with the checked properties have to be inspected more carefully.

An implication which results from a SAT which indicates that the property holds has to be checked. And if the implication check failed, this solution shall be masked (considered invalid) and the SAT problem solution has to continue until finding another solution; either without implication, or with one that is satisfied. An UNSAT indicating that a property holds cannot have an implication. However, if while waiting for an UNSAT as a proof that a property hold, a solution was found which have an implication, that solution's implication has to be checked. If the implication is satisfiable, then the property might not hold. If the implication is not satisfiable, then this solution is masked and the solving continues, where it stopped, until either UNSAT is reached or another solution, indicating violation, is found. And the same applies again if the found solution had implication(s).

6.5.1 Stepped (Incremental) Reconstruction

As discusses in this section, implications can sometimes be necessary to calculate some check-results, but in other cases they can act only as consistency checks. In both cases, calculating them can be done in a stepped manner, e.g. after obtaining one trace-cycle's reconstruction set, if the set contains implications, they can be done afterwards. Also if there was a check, after a SAT result which have implications on adjacent trace-cycle(s), the implied check can be triggered as a next step after reaching that SAT result.

The stepped reconstruction scheme is very useful in giving an insight about the result much earlier, than if we waited for the whole chain of implications consistency check. This can be seen as a preliminary verdict followed –after finishing the whole chain of implications– by a final verdict.

Steps in the reconstruction can also be found when the window of interest (the period over which we want to conclude something) contains more than one trace-cycle. Each trace-cycle's reconstruction can be done separately, even on parallel processes, then implications resolution for each on the other can be also staged similarly.

6.5.2 Implications Illustrated on Full Trace

A reconstruction *TC-*$\mathscr{R}$ for a trace-cycle tc_i, denoted by R_{oi} (to denote that there are not properties involved). A forward implication of R_{oi} is denoted by $\overrightarrow{R_{oi}}$. Backward implication by a reconstruction R_{oi}, is denoted by $\overleftarrow{R_{oi}}$. From a sequence of Timeprints $tp_1, tp_2, ...tp_i, tp_{i+1}, ...$, a set of full reconstructed traces, $\overleftrightarrow{R}$, can be obtained as:

$$\overleftrightarrow{R} = R_{o1} \Cap \overleftarrow{R_{o2}} \cdot \overrightarrow{R_{o1}} \Cap R_{o2} \Cap \overleftarrow{R_{o3}} \cdot \overrightarrow{R_{o2}} \Cap R_{o3} \Cap \overleftarrow{R_{o4}} \cdot ... \cdot \overrightarrow{R_{ol-1}} \Cap R_{ol} \tag{6.4}$$

where $\Cap$ is a concatenation conjunction, connecting a subset of specific elements/solutions, within a neighbor-reconstruction set, by a one-to-one implication, towards another subset containing specific elements/solutions in the main reconstruction set of a trace-cycle. The result, $\overleftrightarrow{R}$ is actually a list of sets, each element of the list has is a reconstruction set, within which some solutions might be connected to other solutions, in adjacent reconstruction sets.

Notice that, with the stepped reconstruction, the set $\overleftrightarrow{R}$ is temporarily –or stepwise– defined, and shall be concretized with each step, and only considered final after all forward and backward implications are all resolved.

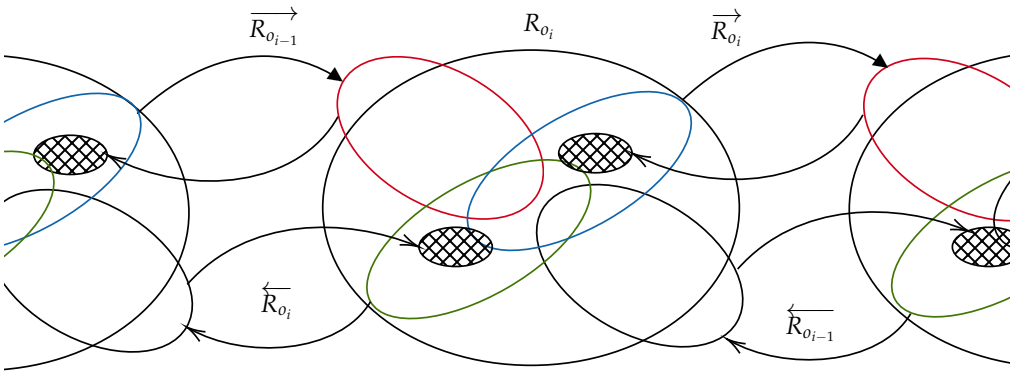


FIGURE 6.3: Reconstruction Implications

This *local one-to-one conjunction over solutions* is explained in Figure 6.3, and later in Example 3. The figure shows an i^{th} trace-cycle in the middle with implications on previous and next trace-cycles. The blue ellipse represent the subset of reconstruction set which have implication on the next trace-cycle. That implication is shown as the red subset, which intersects with the next trace-cycle's reconstruction set, the solutions which might be outside that intersection are those implied by the i^{th} trace-cycle, but cannot be real because the $(i+1)^{th}$ trace-cycle do not contain them. Hence, the elements in the blue subset implying these have to be removed (shown as the hashed ellipses). The same applies to the green ellipse, which describes the backward implication on the previous trace-cycle.

Example 3 *Let's consider a trace-cycle, where its* $tp_i.XOR = 00000001$ *and* $tp_i.n = 4$*, as in our running example. But here, we do not know the original trace. We only have the Timeprints trace* $tp = tp_0.tp_1....tp_{i-1}.tp_i.tp_{i+1}.....$ *Let's assume we had the following readings in that Timeprints log:* $tp_{i-1}.XOR = 01111110$ *and* $tp_{i-1}.n = 3$ *,* $tp_i.XOR = 00000001$ *and* $tp_i.n = 4$*, and finally* $tp_{i+1}.XOR = 10011010$ *and* $tp_{i-1}.n = 4$*. We also use the same time encoding* TE *shown here again in Figure 6.4. We look now at the previous and next trace-cycles to see when and how the implications from adjacent trace-cycles affect a trace-cycle's reconstruction set.*

The reconstruction set R_o *for each of the three trace-cycles* tc_{i-1}, tc_i, tc_{i+1} *are shown in Figure 6.4. Given is that the property* P_x *holds, where* P_x *is the property that changes come consecutive, either exactly in directly 2 consecutive clock-cycles (i.e.* $P_{2consec}$*) or separated by a maximum of one clock-cycle delay. All the deeply dark shaded solutions are then excluded, as they do not fulfill* P_x*. We enumerated the solutions (in red) to facilitate explaining their implications.*

To illustrate the implications we start from trace-cycle i. The three solutions remaining in the reconstruction set have the following implications. $R_{o_i}.1$ does not have any ones on the edges, hence, this implies that the last element in the change vector of the previous trace-cycle $i-1$ can only be one as a second (the consecutive) change. Similarly to its next trace-cycle tc_{i+1}. On the contrary, $R_{o_i}.4$ does have two ones before the last zero at both edges. This has the direct implications that, in order for this solution to be a valid trace-cycle which fulfills P_x, there must be a one at the end of tc_{i-1}, so that the one at its beginning be the second in the two consecutive ones; and there must also be a one at the beginning of tc_{i+1}. $R_{o_i}.7$ does have one 1 at one edge, but that one –unlike that of $R_{o_i}.4$– is a second one in the consecutive ones, hence, it does not necessitate that the first bit in the change vector of the next

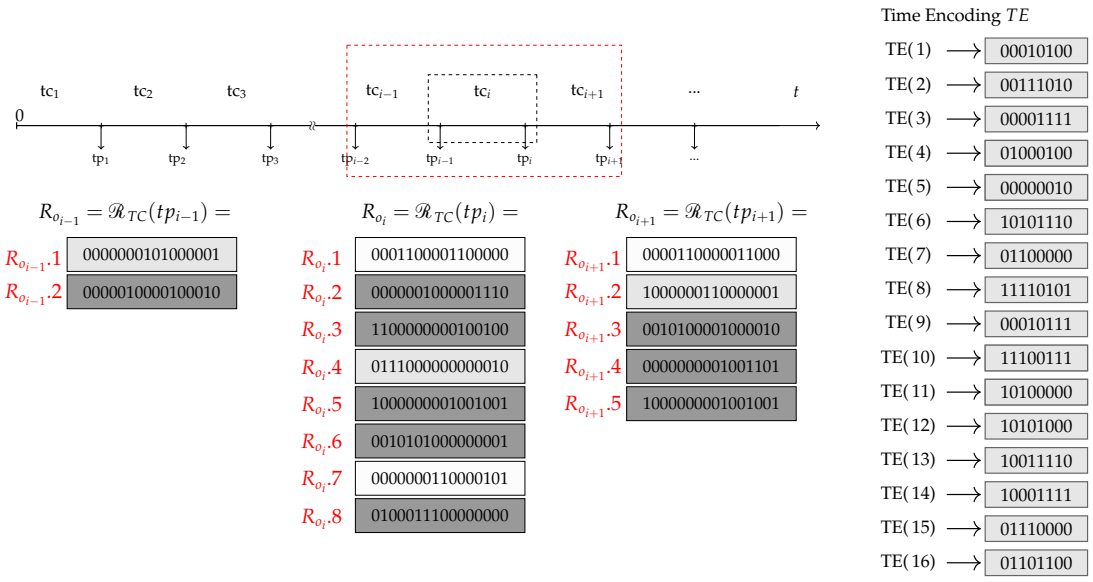


FIGURE 6.4: Three trace-cycles, their Timeprints and their respective reconstruction sets

trace-cycle be one. The implications of the solutions of R_{o_i} on $R_{o_{i+1}}$ (which we called above forward implications $\overrightarrow{R_{o_i}}$) are all met by some solutions in $R_{o_{i+1}}$, hence, there are no solutions in either R_{o_i} or in $R_{o_{i+1}}$ that are removed by these implication (the hashed areas in Figure 6.3). However, the implications paired some solutions in R_{o_i} with their respective counterparts in $R_{o_{i+1}}$. Namely, $R_{o_i}.1$ and $R_{o_i}.7$ can only be solutions if $R_{o_{i+1}}.1$ is the actual trace-cycle, while if $R_{o_i}.4$ can be a solution only if $R_{o_{i+1}}.2$ is the actual solution.

If we look at the backward implication $\overleftarrow{R_{o_i}}$, the implications made by $R_{o_i}.1$ and $R_{o_i}.7$ are not met by the only solution in $R_{o_{i-1}}$ which fulfills P_x. Here, the $R_{o_{i-1}}$'s forward implication $\overrightarrow{R_{o_{i-1}}}$ removes the two solutions $R_{o_i}.1$ and $R_{o_i}.7$, resembling the hashed area in Figure 6.3. Notice that the implications in this example left us with only one solution. This illustrates how useful implications can be in reconstructing the actual trace. Here, we have shown how the *local one-to-one conjunction* $\Cap$ *over solutions* in equation 6.4 is actually applied. We leave a full formal definition for future work. It is also worth mentioning that defining the property P_x formally is not straight forward. However, as our trace-cycle length m is finite, it is always possible to describe any property in terms of all its possible (finite) traces/solutions.

Notice that the implication illustrated here are focused on those resulting from local properties; or one trace-cycle reconstruction. For the implications of more global properties, which are described over the whole trace, see discussion in section 8.2. Before we go there, we briefly here discuss the hardness of the *TC-*$\mathscr{R}$ problem, counting the number of solutions, before concluding the chapter.

6.6 Hardness of TC-$\mathscr{R}$

The reconstruction problem in its simplest form: given a Timeprint ($tp.XOR$) obtain all candidates/solutions which could lead to it, is actually a Subset Sum problem [94]. The set has m codes, and a subset of $tp.n$ elements, of which the corresponding time-codes are added to result in $tp.XOR$. Subset Sum problem is known to be NP-hard, according to the classical problems hardness classification [70]. For $\mathscr{R}_{TC}(xor, n)$, the $tp.xor$ is given, and the subset of $tp.n$ time-codes, among the m codes, is to be found. All solutions leading to $tp.xor$ are required to be obtained anyway. Which makes our reconstruction problem a counting/enumeration version of an NP-hard problem, according to that classical complexity definition. This justifies using SAT-solvers to do the reconstruction problem, and that this might be the most efficient way to do it; utilizing advances in recent SAT solvers.

Notice that having the *Checking Properties*, as a reconstruction target, reduces – not only the time needed to reach a checking result– but also the reconstruction problem's complexity into NP-complete.

It can be shown that all the resultant candidates, which would result in a certain Timeprint's XOR part, can be described as kernel plus image. But his form results in unacceptably large number of solutions; this is why at least the form with the number of changes is necessary. When adding other properties, (although this can be thought of as making the problem harder, in the sense that adding the cardinality did) the problem search space is pruned, causing the SAT solving to be faster.

6.7 Counting Number of Solutions

The number of solutions is useful in determining the size of the reconstruction set. It can save a lot of time when solving the $all - SAT$ problem (avoiding the last computation till reaching the UNSAT). Notice that the number of solutions for the bare reconstruction (without properties) follows a normal distribution with respect to the number of changes. This already gives a hint of how long should the trace-cycle be. Average number of changes in a trace-cycle should not be high to the level that there are so many solutions; which cannot anymore be pruned by verified properties. In [44] we describe an algorithms, that can be used to calculate efficiently the number of solutions.

6.8 Chapter Summary

In this chapter, we explained methods we used for reconstructing the original change vector, within all possible change-vector candidates. In summary, a signal trace is composed of consecutive trace-cycles, each contains m clock cycles. Within one trace-cycle each clock cycle is assigned a bitvector in $\mathbf{F}^d$ uniquely identifying it. Roughly, the logging procedure adds up all vectors of clock cycles, where the traced signal flips from 0 to 1 or vice versa. The result is the target vector tp. Moreover, the procedure records the precise number k of changes. In this setting, a witness is a reconstruction of the traced signal. The characteristic of the problem is that the size of TE is typically large while k is small. This is due to the fact that the interval of clock cycles is comparably long, while a change in the signal only happens rarely. The logging mechanism is typically useful for post-failure analysis. Once a failure occurs, the value of the target vector tp that was logged is retrieved. Subsequently, all possible traces are checked to see if they lead to the specific target vector. The problem was presented in various forms (with, and without properties). Various forms were formulated and encoded as a satisfiability problem.

We differentiated between reconstructing the full set of candidate traces and the reconstruction for checking specific properties. We generically formulated checking global temporal properties using reconstruction. The limitations of the reconstruction implications are discussed later in Chapter 8. Examples of the reconstruction problem (and its solution) in many its flavors are presented later in the case-studies discussed in Chapter 9. In the next chapter (Chapter 7), we focus on the evidence of Timeprints, the core of the proposed evidence-oriented verification methodology.

Chapter 7

Foundation 4: Evidence of Timeprints

Evidence is needed to build correct and just conclusions. Judgments are made by humans all the time and in different contexts. Whether these judgments are objective, or based on well-rooted evidence, is a relative matter. When it comes to jurisdictions[1], there are relatively well defined roles and laws to base judgments upon. Among these rules, evidence plays a central role. In our context, to judge whether a signal violated its specification or not, an evidence is needed. And the role of Timeprints as evidence is the subject of this chapter.

If an autonomous or semi-autonomous system failed or malfunctioned, and this failure/malfunction lead to damages, who exactly is responsible/liable? Currently, evidences which can be used in such situations can be classified into four categories: 1) failure visible symptoms, perceived either by humans, like eye witnesses or operators, *or* other systems, like street cameras, adjacent cars, ...etc, 2) standard diagnostics and logs, as every industry defines required standardized system health information, to be regularly logged and stored, 3) system manufacturer custom logs, if at all exist and made available, and 4) process, code and other design artifacts, which can be consulted to check for inconsistencies, or to verify that they can lead to the visible symptoms in the case encountered. For example, following the crashes of two Boeing airplanes, many reports from various entities were issued, [4, 78, 126, 138, 160, 169]. All these reports contained information about the communication between pilots and airports, diagnostics (sensor readings) and commands issued by the system and pilots. This data was reported by the system about itself. Only one [65] relied additionally, in their judgment, on details of witness testimonies regarding design procedures, documentation and internal employees communications. Relying on

[1] Jurisdiction is he official power to make legal decisions and judgments.

such coarse self-reported data is not enough transparent, and that reporting accuracy level cannot reflect the required details of execution.

In this chapter, we show how Timeprints, which have been shown to provide information from abstractions (via reconstruction), can function as evidence. We first define what we mean here by the word *evidence*. Then, the context, conditions of using Timeprints as evidence are presented. As example, Timeprints' ability to describe evidence of *Delays* over a trace-cycle is illustrated.

7.1 What is Evidence?

A trusted piece of information can work as evidence that something happened or not, when it is a direct result of the happening, and at the same time, cannot be a result of another happening. Oxford's Lexico dictionary defines evidence as: " the available body of facts or information indicating whether a belief or proposition is true or valid." [2] To apply this in the context of obtaining evidence about occurrence of signals in reality versus their specifications, the framework of considering Timeprints as evidence is proposed in this chapter. The framework is composed of theoretical structural parts; as well as practical tools. From theory, we borrow concepts from *Theory of Evidence* and propose a practical proof structure, in which Timeprints and system's specifications are inputs, and the output is a verdict, attesting weather the input proposition holds.

Evidence-Oriented Tracing To make a trace act as evidence, it has to be indicative of that a belief (or property in our notions) is true. Thus, what needs to be in an evident trace, is enough data to prove or refute a target property or proposition. To realize this, we change the focus of tracing from tracing system's states, to a more evidence oriented tracing as follows:

- Instead of reasoning about monitoring finite or infinite traces, we shift the orientation towards looking for evidence over intervals or time-windows of interest, among continuously logged traces. Given this, we ask if certain amount of tracing data is enough to make a trace evident.
- Instead of focusing on design space exploration and checking all possible executions of a design, the focus is turned into the one logged trace of actual

[2] Cambridge's dictionary defined evidence as: "Anything that helps to prove something is or is not true". On Merriam-Webster a: an outward sign : INDICATION, b : something that furnishes proof : TESTIMONY, specifically : something legally submitted to a tribunal to ascertain the truth of a matter

execution is at hand. The problem then becomes: exploring possible realities of what actually happened and would result in the logged abstract trace.

- The tracing of system execution becomes tied to certain system signals: inputs, outputs, and/or intermediate signals, instead of checking the design.
- Run-time checks are used as invariants to help narrowing the space which the trace-abstraction indicates.
- Evidence oriented tracing is done continuously all over the execution of the running system. However, checking the properties and looking for evidence is conducted only on-demand e.g. when something goes wrong, and hence root-cause analysis requires obtaining evidence to support or refute assumptions. This means checks are done by reconstruction of the trace-cycles containing the suspected time intervals only.

These fundamental differences to the classical all-traces verification enable doing more efficient checks of properties over trace-cycles as in the following subsection.

7.1.1 Orienting Verification Towards Evidence

Verifying increasingly complex systems is a moving target. Machine learning results in updated algorithms, which have not been though of initially at verification time. Autonomous systems are required to operate independently for long time, during which it faces challenges which designers might not have thought about during design time. Thus verification's current focus on design specifications cannot cope with such cases. It is almost impossible to cover all possible behaviors of such systems by testing and/or simulation, simply because these cannot be all defined at design time, and even if we assume they could be, the number of variables makes all possible scenarios and tests required unpractical. Formal verification do not only attempt to solve the need for exhaustive and prohibitive testing/simulations, but also to provide rigid and reliable mathematical grounds for proving systems properties. Yet, formally verifying exact large and complex system designs is computationally hard. Hence, abstractions are used at various levels. However, abstraction might lead to ignoring details, which might turn out to be important at run-time.

To extend the strength of formal verification to checking the system's performance, while running in field, online monitoring emerged. Run-time monitoring is one form of Run-Time Verification (RV) [101]. RV overcomes part of the computational complexity of formal methods by focusing only on the one trace of execution,

either in simulation or at actual run-time. Also unlike testing, which happens only before production, RV can be applied all over the actual run-time during deployment.

Figure 7.1 shows going from design verification to run-time verification, via checking one trace of execution, instead of the full model, versus trace-specifications, deduced from system specs. However, RV, when done online, can only check what the monitor has been configured to check. This makes RV not sufficient for debugging and monitoring unexpected problems.

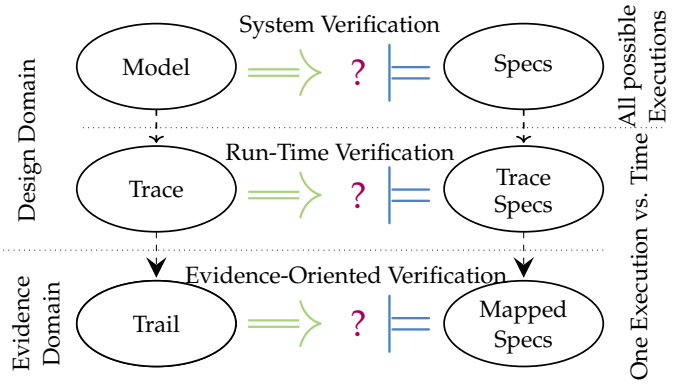


FIGURE 7.1: From System to Evidence-Oriented Verification

Timeprints presented here are refined traces, i.e. trails, which abstracts one trace of execution, and can be used to check it against some properties; including ones which have not been defined nor expected at design time. The target is to make these trails able to act as evidence, and orient our focus on making them provide proper and provable level of trust. In that case, as we don't have full signals, but rather an abstraction, or a trail of it. Where a trail in our concrete case is a sequence of Timeprints. A trail is checked against *mapped specifications*. This mapping is a projection of the original system or trace specifications, (checked using run-time assertions for example) onto the Timeprints-encoded trails domain. This projection (mapping) is going to be clarified more later in this chapter. By this we obtain Evidence-Oriented Verification (EOV), as shown in the bottom of Figure 7.1. [3] EOV is focused on obtaining the evidence, rather than tracing a complete signal trace. Before we clarify this in detail, we introduce theory of evidence, as we need it in defining EOV.

[3]Notice that the three approaches in Figure 7.1 do not compete, but rather complement each other.

7.1.2 Theory of Evidence

In the 70's, Shafer presented in his book [151] "Mathematical Theory of Evidence", the today's known as Dempster/Shafer's theory of making decisions under combined uncertainties. In the first sight, the most attractive benefit of using this theory, instead of the more common Bayes theorem [27], is the former clear terminology when it comes to evidence and making judgments. Baye's theorem's formulation uses chances/probabilities, which are not intuitively the same as belief/evidence. For example, when one says: there is a chance x that this event happen, or this event happened with probability x, the statement always raises an implication: "but there is a chance it didn't, even if small!" and what actually happened can exactly fall in the remaining $1 - x$. While using the terminology of *Theory of Evidence*, one would say: a belief function supports some evidence (that the event happened) with degree x. This statement in our context sounds not only more complete, but is also naturally focused on relation between facts and inferences drawn based on them. [4] Facts that actually happened (existing evidence) are at one side, and those inferences, we are trying to make based on such evidence, are on the other. The theory of evidence addresses relation between the two, and provides tools to assess how much an existing evidence supports a belief in certain hypothesis. It does so by describing the degree of belief supporting an inference, based on facts or other inferences. For a Timeprint to act as evidence, degree of belief of "1" is needed.

Beliefs, in the theory of evidence, are different from probabilities, in their freedom from the concept of randomness, which is fundamental in defining the chance. Probabilities and chances are very useful when applied in contexts like received signal demodulation and decoding, using stochastic processes in communication systems, because of the inherent receiver's ignorance of the received signal on one hand, and the plausibility of modeling transmission channels statistically on the other. But when it comes to extracting evidence about what might have went wrong in an unexpected context, this concept of randomness-distribution does not fit for this particular application. Here, the given facts can contribute into reaching an evident degree of belief (i.e. 1), based on other existing facts and their relations, not merely on a priori defined probability distributions. [5]

[4] Of course the terminology was not the only reason to favor using this theory for evidence analysis. However, that was the first source of attraction, and as a consequence of clear and accurate terminology other benefits followed.

[5] Probabilities distribution can still be used, but with a cautious well-studied and justified reasoning behind doing so. And the Dempster/Shafer's theory does not prevent from using such distributions.

Why Theory of Evidence? Unlike Bayes theorem, the theory of evidence is not focused on the act of judgment based on probabilities, but it rather addresses methods of combining the underlying beliefs (or partial-beliefs) to draw conclusions in non ordinary cases i.e. without the prior assumed probability distribution. In this sense Bayes theorem is a special case of the theory of evidence, in which the wights of belief are probabilities.

Theory of evidence uses the rather abstract weight functions instead of probabilities, as degrees of belief. These weight functions are used to relate the degrees of belief between the different possible outcomes. In our reconstruction set for example, it allows expressing that only one element in the set has actually happened, but we do not yet know which exact one. There is a clear notion which stresses that any subset of such a reconstruction set cannot be said to contain the exact one that happened; hence, encouraging refraining from jumping to assign probabilities over such subsets. As a result of this, theory of evidence considers Bayesian Belief functions as a subset of the class containing all belief functions.

Degrees of belief are updated based on outcomes, according to user defined functions, instead of conditional probabilities in Baye's theorem. This is more inline and intuitive when used with our reconstruction set, where the elements can be very far from each other (and from the actual solution). In many fields, known distributions are used for defined classes of applications. There, the assumption is implicitly made, that such probability distributions are representing reality, to a fair extent. If we consider Timeprints as tools to infer or check signals behavior, the transformation done in Timeprints domain makes the classical distributions almost useless. The major advantage of the Timeprints domain is: the classically close data (near to each other in the domain where distributions typically assume) are scattered over the timeprints domain, within which, checks can be more distinctively made.

Another fundamental difference between theory of evidence and other chance-based probability theories, is the wider and more flexible room provided for reasoning about what we do not know. For example, consider a reconstruction set, over which we do not know which one of its n elements has taken place. If uniform probability distribution is used, a subset containing $n - 1$, of the reconstruction set elements, is expected to have probability more than the subset containing any one last remaining element. But we know this is definitely wrong in the case where the one element is the correct one. Theory of evidence provides the capability of expressing this: that we know the element is in the set, and we know that only one happened, but we just don't know which one. This makes theory of evidence fundamental in representing the power of Timeprints. The theory also provides tools

to control derivations and relations of belief functions, in a more controllable way than using global distributions. For example, when receiving a signal, which experienced certain noise, a noise model can be mapped to a probability distribution. When Timeprints abstractions are used, the resulting loss in the signal depends on the known binary encoding, rather than noise. Hence, the sense of uncertainty in Timeprints reconstruction does not result from what we are trying to measure, but rather in how we did the abstraction in the first place. That detailed ability to express the loss which results from the Timeprints-abstraction (which can be expressed under the representation of ignorance in theory of evidence, instead of the classical uncertainty) is useful to express many concepts related to Timeprints and traces reconstruction from them.

Criticism of Theory of Evidence The criticism theory of evidence received, can be referred to misuse of the freedom given by the theory in describing the belief, rather than problems in the theory itself. Illustrative instances of such criticism can be found in [59, 186]. We use here this freedom without allocating probabilities at all, hence there is no way to run into the cases mentioned in these instances. Rather, we use the subsets to either indicate existence of a definite evidence (i.e. belief = 1) or absence thereof. Before we show how theory evidence is used with Timeprints in detail, we first motivate the importance of logging traces that can act as evidence.

7.2 Motivating Example

Consider a simplified down-scaled toy autonomous car, where an ultrasonic sensor is installed and used to detect obstacles at distance $\leq d$ as shown in Figure 7.2[6].

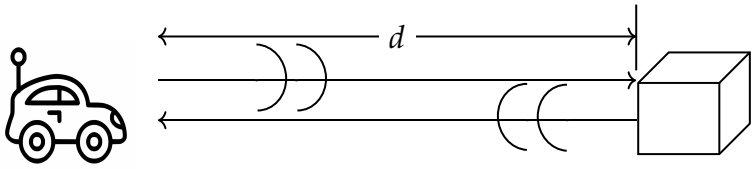


FIGURE 7.2: Toy Car Distance to Collision and Echo Signal

The sensor chip output signal goes high as it sends the ultrasonic signal, (which is done with periodicity T seconds as shown in Figure 7.3) and the signal goes low when the echo is received. The time interval after which the reflection is received indicates the distance. There is a timeout t_{out} seconds after which the sensor does not

[6]Toy car picture taken from https://clipground.com/images/toy-car-clipart-black-and-white-3.jpg

wait anymore for the reflection, which implies objects are far enough that no action is needed to stop or to avoid them. If the reflection is received before the timeout, the sensor produces an echo signal; which implies there is an obstacle at a distance of d or less. A simple micro-controller circuit reads that echo signal and issues a brake command, in the form of pulses. These pulses are sent to the brakes mounted on the rear wheels shaft. If three braking pulses were received within a time period τ_3, the car shall stop after at most t_s seconds from the reception of these three pulses. The relevant promises to our up-coming analysis are given in Table. 7.1. These promises are made by three different modules (Brakes, Controller and the Sensor modules) to fulfill a higher level requirements: that the car should not hit an obstacle which appears, at a distance of d or more in front of it.

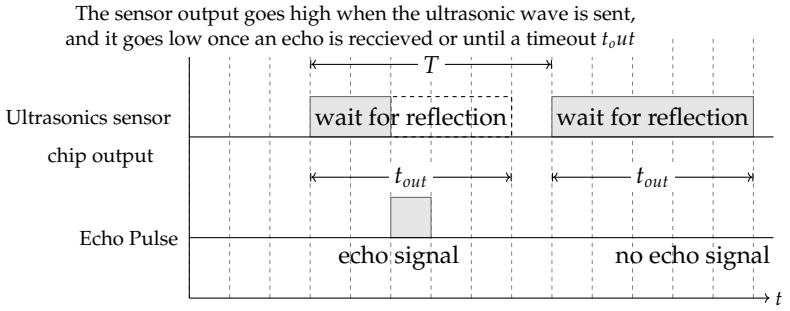


FIGURE 7.3: Ultrasonic Signal and Echo Pulses

The three modules (the sensor, controller and brakes) are connected in cascade, as shown in Figure 7.4. That the car shall stop as a conclusion from these assumptions is drawn from that the maximum velocity of the toy-car is v_{max} which satisfy:

$$v_{max}(t_c + \tau_3 + t_s) < d \tag{7.1}$$

where t_c is the maximum acceptable delay between the issue of an echo signal and the issuing of the respective brake pulse by the controller, τ_3 is the period in which if three brake pulses were issued the car should stop, and t_s is the maximum time since the third pulse issue and the instance in which the car stops. These are also shown in the promises $P1$, $P2$ and $P3$ in Table 7.1.

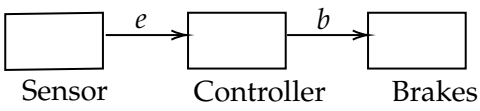


FIGURE 7.4: Output e of the sensor and b of the Controller

#	Promise	Module
$P1$	Three b brake pulses within a period $\leq \tau_3$ will stop the toy-car within t_s from the third brake pulse	Brakes
$P2$	After an echo pulse, a brake pulse is issued within t_c.	Controller
$P3$	Echo pulses are not later than t_e apart when there is an obstacle at distance d or less.	Sensor

TABLE 7.1: Properties expected to hold based on promises made by braking module, micro-controller, and ultrasonic sensor

7.2.1 Promises as Temporal Properties

The braking module has an input pulse signal b that comes from the micro-controller. The brakes shall halt the car upon receiving such three pulses within τ_3.[7]

The promises in Table 7.1 can be described by logical formulas and temporal logic in a straight forward manner. For example, $P2$ can be written in Metric Temporal Logic (MTL) as:

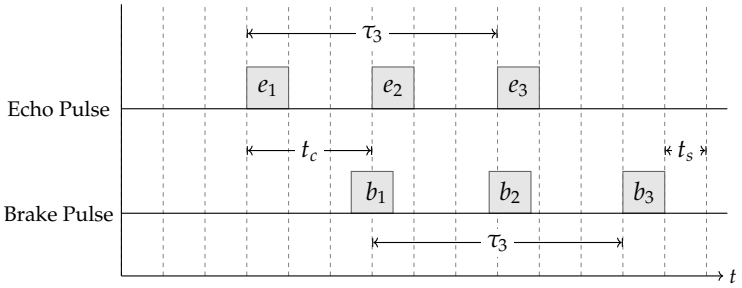
$$e_i \implies \diamond_{[0,t_c]} b_i \tag{7.2}$$

This formulation uses First Order Metric Temporal logic (FO-MTL), as we did in section 4.3. In general, the formulation of these properties in LTL or FO-MTL can be carried on as clarified in [47] for first order LTL with event freeze, and as in [23] for a formulation using first order metric temporal logic FO-MTL.

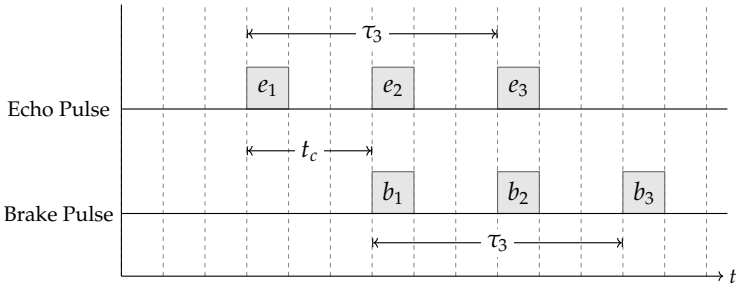
7.2.2 Responsibility for Failures

Figure 7.5 shows two different cases, in which the three echo signals, and their respective three braking pulses are shown. The two figures (A) and (B) shows the last signals before the toy car hit the obstacle. In the first figure, i.e. (A), all the three promises or properties in table 7.1 are shown to hold. And clearly, the car did not stop because the three brake pulses are coming within a period longer than τ_3. Here, all the modules behaved exactly as they should. But the specifications were flawed that when they hold the situation can come to an accident. The problem here was in the implicit assumption that three echo pulses within τ_3 shall result in three brake ones also within τ_3. This implicit assumption is not true as the brake can be issued at any time after the echo pulse is received, and this can vary from one pulse to the other.

[7] There are of course many other assumptions, about the width and characteristics of the pulse. But for the sake of simplicity, we consider in this toy example, there is only one pulse width, and all pulses fulfills the specs needed by the brake-module, to consider that the input pulse is correct. Such assumptions are reasonable, and can be checked in reality by on-line monitors that verify at run-time their fulfillment.



(A) All promises hold. Promise P_1 holds because the three brake pulses took place within more than τ_3 apart



(B) Promise P_1 is violated, as three brake pulses took place within τ_3 but the car did not stop as required

FIGURE 7.5: Two scenarios where the car hit the obstacle.

In the second part, i.e. (B) of Figure 7.5, according to $P1$ in Table 7.1, the car should have stopped after the reception of the three brake pulses, irrespective of any other assumption or condition. So in this particular trace, the brake module is definitely a major contributor to the failure.

Having such detailed traces enabled us to discover the problem in the assumptions. Although this problem could have been discovered by a proper use of model-checker, failing to do so properly can go un-discovered if it were not for detailed traces. In the both cases, no run-time monitors could have flagged any problem; and if it were not for the detailed traces, we would have no clue who to blame for the accident or what exactly the problem was.

This example shows that traces can give us a clear picture of what actually happened, irrespective of which design-problems do (not) exist in the system. In contrast, the main focus of the formal verification communities was the design artifacts. On one hand, because obviously they run the system, and on the other hand because they are abstract artifacts, where mathematical verification and model-checking can be applied. There is no doubt that formal verification, in its current form is necessary. However, it cannot cover many aspects in today's complex systems, which accurate tracing can cover. Relying on design-based formal methods only can mask sporadic bugs that result from factors not considered during design.

Discriminating between the failure causes this way –based on traces– is not only important for liability assignment. But when systems fail although they are correct according to their specifications, we want to learn about what was missed and caused the failure. In this example, case (B), if we just went through the process; it is likely to find the problem in the specs and consider it as the failure root-cause, and miss the braking module problem. However, the problem in the specs should have been discovered by proper use of formal methods; and the masked brake-module problems would persist and continue to cause problems. But if we did both: reviewing the process artifacts **and** inspecting in-field traces; we would fix both errors.

In practice, detailed traces (which can show such problems) incur unacceptable amounts of data. Here, using Timeprints appears as a potential solution. Yet, Timeprints might –in many cases– not be able to reconstruct the full traces. Hence, we need to define when Timeprints can be used as evidence. This includes designing a time encoding which enables Timeprints to: on one hand cover as much as possible of the unknown/unexpected properties, *and* be evident in the cases where a definite evidence is deemed desirable (example is the need of evidence of delays).

7.3 Timeprints as Evidence

A trace t is evident if it shows/indicates with probability –or degree of belief– equals one that something (e.g. a property) holds. As in our motivating example, a trace which contains the value of the signal every clock-cycle is evident about the time in which the signal took one value or another. If we want to use Timeprints instead of a full trace, for a Timeprint to be evident, it has to show/indicate that something (e.g. a property) with degree of belief that equals one.

When it comes to the reconstruction set from a Timeprint, we are sure that the trace-cycle which actually happened, denoted by tc^a, belongs to this set; i.e. the degree of belief that one of the traces in the reconstructed set $tc^a \in R$, happened is $Bel(tc^a \in R) = 1$. And we are sure that a trace which does not belong to the reconstruction set did not happen; i.e. probability that some trace $tc^x \notin R$ happened is zero: i.e. $Bel(tc^x \notin R) = 0$. From the Timeprints alone, we have no more information than this. However, among solutions/elements of the reconstruction set, one can assign belief functions explicitly based on assumptions and other available data, until Timeprints can prove or refute, that some property P is satisfied. This shall be seen later in this chapter.

In this section, we first explain how reconstruction sets are mapped to artifacts of theory of evidence. Then, we show how evidence of Timeprints regarding a property P can be formally proved.

7.3.1 Frames of Discernment

Within the theory of evidence, to draw conclusion from some facts and beliefs, a *Frame* over which such conclusions can be made has to be first defined. Such frame expresses the set containing all possibilities. Weights of evidence can be then assigned over these *frames of discernment*. In the theory, a frame of discernment contains the set of all possible implications of some piece of evidence. This frame perfectly mirrors our reconstruction set R or more precisely $R_o(tp_i)$, which expresses all possible values of the change-signal tc, over some trace-cycle i, denoted by tc_i, which can result in the logged Timeprint tp_i. From this set, we know that one trace has actually happened, but we do not know which one.

We shall consider that any property can be represented by all traces fulfilling it. Over trace-cycles, the property holding on the full trace can be represented by sets of segments, where each segment corresponds to a trace-cycle. Given this, we focus in this Chapter on one trace-cycle reconstruction sets.

The reconstruction-set R_o resulting from any specific Timeprint tp_i, where $|tp_i| = b < m$ is a proper sub-set of $\mathbb{F}_2^m$, which represents the domain of all trace-cycles tc_i, where i is enumeration from 1 to 2^m, and m is the size of all such possible traces. We can write $R_o \subset \mathbb{F}_2^m$.

When there is no other information at all, except for the Timeprint, we still know there is only one actual trace-cycle tc^a which actually happened, and that

$$tc^a \in R_o \tag{7.3}$$

From this, for any set A, the belief function which expresses evidence of the Timeprint tp in proving any set A can be written as:

$$\forall A \subset \mathbb{F}_2^m, Bel(A) = \begin{cases} 0, & A = \phi \\ 1, & A = R_o \\ 0, & A \subset R_o \end{cases} \tag{7.4}$$

which clearly shows that we are sure that one of the set R_o has happened (because of the logged Timeprint), but there is no evidence at all from the Timeprint alone that favors any of the traces in this set over the other.

When the set R_o has more than one trace, it is not possible to know which one has happened. We can, however, use properties known to hold to filter them. We assume we can formulate the property P_i^* for the current i^{th} trace-cycle from the global property P. P_i^* represents here the property we want to check whether it is satisfied or violated over the trace-cycle tc_i.

This means, for a Timeprint to act as *Evidence* that some property P holds, over the trace-cycle tc_i the condition is:

$$\overline{P_i^*} \cap R_o = \phi \tag{7.5}$$

Where $\overline{P_i^*}$ is the complement of P_i^*. This means evidence can be expressed as inclusion problem, and a Timeprint is evident if its reconstruction-set is all included in P. Hence, the evidence condition can also be expressed as:

$$R_o \subseteq P_i^* \tag{7.6}$$

Remember that the property P_i^* which we want to check, is different from assumed properties, which we consider as input, based on which we obtained the updated reconstruction set R (where $R \subset R_o$). If we have such assumed properties, which we are sure they hold (proved by RV monitors for example), we can us that to make the evidence condition lighter. By this we mean instead of requiring that all solutions/elements in the reconstruction set R_o to be included in P_i^*, it suffices that the elements/solutions in R are all in P_i^*. In this case, where a subset R of the reconstruction set is obtained, the evidence condition can be written as:

$$R \subseteq P_i^* \tag{7.7}$$

The more and stronger the assumptions are, the smaller the reconstruction set size will be, and hence the higher the probability it can be evident. This will be clarified more in the case-studies. In our evidence-oriented tracing, an assumed property represents a fact, or a widely accepted, or formally verified assumption. To distinguish between these two types of properties, we call the property we want to check, the *Objective Property OP*, and use P_O to denote it. We then call the property we assume holding and hence use it in the reconstruction, the *Assumed Property AP* and use P_A to denote it. Hence, in the following, properties P_i^* defined over trace-cycles, shall be either P_A or P_O.

Definition 4 *A trace-cycle's Timeprint (tp) is a sufficient evidence on its own that an objective property P_O holds,* ***iff*** $R_o(tp) \subseteq P_O$.

If the set R_o contains a lot of solutions, some assumed property(ies) P_A would be needed. While the reconstruction set of a Timeprint tp is denoted as $R_o(tp)$, the intersection between $R_o(tp)$ and the set containing all traces fulfilling an assumed property P_A can be written as:

$$P_A \cap R_o(tp) = R(tp, P_A) \tag{7.8}$$

If no properties are assumed $P_A = \phi$ then $R = R_o$. We define the evidence with an assumed property as follows:

Lemma 5 *A trace-cycle's Timeprint tp is evident that an objective property P_O holds,* ***iff*** *$R(tp, P_A) \subseteq P_O$, where P_A is some assumed property, proved to hold in reality.*

Notice that here, $R(tp, P_A)$ is the new frame of discernment, and that is why when all of $R(tp, P_A)$ is included in P_O, it suffices to prove that the property P_O is satisfied.

Proof Sketch Lemma 5 follows directly from the previous definitions. However, we give an explicit sketch here.

Proof. Over a trace-cycle, all possible traces set T are devided into two sets, by a property. A set P_O in which all traces satisfy the property p_o, which is described by all traces satisfying it; i.e. P_O. A reconstruction set of a Timeprint, $R_o(tp)$, is the set of all possible traces that might lead to the logged Timeprint tp. We are sure that the trace which has actually taken place is a member in this set. Moreover, as we know that P_A holds, then the trace which actually happened named tc^a, is in $R(tp, P_A)$. Then if $R(tp, P_A) \subseteq P_O$, then $tc^a \in P_O$. This proves the forward path (if). For the other direction, we use contradiction. Assume $R(tp, P_A) \subseteq P_O$ and $tc^a \notin P_O$. This is not possible because by the definition of the Timeprint, $tc^a \in R_o(tp)$, and as P_A holds, $tc^a \in R(tp, P_A)$, hence, having $R(tp, P_A) \subseteq P_O$ contradicts with $tc^a \notin P_O$. □

Notice that from equation 7.4, R or in $R(tp, P_A)$ are considered the new frames of discernment, and the subset A can be seen as the part of $P_O \cap R$ or in $P_O \cap R(tp, P_A)$ when P_A is known to hold. This means the belief function of this set that is known to hold is 1.

7.3.2 Proving Evidence of Timeprints

As mentioned in the introduction, beside the Timeprint itself, we need a proof that the Timeprint is evident. We can now state: regarding whether a property P_O is

holding, a Timeprint is evident, when it is enough (together with acceptable/provable assumptions P_A) to refute or prove P_O. This can be formalized as:

Evidence of Timeprint ($Ev_i(tp)$)	
Input:	Timeprint tp_i of a trace-cycle tc_i, a Time-encoding TE, an objective property P_O and assumptions P_A.
Task:	Decide whether tp_i is evident or not, that P_O holds over tc_i.

The solution to this problem can be written as:

$$if \quad R(tp, P_A) \subseteq P_O \quad then \quad tp \quad proves \quad P_O \tag{7.9}$$

and

$$if \quad R(tp, P_A) \subseteq \overline{P_O} \quad then \quad tp \quad refutes \quad P_O \tag{7.10}$$

where P_A can be obtained from Run-time Verification checks, and tp is obtained from the Timeprints logging procedure. When either 7.9 or 7.10 is true, the problem is solved: tp is evident and hence, evidence proves P or refutes it. Otherwise, if neither equation 7.9 nor 7.10 is true, then tp is not evident and cannot be used as a proof for the property P_O.

7.3.3 Coverage of Evidence

We call the extent to which Timeprints can provide evidence *Coverage*.

We also want to ensure –before deployment– that for some predefined time encoding TE, some property P_O can always be proved or refuted. This means we want to know whether TE can be *Evident* regarding a property P_O. This problem can be defined as:

Coverage of Evidence of Timeprints using Encoding ($CovEv(TE)$)	
Input:	a Time-encoding TE, an objective property P_O and assumptions P_A.
Task:	Decide whether all Timeprints resulting from TE are evident in proving or refuting an objective properties P_O.

To solve problem 7.3.3.$CovEv(TE)$, we need to prove that for every possible reconstruction set, that all solutions contained in it, belong to the sets defined by the P_O. For each encoding TE, there is a fixed kernel which defines a prior set of safe-properties P_S. However, this set is only defined by the kernel of TE, and it does not express a meaningful property. This makes the ability of a TE to prove or refute any

objective property P_O, highly dependent on how the assumed properties P_A are defined. At post-mortem posterior P_A's can be added as reasonable and needed, until reaching a satisfying proof.

Notice that equations 7.9 and 7.10 are sufficient conditions to consider a Timeprint tp as evidence. However, if they are not satisfied it does not mean the P_O does not hold, but rather that tp cannot prove whether it holds or not.

7.3.4 Formal Evidence Proof

Before execution, i.e. for all possible Timeprints, proofs of coverage can be carried on for certain P_O, given certain P_A.

After execution, given the logged Timeprint(s), it might still be possible to carry on a formal proof of the logged Timeprint's evidence; even if that property was not originally covered. However, in this case, this is not guaranteed to be always possible.

When the proofs are extended for all possible Timeprints, the coverage is guaranteed for all cases, before the system's deployment. However, it is not always possible to reach this level for any TE regarding some objective property; and adding assumed properties to utilize, until the condition in Lemma 5 is reached is not always easy, as shall be seen within case-studies later in Chapter 9. Hence, if formal proofs of coverage cannot be reached, statistical proofs can still help; as explained in the following section.

7.3.5 Statistical Coverage

If formal coverage proof for an encoding, regarding its evidence of some property, is not available, we might still be able to have some Timeprints as evidence. To be able to judge whether it is reasonable to use some encoding, even if the proof is missing, statistical coverage can be used. To obtain statistics, evidence of Timeprints for specific instances (which are expected to happen) can be used and their evidence checked. These instances can be generated by simulation for example. In Chapter 9, a case study for coverage of delays on CAN messages is presented.

Next, we show how to express delay as a property as was just shown here, and how to judge whether some Timeprints TE can be proved to capture delays.

7.4 Delay Evidence

In this section we apply previous concepts to delays as a property. We first show how delay of a binary stream can be expressed. Then we show an example applying how we defined delay to a trace-cycle signal. Last but not least we show how Timeprints can be evident in expressing such delays.

7.4.1 Delay as a Property

The property under inspection, is the one we want to check whether it holds or not over a trace. Delay as a property under inspection, shall be defined here in that sense as follows. Given a signal s, which has a trace t, we want to check that some N^{th} occurrence of change in the signal, happened before some time D. Such N^{th} change, would happen to be in some x^{th} trace-cycle tc_x.

Within the tc_x, where the trace-cycle is a segment of the trace $tc_x \sqsubset t_s$, the change which we want to check whether it has missed its deadline D, has the order N, i.e. we want to check that the N^{th} change in signal t_s, happened before D^{th} clock. Where $D \in I_x$, and $I_x = t[mx + 1, ..., mx + m]$.

Let's denote the deadline position inside the trace-cycle x by d. This gives $d = D - mx$ and $1 \leq d \leq m$. And let's denote the order of change within the trace-cycle tc_x by n. The Timeprint TP_x of the trace-cycle x has the associated number of changes n_x. And n, the order of change within the trace-cycle x can be written as:

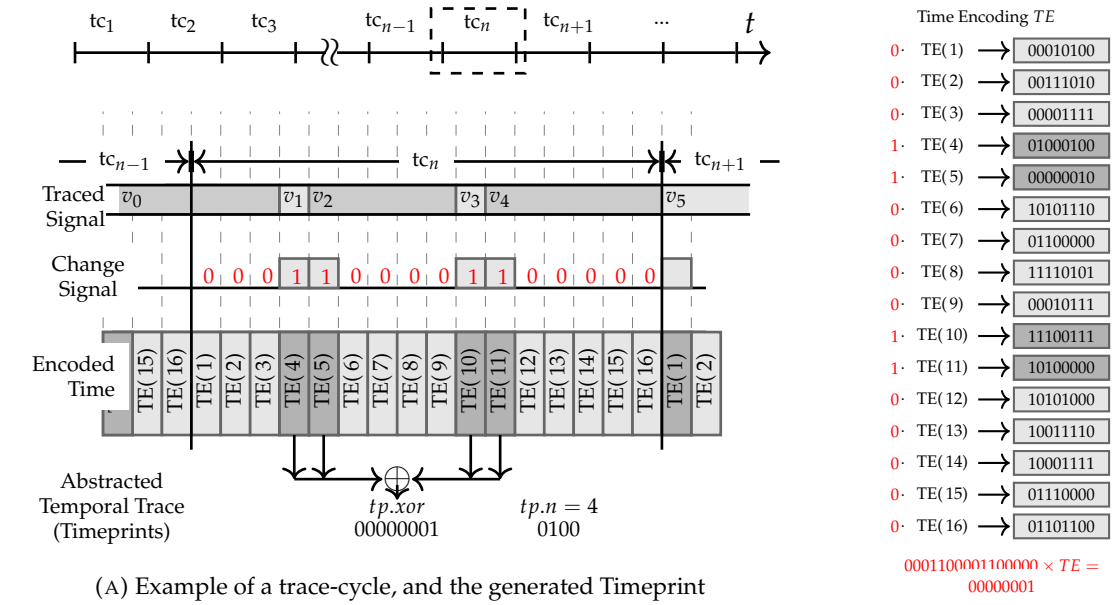
$$n = N - \Sigma_{i=1}^{x-1} n_i \tag{7.11}$$

Where $\Sigma_{i=1}^{x-1} n_i$ represents the sum of all changes in all previous trace-cycles. If $n \leq 0$, it means the n^{th} change already happened before the trace-cycle x, implying the signal came on time. If $n > n_x$, then the deadline d was definitely missed. Else, we are left to check whether the n^{th} change inside tc_x, happened before d.

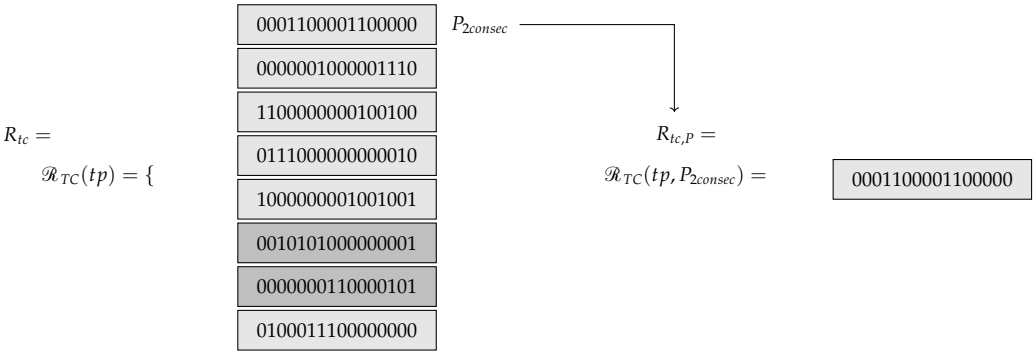
Checking this can be done by the reconstruction of the x^{th} trace-cycle. According to Definition 4, the reconstruction set need to be a subset of the set of all possible traces satisfying that the n^{th} change happens before d. And because the reconstruction sets are not that limited, Lemma 5's assumed properties P_A are needed.

7.4.2 Evidence of Delay with Assumed Properties

To capture delays experienced by a signal, we formulate the signal's characteristics that we know by assumed properties P_A, as needed.



(A) Example of a trace-cycle, and the generated Timeprint



(B) Reconstruction from Timeprints

FIGURE 7.6: Timeprints Generation and Reconstruction

For instance, in our running example in Fig 7.6, if the first change is the one we want to check whether it has met a deadline d, and the deadline is at $d = 8$, already all traces in R_o, depicted in Fig. 7.6.(b), fulfill this condition, i.e. $R_o \subseteq P_O$ and no need to define any P_A. Here, TP is enough to prove that the deadline at $d = 8$ was met, irrespective of which trace is the one that actually occurred.

But if we want to check if we can have evidence of whether $d = 7$ is met, the inclusion will fail, as many traces in R_o have the first change after the 7^{th} clock cycle. Here, we need to define some assumed property P_A, which we know it holds for the traced signal, until we can get the inclusion working or to get one solution. If we use the property that the pattern 101 does not occur; i.e. $\forall(1 < i < m-2), (x_i = 1) \implies \neg(x_{i+1} = 0 \wedge x_{i+2} = 1)$, then we get the shaded solutions in Fig. 7.6 excluded, and now the reconstruction set is contained in the set of all traces which has the first change happening before $d = 7$.

If we use the property $P_A = P_{2consec}$, described in the figure, we also get R_P which contains only one solution. But moreover, this reconstruction set has only one solution. And whenever a reconstruction set R contains one one solution, it becomes evident for any delay d. i.e. is able in all cases to prove/refute the delay because it has definite one time of occurrence.

Notice that all these proofs are posterior, i.e. after we have the timeprint tp, and hence, we know its reconstruction. To have a priori proof about some property, we define the property's *coverage* in the next subsection.

7.4.3 Obtaining a Priori Proofs for Coverage of Delays

We have shown so far how one delay d can be proved, if it occurred. But we do not know before the execution starts at which instance d is going to happen, inside a trace-cycle. So we still need to check and prove before deployment, that all possible delays over a trace-cycle are fully *covered*, by certain time-encoding TE. Of course a TE which uses hot encoding, i.e. simply log one or zero every clock-cycle, can prove any delay. But because we want to log at the end of the trace-cycle a much smaller number of bits (only the Timeprint), we lose some information, and we don't want our capability of detecting delays to be affected because of that.

For any given Timeprint, if the reconstruction contains one solution, it is evident (without any further requirement) in proving/refuting any temporal property, not only the occurrence of any delay d. The reason is that any such reconstruction gives one exact timing for all change instances, from which it is straight forward to say whether there is delay in some n^{th} change or not. This is similar to the case shown in the motivation example, where a detailed trace can help proving what happened exactly. If the reconstruction-set contains more than one solution, then they all need to contain the n^{th} change before d, for all possible delays d and for all possible n. As this is very hard to describe, as shown in the next paragraph, we opt (when assumed properties are not enough) to use the uniqueness of solution in the reconstruction set.

Let P_O express the objective property: that for all $n < d$ and for all $0 < d \leq m$ the n^{th} change happened before delay d. i.e. that all the signals fulfilling an assumption P_A, and their n^{th} change happened before the delay d results in different tp from those happening after it. In other words, any reconstruction set for a TE and P_A, (i.e. all solutions in any Timeprint's reconstruction-set) have to be evident. And for a reconstruction to be evident, all solutions R_{P_A} have to be subsets of P_O for every delay d, where $1 \leq d \leq m$ and this is all done for every n.

Alternatively, we can use the uniqueness of solutions in the reconstruction set, as mentioned before. The property expressing delay coverage (that all possible delays are covered) can be written as:

$$\forall tp_k, |R(tp_k, P_k^*)| = 1 \tag{7.12}$$

Where tp_k expresses all possible Timeprints that can result from any trace-cycle that fulfills an assumed property, which is known to hold over signals for which we want the Timeprint tp_k to be evident.

Notice that this definition implies that for every encoding TE, one can already obtain a formulation for the set of all properties which can be covered by that encoding, let us call this set $Prop$. Hence, when we want to check whether a property P can be covered by some TE, we can see if it is a subset of $Prop$.

7.5 Summary

In safety critical systems, design abstractions are usually verified [13], and some verification mechanisms are even extended to implementations as run-time checks [21]. In the upper part of Fig. 7.7, the arrow expresses the logical design contribution, to the physical implementation and visible failure symptoms. The traces available for analysis when something goes wrong, are very coarse compared to the complexity and speed of these systems. When it comes to full autonomy, there is going to be no human in the operation loop to blame, or to rely on to timely react in these cases. The designers and manufacturers shall be forced to carry the responsibility of the resultant failures. Hence, accurate and transparent tracing of signals and their timing would then become more tangible in the process of liability assignment within systems composed of parts from different designers/manufacturers.

Reaching such accurate and evident tracing is technically challenging. The first challenge is to capture/store/process the resultant huge amounts of data efficiently, while being completely non-intrusive and transparent, so as not only to have zero effect on the traced signals, but also to provide clear systematic logs that enables recovering execution details. The second challenge, is to have the efficient handling (which is expected to rely on abstractions), proved to cover the problems (here delays) expected to occur.

Timeprints address the first challenge as light weight logged abstractions, triggered by the temporal behavior of signals. In Fig. 7.7, an arrow expresses that the effect of failure visible symptoms shall appear in the generated Timeprints. It is possible to retrieve some cycle-accurate details, and/or check some temporal behavior

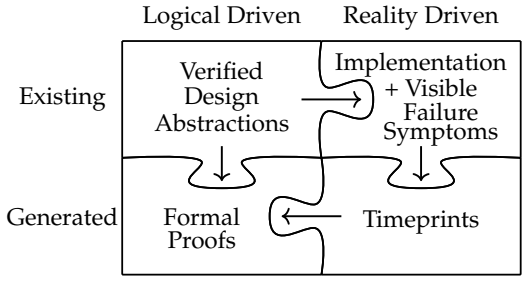


FIGURE 7.7: The Role of Formal Proofs in Supporting Evidence of Timeprints

properties using them. From the second challenge perspective, guarantees about which properties they can cover were presented in this Chapter.

We borrowed the concept of *frames of discernment* from the theory of evidence [151], and mapped it to the trace-cycles reconstruction sets. Over these frames, we define patterns and properties (like delays), in terms of sets, resulting from assumptions (assumed properties). Then, we checked whether for a given time-encoding, it can cover the formulated delays or not. We also present a definition of *Delay-Coverage*. A proof of delay-coverage of certain pattern, constitutes the forth corner stone to be able to use Timeprints as evidence; as in the last piece bottom-left in Fig. 7.7. The idea is: if this proof is present, all the logged Timeprints will be definitive in deciding whether a signal is delayed or not; hence, a formally proved evidence.

This chapter addressed briefly the evidence of Timeprints, its formulation and coverage. The chapter also illustrated these concepts specifically for the delays. However, to start extracting evidence in practical contexts from Timeprints, the context of tracing and a framework for verifying evidence is still needed. This is the topic of the next chapter.

Chapter 8

The Big Picture: Putting Pieces Together

In the previous chapters, the foundational concepts behind Timeprints were explained. Here, we bring them together in a framework, which takes signals' traces as input, and enables verifying their behavior and properties in *accurate and evident* way. This chapter also crystallizes and visualizes the usage of Timeprints, from a bird view perspective. This comes here in order, after the different deep dives in the various formal aspects of the Timeprints over one trace-cycle, showing how that extends naturally to all signals. We also show how some tasks which used the traditional signals' traces and verification techniques can be carried on using Timeprints. Hence, this chapter connects the dots and ties the knots, bringing that framework to life.

The chapter starts with highlighting the Timeprints traces continuity, transparency and size consistency, as distinguishing features of Timeprints. After that, we present the context of using Timeprints effectively in practice. We provide first a verification framework, which shows how systems tracing, I/O relations checking over windows-of-interest can be done via reconstruction.

This leads directly to the next part, which discusses the generalization and the limitations of the presented approaches. Then, the chapter discusses briefly how multiple points/signals, with even possibly independent clocks, may utilize Timeprints. This chapter acts as a conjunction, or bridge, between previous theoretical concepts in the previous four chapters, and the next practical chapter *Applications*, which will present the conducted experiments.

8.1 Tracing and Verification with Timeprints

Major distinctive advantages of Timeprints are their transparency, consistency and continuity. Transparency implies open, automated and well-defined generation method, as well as its respective open hardware and software implementations; which are fully independent from the systems being traced/verified, and fully non-intrusive at the same time.

Consistency implies that all over the execution, for all the back-to-back trace-cycles, exactly one Timeprint of fixed length is logged. Hence, the number and size of Timeprints logged are linearly proportional to the execution-time. The inter trace-cycle continuum is supported by the small size of the timeprints, w.r.t. the trace-cycle length. This makes logging the Timeprints continuously possible. The speed of logging is required to guarantee all Timeprints are transmitted within the trace-cycle. To insure this, run-time verification can be straight forwardly used to check that no Timeprint is missed. Another check is concerned of checking that any change in the traced signal leads to a different Timeprint, and that there is always a Timeprint sent within every trace-cycle interval.

Traces logging continuity is a fundamentally missing feature in many existing tracing methods. Many tracing methodologies focus on efficiency; hence, in many cases only new or important values are logged, either alone or with an associated Timestamp. So whenever there are no new or important values there might be no data to log, or when a lot of changes happen, the data size becomes huge. Either way, the log size becomes dependent on the signal's rat of change. Notice that accurate Timestamps increase the log size significantly.

To avoid such a burden, tracing temporal behavior's partial order, as introduced by Lamport [95], focuses on ordering the events, which have inter-dependencies, instead of logging absolute Timestamps which depend on clocks that will suffer anyway from jitters and skews. Total order can also be supported on Lamport's proposal using a system of clocks, within which it is acceptable to have some events happening in the same logical time. The *Timestamp* in that sense serve as a label, from which the order (or the concurrent events) can be deduced. A combination of both total and partial order can be used to analyze distributed systems, where not a very accurate clock is expected to be used, and events which require ordering should ensure taking place over more than one clock period; of that course clock.

Comparing this to Timeprints, which also depend on clocks, as in total-ordering,

the Timeprints' clocks can be much more accurate (smaller clock-cycle) and independent. This is enabled by that fact that: obtaining the actual log happens automatically, synchronization and inspections happen later, after the logging is already done. Timeprints hence allow more accuracy and fixed size logs. Timeprints do not aim at replacing such ordinary Timestamps. They rather amend their dependence on self-reporting, and extend the tracing ability to obtain accurate evidence about exact timing so that the self reported data can be checked.

When it comes to systems like CPS, where the interaction with the environment takes place over time, even for a single system or signal, the partial or even total order is not enough for expressing the amount of time. Time-lapses between related events, and their respective specification, becomes relevant. Metric [90] and interval [9] logic, as mentioned in Chapter 2, considered more this amount of time.

Timeprints as Heartbeats Timeprints, with their fixed number of bits being logged/transmitted at the end of each trace-cycle, act as systems liveliness heartbeat. Similarly, they become constant if the system's signals are silent. From this perspective, they do not only serve as a way to check properties, or to provide evidence, but they also are direct indication of liveness.

As they can be used for tracing any signal, they can be used for tracing software execution on hardware as well; as shall be seen later in Chapter 9. Because they work at the hardware level, the Timeprint can be generated by changes in the program counter value. Timeprints can be also generated for other important signals in embedded systems such as interrupts or input/output devices. Timeprints can in this sense also be used in checking operating systems integrity, at the level of hardware; for example can be integrated with solutions like [64], which proposes light weight extensions for checking operating systems integrity using hardware.

In the next section, we address the general case, where properties are described, as classically by temporal logic, on a whole trace of execution, rather than trace-cycles as what has been the focus so far.

8.2 From Global Properties to Trace-Cycles

Temporal properties are classically defined by various types of temporal logics.

Whatever form/logic the model/trace/specs are described with, it can be translated into all the possible execution traces fulfilling it. However, such traces do not always contain explicit mention of time. Moreover, describing a property –that is defined globally over a full trace- over a trace-cycle is not an easy task. Intervals,

dense and discrete time annotations have been used in Interval Logic [9], Metric Temporal Logic [90], and its discretised version [108]. In all these logics, when we talk about one trace of execution, that one full trace under inspection can be itself described by listing all possible full traces (with respect to a discrete clock) fulfilling the description of that one trace.

This listing/set, or an equivalent action that involves inspecting all intervals of one such trace, is usually needed when the system is interacting with external actors which will act on their own timing, and their actions can affect the trace of execution.

A property which is defined all over the system can fall between these two categories: 1) the property is defined over a system's full trace of execution, or 2) a property that is expected to hold locally, over one or two signals, within a finite time interval and only when some other timed preconditions already hold. For the first category, i.e. the property is defined over one full system's trace of execution: we need to project it first on the traced signals, then for each signal, understanding the interval over which the property should hold, and then deal with it as a local property. For the second category, namely local properties defined relative to some timed event, these can be projected first to windows of interest, where the initial conditions can be defined, then the window of interest can be itself projected on trace-cycles. Having windows of interest (WoI) is the first step to map the property to the trace-cycles as shall be seen later. WoI helps utilizing the strength of Timeprints, by focusing the reconstruction on a small number of trace-cycles.

A global property, described in propositional LTL for example, is usually expected to hold all over the trace of execution, and not only within one trace-cycle. To project a temporal property on trace-cycles, a possible theoretical method is by using Hyperproperties [51] as shall be seen shortly. This can be used generally to address the first category. When having a window of interest, defined by an explicit moment in time, we can go upwards towards fitting the local check into the generic context of the respective global property.

For global properties, the projection can be done by expressing the global-property by listing all possible infinite traces fulfilling it. As tracing in our case is finite, methods for padding the finite traces into infinite one, as done in [51], can be used for the global properties projection. Other methodologies and techniques (as the examples in [26, 145]) were used to address cases where properties cannot be decided until reaching the full trace. These are not relevant because in Evidence-Oriented Verification, the execution has already taken place, and what we are trying to prove is something that already has taken place in the past.

The projection of Global properties over trace-cycles, makes the reconstruction

sets become sets of *sets of traces*. Properties about sets of sets of traces are called in the literature *Hyperproperties*.

8.2.1 Hyperproperties

Any property, which one might want to determine whether it holds on a specific execution trace or not, can be expressed in terms of all traces fulfilling that property. Such set of traces is just one possible way for describing a property. Clearly, using this description, checking weather a property holds over a trace (is satisfied by an execution expressed by that trace), results in a Boolean value (it either holds or not). Such check, when applied, leads to *judgment* or a decision, of whether the property holds or not.

When it comes to addressing properties, whose satisfaction can only be described over **sets of** *sets of traces*, then we are talking about *Hyperproperties*. A very clear example, which actually was associated with its first word-coining, is non-interference properties. These are a class of properties which denotes that: a specific set of traces satisfying some original property, do not interfere [1] with another set, which satisfy another undesirable property (or violates the original properties). It is clear that for something like interference, we are talking about the relations between sets of properties (sets of sets of traces).

Here, a brief overview of the formulation of Hyperproperties is given. Then in the next section, we list the basic assumptions needed to start formalizing delays as Hyperproperties over trace-cycles.

Properties and Trace-Cycles Properties can be represented by the set of all traces fulfilling them P. A trace is a sequence of states. A state can correspond to either analog or digital measurement of the traced signal. In our set-up, each measurement is considered a separate state, even if it is not different from the previous measurement. A *change-trace* then is a sequence of states belonging to either *change, no-change*.

In our temporal change-oriented tracing, we are tracing (measuring) every clock-cycle, whether there is a change or not. When we call a change-trace t, a change-trace-segment t' is written as:

$$t'_{i-j} \triangleq t[i...j] \tag{8.1}$$

indicating that it is a sub-trace of t, starting from its i^{th} state, until its j^{th} state. And t is a finite trace which, in the general case, starts at 0, and ends at e, where e is denoting the current time instance or the moment of last available data for inspection.

[1] Interference can be expressed by their fulfillment of another secondary property

For a change-trace $t \in T$, where T is a set of change-traces, if P is the set of change-traces fulfilling a property p, then:

$$T \subseteq P \implies T \models P \implies \forall t \in T, t \models p \tag{8.2}$$

Over a trace-cycle, tc_n is the m-long trace segment, describing the signal change state over the n^{th} trace-cycle. This can be written as:

$$tc_n \triangleq t[(m * n) - m + 1)...(m * n)] \tag{8.3}$$

and a change-trace can be written as:

$$t \triangleq tc_1.tc_2.\ldots.tc_n\ldots.tc_l \tag{8.4}$$

where $l = \lceil e/m \rceil$, is used to denote the *last* trace-cycle, and it is $\lceil\rceil$ and not $\lfloor\rfloor$ because the last trace-cycle, even if it was not complete, can be padded with zeros till the end of that trace-cycle; to count for the last no-change instances. [2]

To check the satisfaction of a property P, over a trace $t = tc_0.tc_1.....tc_n.tc_{n+1}....tc_{last}$, it means we want to check whether $t \models p$, where $p \in P$. Alternatively, one can solve the inclusion problem, of T, where $t \in T$ and we ask whether $T \subseteq P$. While answering this inclusion problem is straight forward when asked over the full-traces-set T, it is not the same if we are only focusing on some specific trace-cycle t_{tc}, within an exact one trace t, which had actually taken place in reality. We are now lifting the problem on two steps:

1. the inclusion problem becomes: is the specific trace which has taken place, $t \in P$? The problem here is, we still do not have any information which t has actually happened.

2. what we actually have about t is the Timeprints trace of it, which is an abstraction, defined over trace-cycles.

Given these two steps, the property P has to be redefined over the trace-cycles into P^*, where P^* is a Hyperproperty, defined to hold over any arbitrary trace-cycle tc_i, which is a part of the actual trace t that happened, $t = tc_1.tc_2.\ldots.tc_i.\ldots.tc_l$, and produced Timeprints $TP_j, k_j, 0 < j < l$ [3] as follows:

[2] In post-mortem, when the system stops, it means no change, that's why padding of zeros at the last non-complete trace-cycle makes sense.

[3] Notice that whenever we have a logged Timeprint TP_j, k_j, we have already at least one clock cycle in the next trace-cycle tc_{j+1}, and even if there is no other clock-cycles, this one-clock cycle will constitute the last trace-cycle, padding it with zeros.

To check the satisfaction of a property P, over a trace t, means we want to check whether $t \in P$. While this check is straight forward when asked over a full-traces t, it is not the same if all of what is available are Timeprints, rather than the full original trace t_o. The problem becomes: is the specific set of traces, defined by the aggregated reconstructions $R(TP_i), \forall i$, of the logged Timeprints included in P? Aggregated reconstructions from Timeprints contains a lot of candidate traces, from which only one is the one which actually took place. This variant is an inclusion problem, which asks: *is the set which contains all such candidate traces T, (where T is an over approximation) is included in P?*

The Timeprints trace tp of t_o which has happened, is an abstraction, defined by Timeprints abstractions over trace-cycles as:

$$tp = tp_1.tp_2.\ldots.tp_i\ldots tp_l \tag{8.5}$$

where TP_i is composed of the two parts: $TP_i(tc_i).xor$ and $TP_i(tc_i).n$, defined as in 4.2. This same tp can result from other trace candidates $t \in T$. The property P, will have to be redefined piece-wise over the trace-cycles, to enable using the sets of reconstructed traces in solving the inclusion problem. As mentioned before, a property can be fully defined, by the set of all traces satisfying it. Properties over trace-cycles, can be represented by sets (traces satisfying the full trace property) of sets (reconstruction-sets of the Timeprints, logged at the end of each trace-cycle). Properties described over sets of sets of traces are known in the literature as *Hyperproperties* [51].

We define P^* as a Hyperproperty, where $P^*(i) = P_i^*$ is a set defining that P holds over the trace-cycle tc_i, which is a part of the some candidate trace $t \in T$which produces Timeprints trace tp. By t_o, the actual trace, also the Timeprints trace tp is produced. Both t and t_o are in T, and both are variables, which can be considered equivalent when t_o is not known. But we will use t to indicate the list of all candidate traces, and t_o to refer to the unknown original trace. P^* is defined as follows:

$$P^* \triangleq \{T \in Prop | \forall i, tc_i \sqsubset t \wedge t \in P \implies \forall j, tc_j \in R(tp_j, P_j^*)\} \tag{8.6}$$

This Hyperproperty describes how the trace property P which is defined over the full trace t, is defined over every trace-cycle tc_i of it. P_i^* describes not only the set of tc_i candidates in $R(tp_i, P_i^*)$, which contains all possible change-trace reconstructions in trace-cycle i. But also describes for all other trace-cycles $(\forall j \neq i)$, the implications of each element in $R(tp_i, P_i^*)$. The reconstruction sets of all logged timeprints tp_j, each contain candidate segments (trace-cycles) of the same trace t

$\forall j, 0 < j < l, \exists tc_j \in R_o(tp_j) \wedge tc_j \sqsubset t$. Notice that not all elements in the reconstruction sets are real; however, even all elements can have implications on adjacent trace-cycles. This constitutes lists of possible full traces marked with implications source trace-cycle within reconstruction sets. Notice also that these implications can have self-reflections on the source trace-cycle updating the valid elements of the reconstruction set. Instead of such complex lists of implications, using Hyperproperties makes just the description easier. However, Hyperproperties are known to be very daunting to check [66].

The satisfaction of this Hyperproperty means: the trace's Timeprints trace tp, when reconstructed, intersects with the traces set P in a non-empty subset which contains t_o. Although it is not defined which one exact trace in this subset is t_o, which actually happened, it is guaranteed (with fulfillment of $P_i^*, \forall i$) that all of them, would fulfill P, because all of them do. Note: we use R_o, R to describe $R_o(tp_i)$, $R(tp_i, P_i^*)$ when clear from the context. We also overload the use of R to refer to any reconstruction set in general, if no specific trace-cycle is specified.

Alternatively, as checks in EOV are done in postmortem to check specific intervals, the reconstruction can be done over Windows-of-Interest (WoI) instead of jumping to check Hyperproperties.

8.2.2 Windows of Interest

For a window of interest, WoI, corresponding to an interval I, over which we want to check whether P holds: we first need to define the duration of such window/interval $|I|$ w.r.t. the property length $|p|$, and then map this to trace-cycles within the WoI.

Depending on the relations between trace-cycle-length $TCL = m$, the interval WoI length $|I|$, and property length $|p|$, we can distinguish between the colored cases in Figure 8.1, these different cases are shown, each with a description.

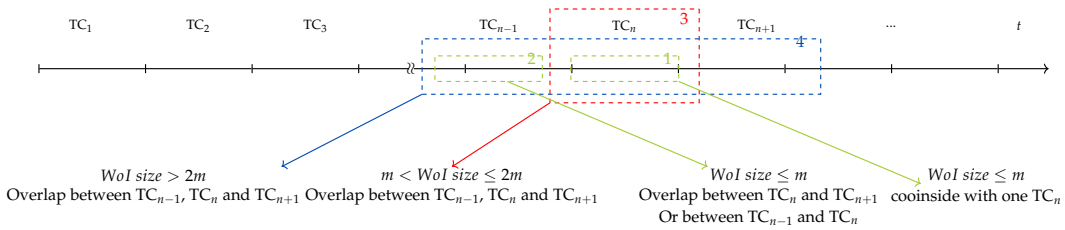


FIGURE 8.1: Window of Interest (WoI) and mapping to Trace-Cycles (TC)

As each single trace-cycle, has implications on the previous and next trace-cycles. These implications can be chained as well, see 6.3. For formalities of the classical

Hyperproperties to hold, a WoI contains an integer number of trace-cycles, hence the implications are carried on as illustrated in 6.3.

Verifying that global properties hold on only a segment like the WoI is not straight forward. A trace-segment may verify or violate the global property based on the rest of the trace outside this segment. Hence, the status of "Satisfaction is possible" as well as "Violation is possible" are among the SAT-check results. It is worth mentioning that the way we presented the implications so far is not complete. Moreover, its soundness depends on the correctness of listing all possible traces satisfying a property. We are considering now defining a logic that enable checking the properties on trace-cycles and relate them to global temporal logic.

8.3 Evidence in Context

Evidence shall be required after some incidence or failure happen, and trails of execution are consulted to check what happened. Figure 8.2 shows elements used to obtain evidence.

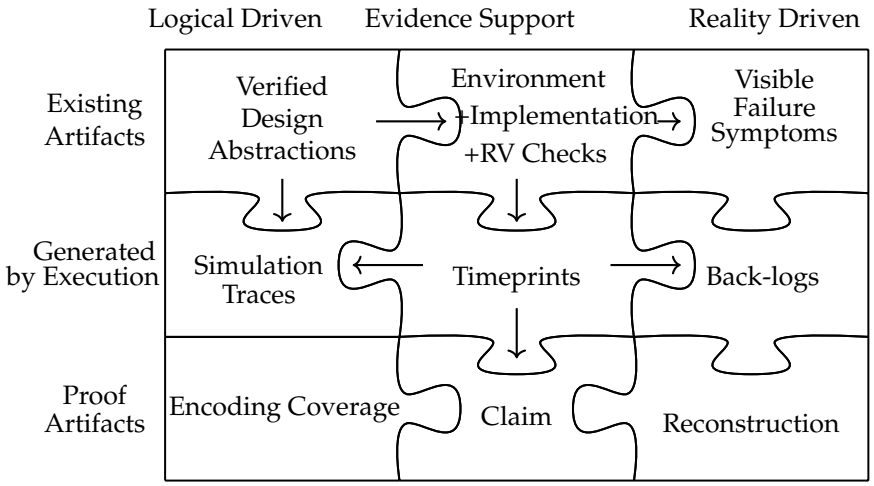


FIGURE 8.2: Evidence Components Inter-winding

The Figure start from the upper left corner. The system design artifacts are used to actually manufacture the system. They are also consulted later when there is a problem, either to be checked themselves for possible problems or to generate simulation traces that help pruning the reconstruction space. They also provide RV monitors to conduct checks at run-time; the results of those checks can also be used in the reconstruction if they are logged. The designed RV checks also can help choosing time-codes that help generating Timeprints that are capable of covering/revealing more problems.

Relation to Time-Encoding Clearly, reconstruction depends on the time-encoding set which is used. If the pattern occurring already belongs to the safe-properties of some time encoding *TE*, (see Chapter 5) the reconstruction shall never end in the state of having elements belonging to P and other in $\neg P$, and the timeprint shall function perfectly as evidence of whether the signal occurred on time, or not. One measure of a good time-encoding, is related to coverage of properties. The capability to provide evidence for some class of predefined properties, like the delays for instance, is essential. This leads us to the discussion of the next subsection which discusses what is needed so that Timeprints can act as evidence.

8.3.1 Evidence Context Pillars

To obtain evident traces Timeprints alone are not enough. We discuss here briefly what is needed as a basis to ensure the evidence of Timeprints.

- The context history has to be defined. This includes which signals are traced and which ones have Timeprints, the time of the last available Timeprints, any relevant RV-Check traces of failure visible symptoms, and formulated assumed as well as objective properties. If there is software involved for example, the exact version
- The Timeprints syntactical relevant inputs, i.e. the exact generating function, time-codes used in generating the Timeprints. Notice that in the general case time-codes might be designed to be changing over time.
- Verified Timeprints aggregation-logging hardware and reliable software interface to do the formal reconstruction.
- Verified (reviewed and formally proved) properties (for system's description, assumed, RV-checked and Objective properties) descriptions.
- An independent entity (or entities) in which the logged Timeprints traces are sent and stored. A complete chain of Timeprints transmission has to be defined transparently.

8.3.2 Evidence Context Formulation in the Literature

Despite these differences, between traces of models execution, and those logged from actual physical system execution, we can still make use of the rich literature in temporal logics and the associated verification methodologies. In the following, we start from existing work and explain the difference to our usage.

As mentioned briefly in the introduction, obtaining *evidence* from both the process and the product while operating in-field, was suggested to check safety properties at very high levels, as in the framework of *Open Adaptive Systems* [166] and [149]. In [142], the authors suggested evidence obtained by certificates named ConSerts [148] based on Digital Dependability Identities [141] to a case study of communications between vehicles, in a cooperative vehicle platooning function. within a scenario within [147]. The cars coordinate together via exchanging data through communications between vehicles. However, their work does not address how the communicated data are reported. While in their work, they focused on the high level concept, we focus here more on providing transparent tools than can be used in various contexts, to produce traces that can be used as evidence.

For our tools to succeed, the context in which they are used has to be defined. The Timeprints are logged continuously for a traced signal, yet the description of the system containing the signal and what environment it operates in need to be considered and described/formulated as needed. Defining the context also implies describing the timing and order of major events; so the Timeprints trace can be aligned. The context is very important because the Timeprints are just abstractions which actually means nothing without the context and the encoding which resulted in them.

8.4 Tracing Multiple Signals with Timeprints

Tracing multiple signals is related from one side to spatial abstraction, and how we can combine multiple signals to be traced together using Timeprints. From the other side, it is related to the reconstruction, by which combining the Timeprints-reconstruction of two different Timeprints can lead more efficient reconstruction.

Tracing multiple signals entails that we shall face the option of using more than one tracing clock. The other variable which can be changed in the length of the trace-cycle, even if the same tracing clock was used. Varying both (the clock-cycle width and the trace-cycle length) is still fine as long as the clock-cycles shall have at the end a minimum common multiple, and trace-cycles boundaries will always eventually intersect. The important aspect is that this intersection (meeting) of the trace-cycles boundaries, happens within very short intervals. Because the combined reconstruction, when carried on, shall happen over these intervals. And if these intervals are long, the reconstruction computing effort might turn-out prohibitive.

There is always going to be a common multiple at some point, but it is worthwhile to choose widths and lengths, such that their common multiple is really small. To make use of tracing multiple signals, dealing with them simultaneously –during

the reconstruction– can make reconstruction more efficient. It also enable checking input/output relations and can help pruning the search space efficiently.

Tracing with the same clock can be clarified here, to give an idea how this can be useful. We shall also start by assuming both signals are traced with the same trace-cycle length. If we are tracing to signals, i and o, both with the same clock clk and with the same trace-cycle length m. This is the simplest case, where we can use directly the reconstruction of any one signal of them, or of both combined.

The reconstruction of both can be used as follows: Let's call the XOR part of the Timeprint $tp.XOR$ as TA_i for the input and TA_o for the output. We also denote the number of changes part, i.e. $tp.n$, as k. The two Timeprints can be AND'ed and OR'ed together, and the result can be encoded together with the relation between the signals i and o. This means taking the bit wise AND and OR as follows:

$$TA_{io}^{AND} = TA_i \wedge TA_o \tag{8.7}$$

$$TA_{io}^{OR} = TA_i \vee TA_o \tag{8.8}$$

for the cardinality we have two different counts, for the OR count, the cardinality is going to be –in principal– adding both. But we know that the result is only an upper limit, expressing the case where changes in both signals did not co-inside. But we know that this might happen, and hence, the OR'ed cardinality is going to be just an upper limit, when encoded into the reconstruction CNF, rather than an exact number of changes.

$$k_{io}^{OR_{Upper}} = k_i + k_o \tag{8.9}$$

$$k_{io}^{OR_{Lower}} = k_i - k_o \tag{8.10}$$

The cardinality of the AND can be expressed by an upper and lower limits as well. obtained by abstracting the cardinality of both Timeprints as well:

$$k_{io}^{AND_{Upper}} = min(k_i, k_o) \tag{8.11}$$

$$k_{io}^{AND_{Lower}} = min(k_i, k_o) - (k_i - k_o) \tag{8.12}$$

And these also represent lower limits of cardinality, when $k_{io}^{AND_{Lower}}$ is less than zero, that value zero is just taken as a lower bound.

Notice than it might be case that changes do not coincide at all $k_{io}^{AND_{Lower}} = 0$, at the same time, their AND can still have a value (i.e. the $TA_{io}^{AND} \neq 0$). This does not express the case where no satisfaction is in their reconstruction set. Rather, it means the reconstruction set includes solutions where changes in i and o do not coincide.

8.4.1 Similar and Different Clocks and Trace-Cycle Lengths

The case of different trace-lengths is much easier than the case of different clocks. Because when the difference is only in the trace-cycles lengths, the variables used for each trace-cycle length can still be mapped to other variables of the other length; because they express some exact clock-cycle and mapping them is possible. However, path delays probabilities can come into play when we need to consider things happening in adjacent clock-cycles at differently located points.

The case is even more complicated when different clock-cycle widths (which are not multiples of each other) are used, It becomes not possible anymore to directly map the variables to each other. A variable indicting something in one clock-cycle shall correspond to what probably happened in many *parts* of other periods represented by other variables corresponding to the other overlapping clock-cycles. When in part of a clock-cycle the other signal is taking a value, and in another part another value; the above mentioned handling of OR and AND for the combined reconstruction is not possible anymore. In this case, handling of both signals together in some reconstruction CNF need to consider the intervals logic and use theories which handle that. This is a very active area of research at the intersection between SMT and Convex Polyhedra [33, 46], but we keep going further in that direction for future work.

8.4.2 I/O and Signals Interaction Checking with Timeprints

It is also worth mentioning that handling more than one signal together can only be beneficial if there is already a relation between them. Otherwise, making the reconstruction problem harder does not make sense if the signals are not related. Possible relations are inputs/outputs relations; and or multiple bus signals Timeprints. If layers of Timeprints are used, this also is a place where incorporating the layers together in the reconstruction can make sense.

For example this can be really useful if we talk about many signals on a bus, which change naturally together for example, and we have a Timeprint for each. Here we already know that all the signals are related, because the value which appears, changes all values at once; although of course some of the changes are going to be invisible to the change (because some bits of the new value might still have similar value as before, although the overall value has changed).

8.4.3 Modules Blaming for System Delays

A methodology for modules blaming for failures has been proposed in [73]. Our Timeprints can be used to aid in extending the methodology proposed to after deployment, via efficient logs which can enable accurate blaming for delays.

8.5 Timeprints Generalizations and Limitations

As we showed in the previous chapters, Timeprints can reveal some details, and they also abstract away others.

The number of clock-cycles in trace-cycles need to be constant only for every single clock-cycle alone. This means each trace-cycle may have different number of clock-cycles than the other trace-cycles. The only constraint here, to make all our formulations work over them, is that the number of clock-cycles (length) in every trace-cycle is known and fixed before the execution starts. This aims at fixing the trace-cycles length, so that exact interval lengths, in terms of clock-cycles can always be obtained. Using Timeprints, things can also happen concurrently, but when using one tracing clock, this happens if and only if, two changes happen at the same clock-cycle. As we use timeprints mainly for tracing, when tracing one signal, only one value of the traced signal can take place at a time. For digital signals this is even reduced to only one or zero traced at any clock-cycle instance. And as mentioned in the Timeprints introduction subsection, the change signals in this case can be used instead of the signal itself, wherever this makes sense. When dealing with multiple signals and multiple clocks, then notions of concurrency can be used to reflect clocks drift and synchronization problems.

Another problem facing the trace-cycle based abstraction, is how to handle multiple traced signals, with the same or different trace-cycle lengths. This problem is also visited briefly in this Chapter. After these two aspects are addressed, the *Timeprints Transform* (TT), is then ready to be presented. TT is a mapping which projects consecutive cycles of execution into the Timeprints domain; which is finite, yet each element in it is a set.

8.5.1 Multiple-bits Digital Signals Timeprints

Digital signals of more bits (vectors) can be dealt with in two different ways, depending on the application. If the system is functionally verified and run-time proved to act upon well defined specs, so that the consecutive values of the vectors are known and the only problem is in their timing, then the whole vectors can be abstracted

with its own change signal. Otherwise, if the vector values are unknown, not verified, or pure data signals, which their values are actually required to be traced, then a separate timeprint for each bit can be used. In this case a 32 bit word/vector requires up to 32 Timeprints. Using a Timeprint width of 32 for each, over 1024 long trace-cycles leads to exactly 1024 bits logged. This is still much more efficient than logging 32 bits every clock-cycle among the 1024 clocks. However, this reductions comes at the cost of reconstruction, not only in terms of how much computations it requires, but also in terms of the ambiguity that may result while reconstructing the original trace. While the computational effort might be acceptable, ambiguity is not. But as Timeprints' main purpose is to check properties rather than to recover full traces. Hence, the condition of having deterministic reconstruction –here checking-result out of Timeprints is the one to be formulated.

When it comes to abstracting more than one signal, the first question shall be whether to abstract each separately or all combined. For example, signals with proved relations between them (where both are using the same clock), –because it is formally proved and/or run-time monitored– can be abstracted together. In general, whenever the relation is known and the value of one signal can be inferred from the other; abstracting these two signals can be at least related, if not completely combined. The relation between different signals in this case can be used in formulating one big input, to the reconstruction problem, as was discussed briefly in section 8.4.

8.5.2 Timeprints Transform TT

In a sense, Timeprints generation pattern which we showed here looks a bit similar to Fourier Transform (FT). This can be seen from the sens of how it abstracts what happens during a fixed period of time, and convey that into one periodic train of Timeprints, sent itself over time, but in a completely different domain; which we call the *Timeprints Domain*. This domain is composed of all possible Timeprints; i.e. it is basically of the size 2^b, where b is the bit-vector width of the Timeprint.

When changes happen over a signal within a trace-cycle; the Timeprint summarizes this into one point in the Timeprints domain. This one point usually correspond to more than the trace-cycle which was actually mapped to it. But the Timeprints domain, due to its high abstraction level, can be amended by many properties to sort out the different points.

The properties, which can be described in other dimensions, are used to prune the search space as shall be seen in the next chapter. Properties preserving by Timeprints abstractions were discussed in Chapters 6 and 7. It is worth mentioning

that similar paradigm (to Timeprints transform), was recently suggested in relating SAT solving of XOR formulas using Fourier transform in [93].

As abstractions do not preserve every aspect, we discuss generalizing the abstraction function and the limitations thereof.

The Abstraction Function

As in Chapter 4, three levels of abstraction were used: the change abstraction, counting the number of changes, and finally the Timeprints (aggregating time-codes of changes) abstraction. The change abstraction can be generalized to designated types of changes experienced by the traced signal. For example, take the eye-diagram of an analogue signal. At any one time window, the eye-diagram is expected to take one of finite number of shapes which can be classified, according to the measured eye-opening and jitter. Mapping those to an n-digit value, describing the number of the class in which the current eye-diagram is, can be the first level of abstraction. When tracing one digital signal with multiple bits, counting the number of changes can be done for each bit separately or for the whole digit. As in the overview Chapters 3 and 4, XOR is an example of how The Timeprints abstraction can be done. The XOR function is one version of a generic mapping function, which uses codes to change the current value of the Timeprint. Many other functions can be used. Similar to XOR is addition, but a more sophisticated one can be designed to handle different positions in the trace-cycle and in the time-codes differently. In general this can be viewed by a k+2 dimensional matrix, where two dimensions express both position in the trace-cycle and bit-position in the time-code, and the rest shall express the function to do with each element in the two dimensions. The complexity of obtaining time-codes for such function has to be yet studied.

Whatever abstraction is used, it is going to utilize some area and use power to do that aggregation and logging. The time encoding complexity will also impacted by the choice of the abstraction function(s). The impact on the reconstruction time and complexity has to also be considered.

8.6 Summary

In this chapter, we filled some of the gaps between the foundations chapters, discussed limitations and some practical aspects so that Timeprints can be used. This way the chapter acts as a bridge, between theory presented before it, and the applications coming up in the next chapter; where practical case-studies are presented.

Chapter 9

Application: Timeprints in Field

After explaining the foundations and putting them together, the time has now come to illustrate that with practical applications.

We collected in this chapter the experiments done, to illustrate Timeprints generation and their usage in evidence-oriented tracing. The experiments fall into three main areas; namely: 1) bus signals tracing, 2) input/outputs signals tracing and 3) embedded software tracing.

In the first category, two buses are considered. The Advanced High-performance Bus (AHB) [15], over which the address signal was traced with Timeprints, to check for meeting deadline and in capturing un-expected temperature effect. The second bus is the Controller Area Network bus (CAN) [39]. Where delays of sent messages over the bus were checked. We also provide, for this bus, analysis of evidence of Timeprints in proving whether delays happened on it.

In the second area, sensor signals are considered to illustrate input signals tracing. In this experiment we illustrate also obtaining a priori proof about our capability to cover delays experienced by this signal. Adding an output signals, like braking signals to the tracing is also considered briefly at the end.

The Chapter ends with a summary which discusses limitations of the presented experiments and case-studies, and possible extensions, variants of theses experiments and further applications.

9.1 Bus Signals Tracing

The Timeprints and evidence-oriented tracing depends on encoding consecutive time instances, and then using these codes to summarize the temporal behavior. Major type of very fast signals which are challenging to trace are bus signals.

Some of bus signals are clocked while others are not. The clocked ones are much easier and very straight forward to trace with Timeprints.

Asynchronous and non-clocked signals are not less of common than clocked ones; and they can still be traced by Timeprints which work with double the clock of the fastest signal appearing on the bus, or with using the continuous signals tracing paradigm described later in section 9.3.2. Here, we consider the clocked bus signals tracing for our experiments.

9.1.1 Hardware Implementation

For experiments which have been conducted on hardware, we implemented a vhdl module, which was synthesized and loaded into Xilinx Spartan and zync devices, on both Digilent's Nexys3 and Zedboard development boards. The Nexys board was used for the AHB bus experiment described in 9.1.2. Similar implementation can be used for other experiments. However, in the rest we used simulation for generating the traces. Anticipated considerations are discussed at the end of the chapter.

A hardware tracer takes the signal under tracing as an input. Alternatively, there might be a collection of signals that triggers the events we need to trace. In such cases, we might want to trace related events together, in order to use less Timeprints. At the end, for each signal, or a collection of signals, there are going to be a change-detection circuit; which detects whether a change in the traced signal(s) has taken place.

Time-codes are hard-coded and stored (or generated) inside the tracer; the aggregation function –here XOR– is also implemented there. The m-Timecodes , each corresponds to a clock-cycle in the m-long trace-cycles. A timecodes ' hardware-pointer is incremented every clock-cycle, while the XOR-operation takes place at the traced signal's change.

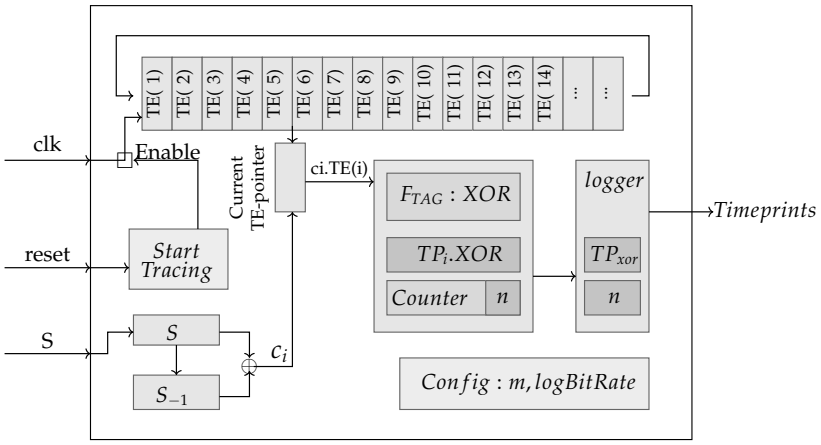


FIGURE 9.1: Hardware block-diagram of a Timeprints aggregation-log module

The first trace-cycle starts right at reset, or at a specific check-point that when reached triggers the $Start - Trace$ signal. The index of current time-stamp, is incremented every clock-cycle to point to the next time-stamp. Changes that happen in the value of S at instance i, causes a change-marker c_i to be $= 1$, where $0 < i \leq m$. Change-markers used as a trigger for the time-codes aggregation; i.e. the moment $c_i = 1$, the indexed timecodes by i is XORed to the current TP_i register, as shown in Fig. 9.1. A counter reaches n at the end of each trace-cycle, where n is the number of changes that trace cycle; n is also logged with the aggregated time-codes .

The tracer produces periodically, each trace-cycle a *timeprint* that represents an aggregation of exact n time-instances of change in the traced abstracted-signal. Hence, a continuous *Timeprints* trace is generated. Notice that the trace-cycle length m is much longer (e.g. 1024) than the logged Timeprint number of bits is: $b + log(m)$ (e.g. 24+10), and hence at the end of each trace-cycle we have more than enough time to send its timeprint before the end of the next trace-cycle, where the new timeprint will be ready.

9.1.2 Case Study 1: Temperature Compensated Refresh-Rate Detection

In this experiment, to obtain cycle accurate log of memory access, we implemented a *timeprints-agg-log*, (aggregation/logging) hardware, described in 9.1.1 and connected it to the address-signals of the AHB-AMBA bus. Timeprints were logged via a simplified USB-UART transmitter. Our *timeprints-agg-log* HW was, together with a LEON3 processor, implemented on a Nexys3 FPGA board. The whole system including the timeprints-generation was also simulated by Questa-Sim RTL cycle-accurate simulator.

Comparing the logs from HW to the simulated timeprints, obtained while running the same software image on both, we identified simulation configuration error, where the memory wait states were wrong in the SRAM model from the Gaisler simulation library [69]. After fixing this error, the number of changes k, in all trace-cycles became exactly the same (in simulation and log). However, the timeprints themselves started to differ after around 3000 clock-cycles (3 trace-cycles, as we chose $m = 1024$). Re-running several times, at different temperatures, the mismatch in timeprints started from as early as the third trace-cycle, to as late as the 28^{th}.

Starting from the trace provided by the simulation, we encoded it with the property that: one change instance is delayed by one-clock cycle. Doing this enabled us to identify the exact delay clock-cycle, each time a mismatch was found. From

```
2.257008s GearBoxInfo(1020)d 1 01             size  -> 58
2.253552s EngineData(100)d 8 00 00 19 00 00 00 00 00-> 125
2.256312s ABSdata(201)   d 6 00 00 00 00 00 00      -> 105
2.260804s Ignition_Info(103) d 2 01 00              -> 67
...
```

TABLE 9.1: Sample of CAN messages as they appear when read with CANoe Analyzer

these exact instances, we noticed that, during the execution, this one clock-cycle delay happens earlier if temperature is higher. The data-sheet of the memory chip, mentions a temperature compensated refresh rate, but it does not specify accurately its dependence of the chip temperature, which increases on its own by the execution itself; i.e. it even differs for different instruction sequences being run. This illustrates directly, how properties undefined at design-time could be traced on the cycle-accurate level, using *timeprints*.

9.1.3 Case Study 2: Checking whether a CAN message was sent before a deadline

The exact timing of the actual messages exchange on the bus in Controller Area Network (CAN), used in automotive for inter-modules communications, is vital in determining the actual transmission time that took place in the real-world. A sample of CAN messages log (as usually reported by the software) is shown in the listing in Table 9.1, where timestamps are at the left.

If we name these messages as $m_1, m_2, ...m_4$, then m_1 would appear on the CAN bus as (where ones corresponds to the bus's recessive state, and zeros to the bus's dominant state):

00111111110000000100000001000000010110000110111111111111

The CAN bus idle state is 1, hence m_1's first bit 0 is the start bit, and then comes the ID (1020=01111111100), then the data size, ... etc[1]. To log timeprints during the in-field message exchange on a 5 Mbps bus, a trace-cycle length of 1000 clock-cycles, and timestamps width of 24, were chosen. This means 5 timeprints and their respective number of change k were logged every second, i.e. 170 bps (5*(24+10)). Such available data from CAN messages logs can be encoded in our SAT-reduction. We built a tool, that directly takes CAN messages, and other temporal properties as input, and encodes the corresponding clauses to the SAT solver input.

In our experiment, two modules exchanging the message m_2 were involved in a late car response, and m_2's transmission time would determine who is responsible

[1]Details are in the ISO-11898:2003 Standard. We ignore bit-stuffing here for simplicity.

for the delay. The CAN messages listing above was from the transmitter. At the receiver, m_2, was received as:

```
2.253596s EngineData(100)d 8 00 00 19 00 00 00 00 00-> 125
```

The deadline was at the absolute timestamp 2.253580s. The logged timeprint, corresponding to the trace-cycle which started at 2.253400s was retrieved. The exact accurate reconstruction took 42.262s, and showed the message started at the 823^{rd} clock-cycle (corresponds to 2,253,564.6us) and ended after the deadline at 2,253,589.6us. Encoding the property that this message transmission happened on the wire before the deadline gave UNSAT in only 10.080s.

Being that small enabled simple and efficient logging and transmission hardware (hence no trace buffers are required) , and allowed saving data of hours in few Gigabytes. We used CANoe CAN-analyzer Demo9$_{\text{from Vector}}$ to generate a full scenario of exchanged CAN messages; over which we applied experimental delays.

9.1.4 Case Study 3: CAN Bus Messages Delays

Controller Area Network (CAN) is a standardized communication protocol used in cars today. The bus is one bit wide, which consists physically of two wires, being together in one of two state: Dominant (0) or Recessive (1). More details about the CAN protocol can be found in [39].

CAN frames are bit-streams, sent as bit-after-bit on the bus. The CAN specifications in [39] defined properties of CAN frames. These can be used as assumed properties P_A, to narrow the reconstruction space and enable covering all delays.

Checking CAN messages with Timeprints

Timeprints can be generated by choosing a fixed set of encoded timecodes TE of width w. As suggested before in [114], a suitable choice of timeprints design parameters is: $m = 1000$, and TE of width 24 and 31 (24 for incremental-linear-independent timecodes with LI-4, and 31 for random-linear-independent wit LI-4; see[112] or Chapter 5 for details). For a $1Mbps$ CAN bus, if we'd assign one clock-cycle for each bit appearing on the bus, one trace-cycle would be $= 1000 \times 1 \times 10^{-6}$, which equals $1ms$.

CAN protocol translated to assumed properties

The CAN frame specifications can be described over auxiliary variables, representing properties encountered within a frame. Then, to represent a delayed frame, the variables together are assigned to their sliding positions within a TC.

TABLE 9.2: Delays-Coverage Calculations, Scenario-Based.

Scenario of CAN Data	Number messages	RAN-UNIQ for each: # of cases of no evidence,	INC-LI4 # of non-provable messages,	RAND-LI4 % of covered messages
#1	4,000	24,926, 3995, 0.15%	370, 355, 91.125%	0, 0, 100%
#2	4,000	24,921, 3995, 0.15%	372, 355, 91.125%	0, 0, 100%

We tried to formally verify, a priori, whether a frame fulfilling Start bit, message length, and Fixed bits (exact known message contents) together will be covered by the chosen TE. When we tried to encode more properties we started running out of memory. We encoded the property that these conditions are fulfilled **and** that no collisions, as CNF formula with auxiliary variables. The result was that none of the input TE sets we used was able to cover all traces fulfilling this small set of assumptions only. This means more assumed properties are needed to reach full delays coverage. But as we tried to add these, the size of the CNF formula increased till the SAT-solver ran out of memory.

So we used simulation to list explicitly some elements of the set of signals, which correspond to the patterns that appears on the bus as a result of transmitting the CAN frame. For each frame, the corresponding message is checked for whether delays over it are covered or not; as per Subsection 7.4.3.

9.1.5 Covering Delays

To overcome the limitations of the huge CNF size, we conducted scenario-based coverage analysis. We here ask the question: is a specific set of CAN messages covered by certain encoding? We generated sets of CAN messages, over two simulated scenarios. Then we checked for each scenario, if we cover all possible delays. Table 9.2 shows the results for 3 different TE-sets, first with randomly encoded Timecodes (i.e. each timestamp code is just a random number), second, we used the incremental linear independent (of degree 4) INC-LI4, from [112]

All the encoding was chosen for $m = 1000$, and widths $31, 24, 31$ respectively for each column. Each cell in the table entries contains three fields. First field is the total number of collisions: this means for each pattern appearing in the scenario (i.e. a specific CAN message with its specific contents), collided (i.e. resulted in the same timeprint) with any delayed version of it, this is counted towards the total number of collisions. The second field is number of patterns that have one or more collisions. i.e. from the first table cell, for Randomly Unique (RAN-UNIQ) encoded timecodes , almost all messages except 5 (3995) exhibit one or more collisions. While for linearly-independent incrementally encoded timecodes (INC-LI4), only 355 out

of 4000 CAN-messages/patterns exhibit at least one collision; the total number of collisions is 370, as 15 of the patterns exhibit two collisions. The third field is the percentage of messages/patterns, that can be said to be DELAY-covered by the corresponding timecodes encoding. For example, by INC-LI4 91.125 % of the CAN data messages are covered, while all messages (100%) in the scenario were covered by RAND-LI4; which is the randomly-generated with checking linear-independent of degree 4. Notice the coincidence that similar number of messages, which cannot be proved/refuted (between scenario 1 and 2). Although the numbers of messages are the same, they had different timings of where these particular messages had no evidence. These incidences of absence of evidence is reported in the first number in each table-entry. The second number of the entry, is the number of CAN messages in the scenario, which had at least one such incidence of absence of evidence. The third number in an entry is the percentage of messages which has at least one incidence of absence of evidence. The absence of evidence, resulted from some CAN messages having similar timeprints when having different delays.

The results [2] shows that good encoding can do really well regarding the coverage of properties like delay. Although we could not prove that these encodings can cover all delays. But we have a statistical evidence that we can cover most of delays. Proofs of delays were obtained for all messages in both scenarios using the RAND-LI4 encoding. And in all cases, we can say after we get the timeprint whether it is evident regarding certain delay or not.

Remarks of Practical Considerations of the CAN bus Although in this analysis we did not consider re-transmissions or synchronization, these aspects can be added straight forward to our model. Failing messages, requiring re-transmissions, can be detected by the number of extra-changes not assigned to any message. These extra ones, do not affect the encoding of the CAN messages themselves, which are encoded by their content. We tried adding few noise signals; which did not affect the results above. But this is not yet exhaustive enough to indicate the performance in the presence of noise. We plan to follow a structures approach, to analyze how robust is the encoding against noise and re-transmissions.

For the synchronization, if the modules connected via the CAN bus are synchronized following [39], then a 4 times the suggested clock here is needed. This would lead to likely repetitions of each change signal going high, which would probably

[2] Run-times on Intel Core™i7-7500U CPU @ 2.70GHz x4 and Fedora25 with 15.6 GiB memory, to obtain the coverage. The scenarios were generated with Vector's CANoe CAN-Analyzer Demo9.

simplify the reconstruction and enable using longer trace-cycle length. But this solution is not yet fully implemented.

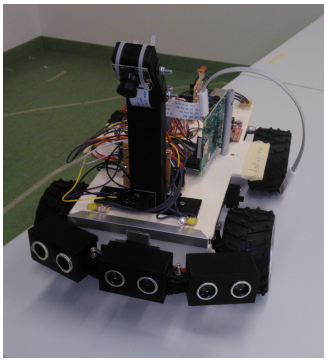
9.2 Input/Output Relations

9.2.1 Case-Study 4: Ultrasonic Sensor Data

Ultrasonic sensors are used for obstacles detection in cars. They work by issuing ultrasonic signals, either periodically or in-demand. They then wait for the echo of the issued signal, which is received, within an interval that is proportional to the distance to obstacles reflecting the signal. In this experiment, we used a simple donkey-car sensor, which its output was programmed as follows. When the time elapsed till receiving the echo signal is less than some threshold (selected as 18 ms, indicating the obstacle is close enough to stop) an *echo* signal is fired. This echo signal is used to trigger brakes, which can on its turn stop the car within certain period of time. That is why the accurate timing of the firing of such echo signals is critical.

Timeprints Design Parameters Selection

We illustrate the whole process of timeprints based tracing and properties checking to the sensors and braking data of an autonomous driving donkey-car; the one in Fig. 9.2. The car was equipped with three ultrasonic sensors and four servo motors for the brakes, one at each wheel[3].



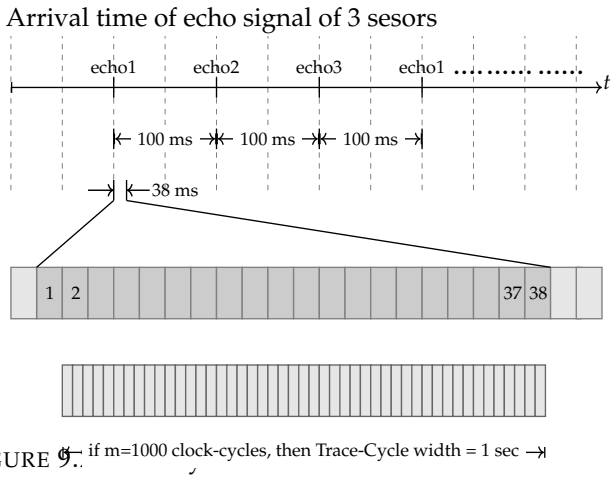


FIGURE 9.3: Sensor data in a trace-cycle

[3]This set-up was already existing in our research-group within the bachelor's-project DRIVE, and the data was obtained upon request from the students.

The car has been equipped with three ultrasonic sensors, each is configured to fire every 300 ms, and they were set up to fire in row, separated by 100 ms each. As can be seen in the Fig. 9.2, the 3 sensors are considered redundant; they will either all had an echo for the fired signal, or not. The difference in their orientation is minimum, and is adjustable in our set-up. The accurate time at which the echo of the fired signal is received, reflects the time taken for the signal to be reflected, and hence if it is smaller than certain threshold it would mean that the car has to start using the brakes to stop enough before it hits the obstacle. The accurate relative difference between the 3 echo signals received can also say something about at which direction exactly is the obstacle; especially if their orientation was different from each other.

To trace accurately the signal's echo time, we trace the signal at the pin "echo" of the sensor, which is raised high (i.e. to 1) by the sensor when it sends the sonic burst, and then goes low when it receives its echo. An echo would be received anyway, but if it was received before 38 ms, it means there is an obstacle closer than the range of about 5∼6 meters, and the distance can be calculated from the delay. If it was received at 38 ms, it means that the obstacle is relatively far away (more than 6 meters away). In our set-up because the car is moving slowly and the room is already small, the car considers braking only when the echo is received before 20 ms. Each sensor receives a fire command from software each 300 m sec, and replies back raising a pin high and then low when it receives the echo or when the 38 ms expires. As a designer's trace-related choice, we choose to combine all the signals together (3 firing signals + 3 echos received), as already one pin indicates the firing and the echo reception; and tracing each pin separately would mean logging three timeprints instead of one. We also know that the sensors send their signals in an interleaving manner; which makes it mostly possible to know which echo belongs to which sensor. Possibility remains, that sometimes due to different shifts in the firing times overlaps may occur. But even these shifts can be described as properties and used to point these out in many cases.

To illustrate using properties, we first use the clear example of the basic property of: 3 changes would occur separated by 100 ms, each followed by another change within 38 ms; see Fig. 9.4. Accumulated delays (shifts due to non accurate firings) also can be modeled, but will not be discussed here to keep the illustration simple. This property can be used to encode shorter time-codes that performs better than those who do not consider such property. But first before we delve into using properties, we show how to decide about the time-codes-set size (trace-cycle length and time-codes bit-width) in the first place. We clarify this more in the following.

Trace-cycle length

First, we have to decide about a trace-cycle size. Because the property is going to be described in terms of changes happening (or not happening) at consecutive clock-cycles within a trace-cycle. An echo transmitted and received from one sensor would cause 2 changes at the clock-cycles where it was raised high, and then at where it was made low. Here we assume it is enough to know when the signal is received within 1 ms resolution. The decision about tracing-precision should depend mainly on the system needs. Here for example: it depend on the allowed time to stop and the distance, the car is allowed to drive before it completely stops, starting from the moment and position it detects an obstacle. One msec accuracy corresponds to 17 cm error range in the distance of the obstacle at the moment it was detected. So, a clock-cycle of 1 msec is suitable. Choice of trace-cycle length of 100$\sim$1000 clock-cycles (i.e. 0.1 to 1 second) is in the desired range from hundreds to thousand; for small log size and reasonable timeprint-reconstruction time. What affects the exact choice of the trace-cycle length is the number of changes encountered inside one cycle; because this affects hugely both the ambiguity and reconstruction time; so we discuss it next.

Number of changes in a trace-cycle

If we choose a 1000 clock-cycles trace-cycle, we shall have ≤ 20 changes corresponding to firing and receiving the echo signals of the three sensors over 1 second. If we choose 0.1 seconds trace-cycle's length (100 clock-cycles), we'll have about 2 changes per trace-cycle, which is very few (makes it for example more efficient to just use the index and not to use any encoding at all). For a 200 clock-cycles trace-cycle, the index would need at least 8 bits, and for 4 changes that are expected within such trace-cycle a log would be 32 bits or even more if shifts lead to more changes. So at 200 clock-cycles, using encoding starts to make sense. In the following we will use both lengths: 200 and 1000 to illustrate the choice of the upcoming design options.

Using Properties

For example, here because we are getting one pair change separated by 38 clock-cycles every $\sim$ 100 ms, we can make the encoding more robust (produces unique results) for occurrences separated by less than 38 clock-cycles; like those in Table 5.3 2(b): D38b2, D37b2, D36b2... etc. Notice that any delay between 2 changes is already covered by LI-4, but these properties can be applied to other simple encodings like Index-k and Random-16/24 to make them produce unique results in these cases. One can choose to encode D100b10, D101b10, D102b10 and D103b10, for the 1000

trace-cycle. These properties encode the consecutive 10 firings within such trace cycle, within 100, 101, 102 and 103 msec distance (of no change, i.e. zeros) between them; as these delays have been seen frequently in heuristics. Encoding a property over a trace-cycle means modeling all its possible occurrences within the trace-cycle.

Notice that applying different properties has to be done recursively, until the set of time-codes saturates, and with keeping in memory removed time-codes that might be returned back if the base-timestamp –based on which they were removed— was itself removed. Saturation means that no removals to be done in the set because of violations of the properties. Of course to return a timestamp from such state it has to be checked recursively, to make sure it does not brake any of the previously checked properties. The list of remaining time-codes is checked at every stage, and is considered fulfilling the properties when all the properties-checks cannot remove any more time-codes from the list. An algorithm has been implemented to apply the above properties recursively. But it shall be published later after being checked for wider range of properties.

Generating Time-codes

For trace-cycles of lengths from 200 maximum time-codes bit-width should be 32, to make more efficient than logging the indexes. Less than this, we can try Inc-Index-k with applying the above mentioned properties. Inc-Index-1 would lead to the smallest bit-width if applied correctly. A faster way to reach the set of time-codes fulfilling these properties is to use a list of randomly generated time-codes and check them recursively. Random of width 8 would be too small even for 200 clock-cycles. 16 and 24 would be reasonable to try. An LI4 fulfilling time-codes set (satisfies linear independence of degree 4, either generated with random, incremental or greedy) would be already fulfilling all the delay between 2 properties (Dxb2). So to these LI4 fulfilling sets we can apply to them only the Dxb10 properties to enhance their performance (would then produce unique results).

9.2.2 Proving Ability of Covering Delays

In our set-up the ultrasonic signals were issued periodically every 100 ms; see Fig. 9.4. Here, we want to prove, before the deployment, that any delay in the echo signals will be captured, if we used certain encoding TE. The encoding we will check is the ones generated by the greedy algorithm from [112] satisfying LI-4. Of course this encoding does not cover all possible delays, as its bit-width is only 24.

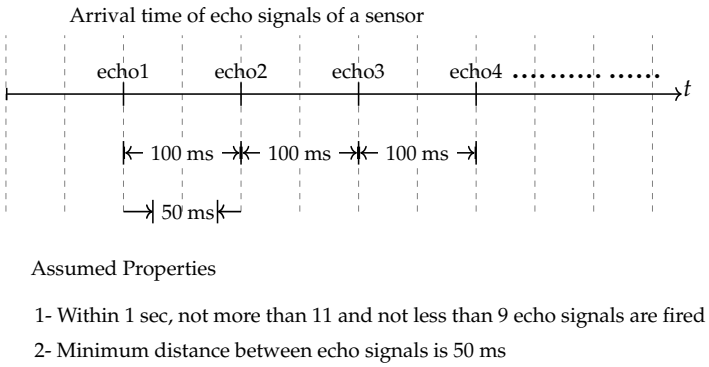


FIGURE 9.4: Sample of sensor echo signal versus time and the associated assumptions

So we need to add assumptions, and here we use the two assumed properties mentioned in Fig. 9.4. Assumption 1 is encoded as cardinality range, while assumption 2 is encoded as implications as in the examples in Section 8.2.1. We encoded them as CNF formula and ask the SAT solver to find any solution in all reconstruction sets, which does not satisfy the Hyperproperty P^*. And $P*$ here, expresses that all possible delays should be distinct, and is defined as follows. For every signal s_1 which satisfies the 2 above mentioned assumptions, the resulting Timeprint should be different from any other signal s_2, which has similar number of changes, but has different timings.

We encoded first only the 1st assumption, but it was not enough, as the SAT solver was still able to find solutions, where there were s_1 and s_2 of different timings, but still result in the same Timeprint. When we add the second assumption, the SAT solver query returned with UNSAT, announcing that this TE can capture all difference Timings constrained by these 2 assumptions. This hints to how desirable it is to implement run-time checks of these two assumptions, to have our evident backed up by them.

9.3 Chapter Summary

The chapter presented realistic applications and experiments, which showed how practical and useful Timeprints are. The experiments illustrated the choice of the Timeprints parameters, a detailed hardware implementation, which have been implemented in VHDL, sythesized, and used to log Timeprints to a host computer.

Beside the real implementations, the experiments also illustrated obtaining evidence of delays, and coverage of properties. This can make us safely conclude that Timeprints can be used in real world applications, to trace wide range of temporal properties and act as evidence when deployed and consulted, for failure analysis.

Here, a concrete hardware implementation is presented. The current implementation uses the clocked-version of time-codes. We also hard-coded these codes into the hardware. Details of our exact implementation comes next in the Applications Chapter. But it is worth mentioning here, that a time-codes generator can be a better option, for more efficient area utilization; especially if really long trace-cycles are going to be used. The Time-codes generators can be thought about as Pseudo-Random Noise PRN-generators. However, we did not think yet about the feasibility of generating our LI-4 and LI-6 codes using generators. It is rather an open problem for future work. Here, we focus more on the general considerations, and how the Timeprints generating hardware is expected to deployed.

General considerations of hardware implementation are crussial for the correctness of the time measurements results. Using digital design for the Timeprints generator, considering buffers for logging the status of last clock-cycle in the trace-cycle, without overlapping with the first trace-cycle. Buffering is possible because the logging is done independently from the capture. Considerations of the continuous Timeprints are considered next. These are important as these are the ones which are capable of providing the best known time measurement accuracy.

9.3.1 Considerations for Measurements Accuracy

The most important thing when it comes to time-accurate measurement, is that the hardware used for carrying on the measurement is itself accurate. Otherwise, we cannot trust the results.

For example, one aspect to consider in both FPGA and ASIC implementations, is where the different time-codes are stored; and whether they are going to have the same travel-time length (when triggered) to the XOR gate and the current Timeprint register. Testing and verifying all the resultant codes-registers triggering and inclusion delays is vital to ensure that they will perform as intended in-field, with the maximum devised operating frequency.

Related to that is the non-clock triggered time-measurement, which really depend on the travel-time of an electrical signal to trigger which time-code gets into the Timeprint. This is discussed in the next subsection.

9.3.2 Consideration for Continuous Signals Tracing

There are already in the literature several implementation of the so called Time/Delay measurement circuits; e.g. [16, 111]. The idea is mainly using a chain of Flip-Flops or latches, and take the signal travel time through them. How far did the

signal reach between two events, between which the time is to be measured. The start event triggers the signal travel, and the second event triggers a capture of at which latch is the signal at this capturing moment.

In the Razor paper [63], a method for dynamic detection of circuit timing error was introduced where monitoring the rate of errors was carried on using the Razor flip-flop which was used to detect the delays. The Timeprints as described uses clocks, rather than a signal travel time through latches. However, time-codes can be used for the consecutive latches, the same way they are used for the consecutive clock-cycles. This can enable the best known level of accuracy which can be achieved with today's technology. Other considerations related to transparency, certification and reliability come later in 10.3.4.

Chapter 10

Conclusion and Outlook

Reliance on Cyber-Physical Systems (CPS's) for safety critical applications is growing. And with the massive increase of existing systems complexity, speeds and dependence on artificial intelligence, the already hard verification task is getting much harder. Tracing an evolving system (if applying machine learning, automatic-updates, and self-modifying codes for example) is much harder than tracing a static one. We believer that, form the early development stages of such systems, tracing their behavior accurately must become inherent part of their development processes.

However, verifying designs during design-phase are not anymore enough. Run-time and transparent (Design-independent) verification, tracing and monitoring are strongly needed. Currently, only development-processes are documented and required to have artifacts proving they complied with the respective standards. But not the deployment-phase behavior, which –till today– is not yet required to be adequately monitored by any standard. Ongoing efforts, as the ITU group (FGAI4AD) addressing communicating autonomous driving decisions within the automotive domain, are good signs of paying attention to this situation.

From theory and engineering perspective: existing run-time verification methods are not independent enough to provide adequate level of transparency, accuracy and evidence. The level required is one which can be relied on, as a unbiased transparent tracing which all stake-holders can accept. These accurate and independent behavior execution traces are ideally capable of covering unexpected problems. Such traces can always serve as non-biased evidence.

These two aspects: transparency and independence, are the main two advantages of Timeprints and the Evidence-Oriented Verification EOV methodology we propose in this thesis. With Timeprints, new applications are made possible, for example: users are able to (with help of Timeprints hardware and software) verify that their complex devices are not executing functionalities which they do not approve, do not know they will be executing, or violate their privacy [71].

Evidence-oriented tracing as a methodology is new. Other methodologies which existed for verification and tracing are almost always concerned of facilitating designs, or monitoring the high-level health status. But neither the design-oriented methods can be extended to the run-time, nor the high-level run-time health monitoring can cover the wide range of problems which are not expected. Evidence-Oriented Verification and accurate and efficient traces like Timeprints; are the first means available for such purpose. We hope many will follow.

This chapter tries to conclude the thesis. It is not meant to summarize the work done; but rather emphasis its strengths and highlight the directions it open. We will start by a very brief summary, just to help lead the conclusion in context. Then, we highlight the strengths and follow by first the implications (requirements to enable Timeprints efficient utilization), then the doors opened as new directions by Timeprints and our evidence-oriented tracing.

10.1 Main Conclusions

First, let's summarize what we have presented in a paragraph. To solve the problem of in-field accurate yet efficient tracing, we presented a solution, based on logging *Timeprints*, by which verifying properties of what happened in field accurately can be conducted, in a framework of *Evidence-Oriented Verification*. These Timeprints are abstract traces of execution; which are efficient yet accuracy-preserving. *Evidence-Oriented Verification* is based on tracing the evidence, that something happened; rather than trying to trace the thing that happened itself. This did not only enable us to be efficient, as Timeprints can be orders of magnitudes smaller than the traced signal, but also enabled more independent tracing, capable of capturing un-expected and/or sporadic behaviors. This was possible because we relied on design-independent time encoding, to trigger the Timeprints. Unlike other run-time verification methods, ours is not fully dependent on the designers understanding of the system under tracing. This is not only useful because it enabled detection of sporadic and non-expected problems, but also because it provides a transparent way to check the compliance of complex systems to some specifications, using materially logged evidence, namely the *Timeprints*. Besides, Timeprints' systematic generating method positions them as reliable and formal evidence, of temporal behavior properties, such as meeting deadlines, aiding both real-time systems debugging (during development) and monitoring of in-field operation (during deployment). This ability to extend tracing and monitoring to in-field operations suggests their use for inter-modules liability assignment.

In the upcoming subsections, main strengths of Timeprints are more specifically highlighted in specific directions. Namely, we focus on Timeprints adequacy as evidence during deployment, and their usage in debugging during design time. We also emphasis how we envision their usage in devices.

10.2 Implications

The Timeprints have different barriers until they can be standardized part of devices. But if these barriers are overcome, they can left the level of safety, privacy and transparency, of electronic devices. The accurate tracing is particularly useful for Real-Time applications and Cyber-physical systems; where devices interact with users and the environment in Real-Time.

The other direction where Timeprints have great effects is privacy. Timeprints can enable capturing intrusions and revealing user privacy-violations. And to reach this, it implies we need to change the way we develop software, and hardware today. We explain this and the different implications needed until this point can be reached.

Lastly, in this subsection we discuss how Timeprints can result, on the long term, in a better user-understanding of their devices operation. These can be provided by looking at Timeprints as abstractions; which abstracts non-relevant details, and capitalize on that to explain the operation and better observe and analyze anomalies.

10.2.1 Implication on Safety and Better Development

Timeprints can enhance safety by two ways. First, knowing that the tracing is accurate and capable of revealing and proving more than anticipated problems, will enforce more careful development methodologies. For example, the pressure which is currently imposed by business and time-to-market, will be greatly lessened. As the proper evidence-oriented tracing, proposed by Timeprint will be able to reveal what exactly caused problems. There are going to be no justification to do reviews faster, just for the sake of getting to market before a competitor. When the tracing is enough transparent, it will not only be focused on tracing what the designer expects. Rather, designers are going to be forced to think about as much problems as possible, because what they miss can still be captured.

The second way the Timeprints are enhancing the safety is through their temporal accuracy and light weight. On one hand, they enable the best possible accuracy for assessing interaction between continuous and discrete signals, in hybrid systems.

In other words, we can now better check that an implementation did (or did not) actually meat its specification. even the metric temporal specifications; which includes interaction time with the external world. On the other hand, they also enhance our capability of reaching the problems root-cause. This is a result of being accurate yet light weigh, and hence can be used on wider range than available methods; hence, can reveal problems which might not have been seen before (as in our Temperature Experiment).

A third way, the Timeprints affect safety is indirect. When they participate by providing cycle-accurate traces of new factors, for which there was no other mean to know about them before they happen. This way, Timeprints act as a learning tool, from which we can know more about what we did not expect; or did not even know it exists.

10.2.2 Implication on Compilers and Better Security

Although we can apply an evidence-oriented tracing with Timeprints to software execution; as mentioned in [113, 115]. In the case-studies there, we needed full accurate simulation to be able to use the Timeprints in detecting hard to find errors or problems. Further checks of data integrity, not just their timing; or checking for anomalies, based on the presented scheme, can be made, independent from the full accurate simulation, if the software itself was compiled using the trace-cycles concept in head. To clarify this more; take the example of protecting mobile-users privacy from applications which try to miss-use the authorizations granted to it. For example, an application which can be used for recording audio messages, can also be used to record while the user is not actually using the recording feature. To detect such usage; and that it is outside the behavior which the user has intentionally allowed, tracing, with Timeprints, the application execution can capture such non-intended use-cases. However, a dedicated interface is needed to enable capturing such behaviors easily; i.e. to escape the need of trying a very wide range of possibilities, till capturing this behavior.

In order for embedded software to be compatible with a concept like Timeprints of evidence-oriented tracing, compilers can greatly help producing chunks of executable; which are meant to be executed within a trace-cycle. With this mapping parts of the executing software to the trace-cycles can make the reconstruction much easier. Using the concepts of executable chunks is not new [103]. It has been used in hard real-time applications to guarantee meeting the tasks execution-time/deadlines. Borrowing the same concept for trace-cycle compatible code; can

greatly facilitate using Timeprints.

Similarly, tweaking the compiler output to comply with security requirements, as robustness to side channel attacks, are common [171]. Generating such periodic pattern of execution is inline with the trace-cycles based abstraction methodology. Mapping periods, which can be taken as blocks as advised by [18], to trace-cycles can be straight forward. This means that having the concept of consecutive constant execution periods; over which Timeprints of software-execution can be logged is compatible with best practices in security. This is actually stemming from the design-independence of Timeprints. Because we are in the first place concerned of tracing evidence; the result is actually not leaking information, but rather what is being logged are meta data which are designed to act as checks; not to reveal data.

However, an application-oriented structured frameworks are needed to prove the two main aspects of Timeprints: adequacy as evidence (initially provided in the thesis) as well as proving that Timeprints do not leak information by themselves. Formal proofs of the evidence aspect was presented in this thesis. The non leaking features are on-going.

10.2.3 Implication on Abstractions and Better Efficiency

Timeprints are abstractions. We have shown in Chapter 4, how they abstract the temporal behavior in a way which enables them to provide information, about what we do not know, given the information which are already proven. Their compact size and orientation towards evidence, provide developers and designers with better observability and understanding, of the implementation problems. From one side, it builds on having parts and features proved to hold, which already enforces formal and structural development methods. Form the other side, because Timeprints basically enable noticing the unexpected, they help focusing on what is missing, and on what might have went wrong. Traditional debuggers and tracers just provide detailed traces of execution, and the developer/tester has to figure out where to look and what to look for among a forest of information. With the timeprints, aspects which are already known or proved to be working correctly are used to extract what have went wrong, using the Timeprints.

Because Timeprints are used on the hardware, they are extremely useful at the integration testing phase. The bugs usually discovered at such late phases are the hardest to detect and analyze. This is where Timeprints can help the most; not only because they enable detecting the unexpected, but also because the reconstruction enables focusing on problems detection. And also the focus itself is fully directed

towards finding the problem, rather than clueless-ly inspecting all the available data. This is expected to shorten the time taken to debug problems; especially the ones which occurs later in the development life-cycle; which are actually the ones which harder to detect and cost the most. Figure 10.1 below shows three different forms of how traces may look like, if logged to trace the values of some signal x over time. As mentioned in Timeprints limitations, they do best getting unique reconstruction for small number of changes; this is where the value.time pair works well. However, the Timeprints then have the advantage of proving a consistent size of the log over time.

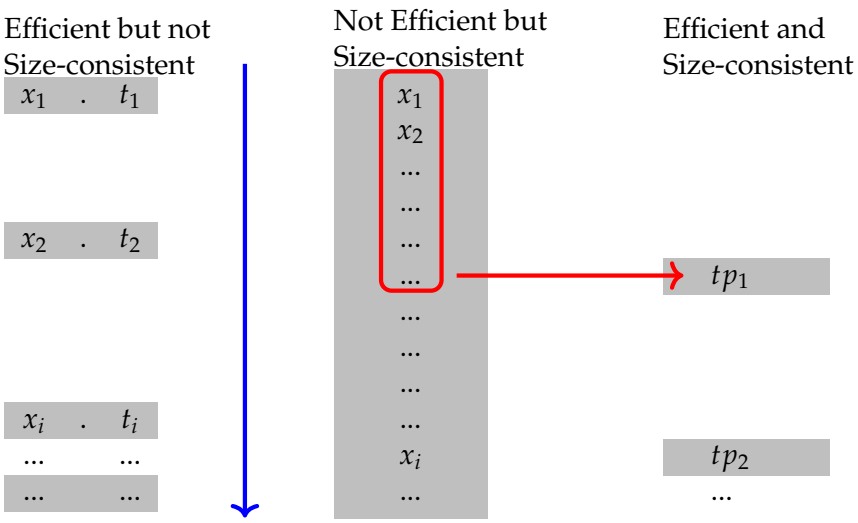


FIGURE 10.1: Traces with timestamps to the left, then traces with a record every clock-cycle, and at the right a trace of Timeprints and how they related to a clock-cycle recorded trace

We have already discussed many of the future directions, but in the following subsections, we focus on more concrete steps which we envision can be carried on in the near future.

10.3 Future Work at Evidence Direction

Timeprints usage as evidence have been the topic of Chapter 7, and two of the case-studies presented in Chapter 9. There the theory behind Timeprints validity as evidence was presented. To make this one step closer to being practically of wide use some work still need to be done. Plausibility of Timeprints as evidence, which was presented here is not enough. There is still a great barrier in writing the code which applies the concepts presented here and be highly trusted. For example, the encoding of properties are still done manually. This is one area where automation is highly

needed. Most manual processes are prone to errors and bugs. Automating property encoding can enable using formal methods once more to verify the correctness of such encoding. This shall enable focusing more on rigid and clear descriptions of properties; and this in turn should enable more people to be able to look and verify the correctness and validity of properties encoding.

Building specialized tools which can directly extract evidence for certain applications is a necessity, to make Timeprints usage on a wide scale possible. However, these tools are all going to employ the same basic concepts. So the potential of reusing many parts of the codes for different applications will be high.

Of course, it is not straight forward to use Timeprints in all complex devices existing today in the market; a lot of work are still to be done, especially at the system analysis level to determine which signals and properties to trace and which timecodes to use. It is a hardware based solution; so compliance has to start from the devices manufacturers. However, as end-users, in that scenario, are the direct beneficiaries from incorporating such a signal/device-tracing mechanism; they have to be the ones who want it to be included in these devices. When Timeprints are logged from devices, where the designers know that they have to think harder because more than what they think can be revealed, they are going to put more effort in designing better and more robust systems. Besides, a proper evidence-oriented tracing can enable capturing attacks and trials to use the devices by any party, which tries to use it outside its announced usage and features. This includes capturing cyber-attacks and security breaches. When Timeprints are included in devices, it means the usage of these devices is producing footprints, or fingerprints like of the actions done over the device. And in court, Timeprints can act as evidence to prove misuse, if needed. And because they are independent, they cannot be used by a party against the other; but rather like an evidence-bookkeeping, which just simple reveal what really happened.

In this sense, Timeprints and evidence-oriented tracing are bringing a new physical artifact, which can be used as evidence for more just technology assessment. This orientation, however, is not only limited to deployment time. Such level of accurate tracing will also benefit designers in debugging and tracing their systems; either software or hardware. We have shown in the fore chapters how this can be done. Today's systems developers and testers have hard time until they can understand bugs root-causes. Debuggers, tracers and other means for on-line testing are very limited in what and how they provide traces. The degree of accuracy provided by Timeprints can support these tools with a more efficient tools, to facilitate reaching this endeavor as well.

Timeprints can be of great use in Post silicon verification/debugging, protocols monitoring and embedded SW debugging. Timeprints are also suitable for embedded software tracing. The work presented in the applications here, still used RTL simulation to benefit from existing near to accurate trace. But it is possible to develop the method to work without needing that level of detailed simulation. This can be achieved for non self-modifying code by having the control flow graph of the executable code. Light size of Timeprints makes them also candidates for assisting in tracing, not only I/O's between chips on boards, or modules on an SoC. But this can also be extended to interposers and Chiplets. Their adequacy as evidence enables them to continue not only being used in debugging, but later also to enable detailed liability analysis of in-field incidences.

10.3.1 Theorem Proving and Evidence of Timeprints

What we have used so far, is very similar to theorem proving. However, we did not use any automated theorem-proving technique explicitly; as we did not yet solve a complex case which requires doing so. However, in case-study 9.1.4, using theorem proving was the natural choice to express incremental usage of assumed properties until the proof is reached. Theorem proving and SMT can also help encoding complex properties like the CRC and frame-length in the CAN-bus frames. These properties are good examples of properties which are very hard (in terms of finding efficient yet correct encoding of them) and/or expensive (in terms of variables and clauses needed) to encode in CNF.

Theorem proving can provide a very efficient alternative to the CNF encoding of properties. It can also be more natural, not just efficient to use when we start dealing with multiple signals and multiple trace-cycles. In these two cases, the size of the CNF formula grows exponentially; and theorem-proving can be a direct solution to get around limitations of memory and computing power. They are also a natural fit for the problem; which can be always summarized down into a Hoare-triple, where the current problem description is a term, the Timeprint(s), a term; and finally the conclusion is the last term which represents the check.

10.3.2 Bus protocols Signatures Standardization

Data exchanges over Buses using different protocols are the basic form of data transfer today. The protocols have clear specifications; by which the compliance is very critical for correct inter-operation of elements connected to the bus. This is one area

where Timeprints can help the most as evidence of whatever have been exchanged on a bus.

Although till now, we only used Timeprints in tracing signals over buses where constant clocks have been used; Timeprints can still be used to trace signals over buses with various clocks, using the smallest; or the smallest common divider of the clocks used. A challenging problem occur in the asynchronous buses, where the speed is very high, that a proper fraction of the smallest common divider of the smallest clock is hard to physically realize.

For buses of low speed as the CAN bus, this is not a problem at all. To explain more, the CAN-bus standard already specifies one forth of the clock as a minimum by which synchronization should occur. Hence, four times of the maximum bus speed can be used for the Timeprints safely.

10.3.3 Periodic Execution Path Signatures as Proofs

Embedded software execution, during deployment, can be fully checked and proved by Timeprints, if the software images were generated using this into consideration. If compilers considerations discussed in 10.2.2, are applied, methods like [19] for example, the periods after which the execution of certain blocks can be known to produce well defined sets of Timeprints.

In this case, different Timeprints for different applications can be generated; not just one Timeprint of the main processor execution. When this is the case, it becomes possible to directly map these execution Timeprints into signatures, which can prove which execution has exactly taken place; and can also directly highlight anomalies if any. This is a very interesting direction which we just started lately to work on.

10.3.4 Hardware Realization Considerations

A hardware which can be trusted to generate what can be relied on in courts as evidence, has to be extremely trust-worthy. From our point of view, trust-worthiness is based in the first place on transparency. The Timeprints generating hardware and the software used for reconstruction, both have to be fully open and clear. Explanations of their functionality and implementation have to be made openly accessible.

The physical implementation itself also has to be transparent to the level it can be checked, certified, and on its course also, monitored continuously during its lifetime. Beside transparency, equally important is the accuracy. Means to continuously check the input clock's for skews and jitters are mandatory. Clear means of synchronizing the tracer with traced systems reset and check-points are needed.

10.4 Future Work at Continuous Signals Timeprints

Continuous signals, can mean two things: either signals which takes different values over intervals of time smaller than the clock period. This one is what we mean here. The other aspect of continuouity is in the traced signal value; which can take values more between the quantized digital values. We don't consider those here.

When focusing on continuous signals, which changes faster than the clock; the concept of Timeprints can be applied on consecutive latches level. By this, a next time-code shall correspond to a signal reaching the next latch. The number of latches which are expected to be crossed by the signal within a clock-cycle, shall represent the length of the continuous-tracing-cycle. A continuous Timeprint, generated at the end of this continuous-tracing-cycle, is considered as an n-dimensional digital signal, which can be traced on its turn by n Timeprints. This way, although a bit expensive (n Timeprints for $b = n$ bits, are needed), is still a good solution when higher than clock accuracy is needed. We still need realistic case studies to show the validity of this assumption. Interaction between different chips and chip-lets, or data exchange in chunks over multiple Giga Hirz buses, can illustrate this usage.

10.4.1 Addressing Dynamic Systems Theoretical Limits

The same idea we used here for continuous Timeprints can be applied to analyzing dynamic aspects of systems. When it comes to interaction between digital systems and continuous world, Such accurate tracing can help us learn more about the continuous world and the environment. We shall still be limited to the latch width; but this is already much better than the clock period which is the current limit.

10.5 Spaceprints, Memoryprints and SpaceTimePrints

We believe that, like Timeprints, we can create *Spaceprints* and *Memoryprints*, to verify the status of pieces (modules) composing a hardware. By Spaceprint, we mean creating a compact signature from the 3 dimensional hardware; which is capable of being trusted as an ID of the underlying hardware it describe. Already something similar exist in the literature, under the name of *Chip-Fingerprint* [81] or Physically Unclonable Functions (PUF) [79]. It is however done for a whole chip. If we can have it on the modules level; they can provide more insight; especially if something went wrong and a module isolation is needed. What also can be added is a time-frame, over which the Space-print is made, logged or checked. Many components

can be added to the print itself to enable them to express factors like aging; beside being able to indicate any malicious configuration or inserted hardware.

Memory is crucial because its content is actually the main driver of execution. Timeprints detect the execution itself; and as seen in the experiments, to use them in tracing software, a major assumption was that the software doesn't modify itself. If the code-memory might change, a signature indicating so would be needed to still stay able to trace the execution. We believe applying similar concept as that of the Timeprints, but for memory content, which can be called Memoryprints can help extending the tracing capability one step further. Care must be taken that the Memoryprints, Timeprints and any other logged signature remain small enough not to leak any information, from which non-authorized parties (whom do not have access to the assumed properties) can reconstruct the execution.

10.6 Message and Signature

The principle of *Timeprints* can be extended to whole systems' *execution-prints*. In this sense, our Timeprints-like logs are periodic (every trace-cycle) system execution signatures of the traced system-signals. We envision having Timeprints from the system's main inputs and outputs, plus some internal critical signals, and efficiently concatenating these together to result in a system execution signature.

This is a sound idea when we first carefully analyze the system as a whole, to select its critical/decisive signals and their interactions. At the same time, we have to ensure that the logged system-*prints* are not going to be revealing or leaking secret information. Having a system execution continuous signature is particularly useful for generating evidence, needed to prove failure root-cause(s) or for analyzing malicious behaviors. Capturing such trails helps detecting intrusions and un-expected behaviors. Systems can be designed from the beginning so as to result in execution signatures, which can reveal patterns of behavior which might indicate anomalies or risks. Having such execution signatures enables faster capture of anomalies; and hence limiting or even totally avoiding possible damages.

By providing the chance to capture un-expected behaviors; a new horizon for tracing is opened; a one was not possible before Timeprints.

Bibliography

[1] *A wave of AV safety standards to hit in 2020*. URL: https://www.eetimes.com/a-wave-of-av-safety-standards-to-hit-in-2020/ (visited on 04/05/2021).

[2] Houssam Abbas, Yash Vardhan Pant, and Rahul Mangharam. "Temporal Logic Robustness for General Signal Classes". In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. HSCC '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, 45–56. ISBN: 9781450362825. DOI: 10.1145/3302504.3311817.

[3] Gul Agha and Karl Palmskog. "A Survey of Statistical Model Checking". In: *ACM Trans. Model. Comput. Simul.* 28.1 (Jan. 2018). ISSN: 1049-3301. DOI: 10.1145/3158668. URL: https://doi.org/10.1145/3158668.

[4] *Aircraft Accident Investigation Report, KNKT.18.10.35.04*. URL: http://knkt.dephub.go.id/knkt/ntsc_aviation/baru/2018%20-%20035%20-%20PK-LQP%20Final%20Report.pdf.

[5] A. David et al. "Statistical Mode Checking for Stochastic Hybrid Systems". In: *Proceedings of First International Workshop on Hybrid Systems and Biology*. EPTS. 2012.

[6] Debjit Pal et al. "Application Level Hardware Tracing for Scaling Post-Silicon Debug". In: *(DAC 2018)*. 2018.

[7] N. Decker et al. "Online analysis of debug trace data for embedded systems". In: *DATE, 2018*. 2018.

[8] R. Drechsler et al. "Completeness-Driven Development". In: *Graph Transformations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-33654-6.

[9] James F Allen and George Ferguson. "Actions and events in interval temporal logic". In: *Journal of logic and computation* 4.5 (1994), pp. 531–579.

[10] Shaull Almagor et al. *Invariants for Continuous Linear Dynamical Systems*. 2020. arXiv: 2004.11661 [cs.LO].

[11] Rajeev Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[12] Rajeev Alur and David L Dill. "A theory of timed automata". In: *Theoretical computer science* 126.2 (1994), pp. 183–235.

[13] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. "Theory in Practice for System Design and Verification". In: *ACM SIGLOG News* 2.1 (Jan. 2015), 46–51. DOI: 10.1145/2728816.2728827.

[14] Rajeev Alur et al. "Discrete abstractions of hybrid systems". In: *Proceedings of the IEEE* 88.7 (2000), pp. 971–984.

[15] *AMBA Specifications*. URL: https://developer.arm.com/architectures/system-architectures/amba/specifications (visited on 04/05/2021).

[16] Niklas U Andersson and Mark Vesterbacka. "A Vernier time-to-digital converter with delay latch chain architecture". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 61.10 (2014), pp. 773–777.

[17] *ARM coreSight and Embedded Trace Macrocell ETM*. URL: http://www.arm.com (visited on 04/05/2021).

[18] Z. B. Aweke and T. Austin. "Øzone: Efficient execution with zero timing leakage for modern microarchitectures". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 1123–1128. DOI: 10.23919/DATE.2018.8342179.

[19] Gilles Barthe et al. "Formal verification of a constant-time preserving C compiler". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–30.

[20] E. Bartocci et al. "Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications". In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Springer International Publishing, 2018.

[21] Ezio Bartocci et al. "Introduction to Runtime Verification". In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Cham: Springer International Publishing, 2018, pp. 1–33. ISBN: 978-3-319-75632-5.

[22] David Basin, Felix Klaedtke, and Samuel Müller. "Policy monitoring in first-order temporal logic". In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 1–18.

[23] David Basin et al. "Monitoring metric first-order temporal properties". In: *Journal of the ACM (JACM)* 62.2 (2015), pp. 1–45.

[24] Kanad Basu and Prabhat Mishra. "Efficient trace signal selection for post silicon validation and debug". In: *2011 24th Internatioal Conference on VLSI Design*. IEEE. 2011, pp. 352–357.

[25] Kanad Basu and Prabhat Mishra. "RATS: Restoration-Aware Trace Signal Selection for Post-Silicon Validation". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.4 (2013), pp. 605–613. DOI: 10.1109/TVLSI.2012.2192457.

[26] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (Sept. 2011). ISSN: 1049-331X. DOI: 10.1145/2000799.2000800. URL: https://doi.org/10.1145/2000799.2000800.

[27] Thomas Bayes. "LII. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, FRS communicated by Mr. Price, in a letter to John Canton, AMFR S". In: *Philosophical transactions of the Royal Society of London* 53 (1763), pp. 370–418.

[28] Gerd Behrmann, Alexandre David, and Kim G Larsen. "A tutorial on uppaal". In: *Formal methods for the design of real-time systems*. Springer. 2004, pp. 200–236.

[29] Bob Bentley. "Validating the intel pentium 4 microprocessor". In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 244–248.

[30] E. Berlekamp, R. McEliece, and H. van Tilborg. "On the inherent intractability of certain coding problems". In: *IEEE Transactions on Information Theory* 24.3 (1978).

[31] Valeria Bertacco. *Scalable Hardware Verification with Symbolic Simulation*. Springer Science & Business Media, 2006.

[32] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[33] Bernard Boigelot and Isabelle Mainz. "Efficient symbolic representation of convex polyhedra in high-dimensional spaces". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2018, pp. 284–299.

[34] Patricia Bouyer et al. "Model checking real-time systems". In: *Handbook of model checking* (2018), pp. 1001–1046.

[35] Patricia Bouyer et al. “Model checking timed automata”. In: (2008).

[36] Torben Braüner, Per Hasle, and Peter Øhstrøm. “Determinism and the origins of temporal logic”. In: *Advances in temporal logic*. Springer, 2000, pp. 185–206.

[37] J.R. Burch et al. “Symbolic model checking: 1020 States and beyond”. In: *Information and Computation* 98.2 (1992), pp. 142–170. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90017-A. URL: https://www.sciencedirect.com/science/article/pii/089054019290017A.

[38] Simon Burton et al. “Safety, Complexity, and Automated Driving: Holistic Perspectives on Safety Assurance”. In: *Computer* 54.8 (2021), pp. 22–32. DOI: 10.1109/MC.2021.3073430.

[39] *CAN Specification, Bosch, Robert Bosch GmbH, Postfach 50, D-7000 Stuttgart 1, Germany, 1991*. (Visited on 04/05/2021).

[40] Gustavo Carvalho, Augusto Sampaio, and Alexandre Mota. “A CSP timed input-output relation and a strategy for mechanised conformance verification”. In: *International Conference on Formal Engineering Methods*. Springer. 2013, pp. 148–164.

[41] Kārlis Čerāns. “Decidability of bisimulation equivalences for parallel timer processes”. In: *Computer Aided Verification*. Ed. by Gregor von Bochmann and David Karl Probst. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 302–315. ISBN: 978-3-540-47572-9.

[42] *Cerebras Unveils Wafer Scale Engine Two (WSE2): 2.6 Trillion Transistors, 100 Yield*. URL: https://www.anandtech.com/show/16626/cerebras-unveils-wafer-scale-engine-two-wse2-26-trillion-transistors-100-yield (visited on 04/05/2021).

[43] Serenella Cerrito, Marta Cialdea Mayer, and Sébastien Praud. “First Order Linear Temporal Logic over Finite Time Structures”. In: *Logic for Programming and Automated Reasoning*. Ed. by Harald Ganzinger, David McAllester, and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 62–76. ISBN: 978-3-540-48242-0.

[44] P. Chini et al. “Fast Witness Counting”. In: *CoRR* abs/1807.05777 (2018). arXiv: 1807.05777. URL: http://arxiv.org/abs/1807.05777.

[45] *ChipScopePro*. URL: www.xilinx.com/products/design-tools/chipscopepro.html (visited on 04/05/2021).

[46] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. "An SMT-based approach to weak controllability for disjunctive temporal problems with uncertainty". In: *Artificial Intelligence* 224 (2015), pp. 1–27.

[47] Alessandro Cimatti et al. "SMT-based satisfiability of first-order LTL with event freezing functions and metric operators". In: *Information and Computation* 272 (2020), p. 104502.

[48] Edmund Clarke et al. "Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems". In: *International Journal of Foundations of Computer Science* 14.04 (2003), pp. 583–604. DOI: 10.1142/S012905410300190X. URL: https://doi.org/10.1142/S012905410300190X.

[49] Edmund Clarke et al. "Counterexample-guided abstraction refinement". In: *International Conference on Computer Aided Verification*. Springer. 2000, pp. 154–169.

[50] Edmund M Clarke Jr et al. *Model checking*. MIT press, 2018.

[51] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *J. Comput. Secur.* 18.6 (Sept. 2010), pp. 1157–1210. ISSN: 0926-227X.

[52] *Collision Between a Sport Utility Vehicle Operating With Partial Driving Automation and a Crash Attenuator, Mountain View, California, March 23, 2018, HWY18FH011*. 2018. URL: https://www.ntsb.gov/news/events/Documents/2020-HWY18FH011-BMG-abstract.pdf (visited on 04/05/2021).

[53] Lukas Convent et al. "Hardware-based runtime verification with embedded tracing units and stream processing". In: *International Conference on Runtime Verification*. Springer. 2018, pp. 43–63.

[54] Martin Davis and Hilary Putnam. "A computing procedure for quantification theory". In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

[55] Giuseppe De Giacomo and Moshe Y Vardi. "Linear temporal logic and linear dynamic logic on finite traces". In: *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. Association for Computing Machinery. 2013, pp. 854–860.

[56] Lauterbach Debuggers. URL: www2.lauterbach.com/pdf/main.pdf (visited on 04/05/2021).

[57] Stéphane Demri, Valentin Goranko, and Martin Lange. *Temporal Logics in Computer Science: Finite-State Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2016. DOI: 10.1017/CBO9781139236119.

[58] Andrew DeOrio, Daya Shanker Khudia, and Valeria Bertacco. "Post-silicon bug diagnosis with inconsistent executions". In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2011, pp. 755–761.

[59] Jean Dezert, Pei Wang, and Albena Tchamova. "On the validity of Dempster-Shafer theory". In: *2012 15th International Conference on Information Fusion*. IEEE. 2012, pp. 655–660.

[60] David L. Dill. "Timing assumptions and verification of finite-state concurrent systems". In: *Automatic Verification Methods for Finite State Systems, CAV 1989. Lecture Notes in Computer Science, vol 407*. Ed. by Joseph Sifakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 197–212. ISBN: 978-3-540-46905-6.

[61] Alexandre Donzé. "On signal temporal logic". In: *International Conference on Runtime Verification*. Springer. 2013, pp. 382–383.

[62] Deepak D'Souza and Pavithra Prabhakar. "On the expressiveness of MTL in the pointwise and continuous semantics". In: *International Journal on Software Tools for Technology Transfer* 9.1 (2007), pp. 1–4.

[63] Dan Ernst et al. "Razor: circuit-level correction of timing errors for low-power operation". In: *IEEE Micro* 24.6 (2004), pp. 10–20.

[64] Salessawi Ferede Yitbarek and Todd Austin. "Neverland: Lightweight Hardware Extensions for Enforcing Operating System Integrity". In: *arXiv e-prints*, arXiv:1905.05975 (May 2019), arXiv:1905.05975. arXiv: 1905.05975 [cs.CR].

[65] *Final 737 MAX Report for Public Release*. 2020. URL: https://transportation.house.gov/imo/media/doc/2020.09.15%20FINAL%20737%20MAX%20Report%20for%20Public%20Release.pdf.

[66] Bernd Finkbeiner. "Model Checking Algorithms for Hyperproperties (Invited Paper)". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Fritz Henglein, Sharon Shoham, and Yakir Vizel. Cham: Springer International Publishing, 2021, pp. 3–16. ISBN: 978-3-030-67067-2.

[67] Michael J Fischer and Richard E Ladner. "Propositional dynamic logic of regular programs". In: *Journal of computer and system sciences* 18.2 (1979), pp. 194–211.

[68] Carlo A Furia et al. "Modeling time in computing: a taxonomy and a comparative survey". In: *ACM Computing Surveys (CSUR)* 42.2 (2010), pp. 1–59.

[69] *Gaisler Research. LEON3 synthesizable processor*. URL: http://www.gaisler.com (visited on 04/05/2021).

[70] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Mathematical Sciences Series. W. H. Freeman, 1979. ISBN: 9780716710448. URL: https://books.google.de/books?id=fjxGAQAAIAAJ.

[71] *General Data Protection Regulation, GDPR*. URL: https://gdpr-info.eu/ (visited on 04/05/2021).

[72] *GM Recalling 4 Million Vehicles for a Software Defect Linked to One Death*. 2016. URL: https://www.cnbc.com/2016/09/09/gm-recalling-4-million-vehicles-worldwide-for-software-defect-linked-to-1-death.html (visited on 04/05/2021).

[73] Gregor Gößler and Daniel Le Métayer. "A general framework for blaming in component-based systems". In: *Science of Computer Programming* 113 (2015), pp. 223–235.

[74] M. Hamad et al. "Prediction of Abnormal Temporal Behavior in Real-Time Systems". In: *(SAC 2018)*. 2018.

[75] Kihyuk Han, Joon-Sung Yang, and Jacob A Abraham. "Dynamic trace signal selection for post-silicon validation". In: *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*. IEEE. 2013, pp. 302–307.

[76] Felix Hausdorff. "Mengenlehre". In: *Berlin and Leipzig* (1927).

[77] Klaus Havelund, Grigore Rosu, and Peter Norvig. "Testing linear temporal logic formulae on finite execution traces". In: (2001).

[78] Jon Hemmerdinger. *FAA addresses potential data-display fault in 787 avionics*. URL: https://www.flightglobal.com/systems-and-interiors/faa-addresses-potential-data-display-fault-in-787-avionics/137403.article (visited on 03/19/2021).

[79] Charles Herder et al. "Physical unclonable functions and applications: A tutorial". In: *Proceedings of the IEEE* 102.8 (2014), pp. 1126–1141.

[80] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. "Online monitoring of metric temporal logic". In: *International Conference on Runtime Verification*. Springer. 2014, pp. 178–192.

[81] Daniel E Holcomb, Wayne P Burleson, and Kevin Fu. "Power-up SRAM state as an identifying fingerprint and source of true random numbers". In: *IEEE Transactions on Computers* 58.9 (2008), pp. 1198–1210.

[82] *Hyperdrive Daily Car Crashes have a Black-box Problem*. URL: https://www.bloomberg.com/news/newsletters/2021-05-12/hyperdrive-daily-car-crashes-have-a-black-box-problem (visited on 04/05/2021).

[83] Canturk Isci and Margaret Martonosi. "Runtime power monitoring in high-end processors: Methodology and empirical data". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE. 2003, pp. 93–104.

[84] C. Kao, S. Huang, and I. Huang. "A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 54.3 (2007), pp. 530–543. DOI: 10.1109/TCSI.2006.887613.

[85] Jagannath Keshava, Nagib Hakim, and Chinna Prudvi. "Post-silicon validation challenges: How EDA and academia can help". In: *Design Automation Conference*. IEEE. 2010, pp. 3–7.

[86] Jinwoo Kim et al. "Architecture, chip, and package co-design flow for 2.5 D IC design enabling heterogeneous IP reuse". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6.

[87] Donald E. Knuth. *Seminumerical Algorithms*. Addison-Wesley, 1981.

[88] Phil Koopman. *A Case Study of Toyota Unintended Acceleration and Software Safety*. 2014. URL: https://users.ece.cmu.edu/~koopman/toyota/koopman-09-18-2014_toyota_slides.pdf (visited on 06/01/2021).

[89] Alexandra Kourfali et al. "An Integrated on-Silicon Verification Method for FPGA Overlays". In: *Journal of Electronic Testing: Theory and Applications* 35.2 (2019), pp. 173–189.

[90] Ron Koymans. "Specifying real-time properties with metric temporal logic". In: *Real-time systems* 2.4 (1990), pp. 255–299.

[91] Saul Kripke. "Semantical considerations of the modal logic". In: *Studia Philosophica* 1 (2007).

[92] Joshua A Kroll et al. "Accountable algorithms". In: *U. Pa. L. Rev.* 165 (2016), p. 633.

[93] Anastasios Kyrillidis et al. "FourierSAT: A Fourier Expansion-Based Algebraic Framework for Solving Hybrid Boolean Constraints". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 02. 2020, pp. 1552–1560.

[94] J. C. Lagarias and A. M. Odlyzko. "Solving Low-Density Subset Sum Problems". In: 32.1 (Jan. 1985), 229–246. ISSN: 0004-5411. DOI: 10.1145/2455.2461. URL: https://doi.org/10.1145/2455.2461.

[95] Leslie Lamport. "Concurrency: The Works of Leslie Lamport". In: New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 9781450372701. URL: https://doi.org/10.1145/3335772.3335934.

[96] Leslie Lamport. "The temporal logic of actions". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), pp. 872–923.

[97] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: New York, NY, USA: Association for Computing Machinery, 1978.

[98] Kim G Larsen, Marius Mikucionis, and Brian Nielsen. "Uppaal tron user manual". In: *CISS, BRICS, Aalborg University, Aalborg, Denmark* (2009).

[99] Kim G Larsen, Paul Pettersson, and Wang Yi. "Model-checking for real-time systems". In: *International Symposium on Fundamentals of Computation Theory*. Springer. 1995, pp. 62–88.

[100] Doowon Lee and Valeria Bertacco. "MTraceCheck: Validating non-deterministic behavior of memory consistency models in post-silicon validation". In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2017, pp. 201–213.

[101] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *The Journal of Logic and Algebraic Programming* 78.5 (2009), pp. 293–303.

[102] B. Lisper and J. Nordlander. "A Simple and Flexible Timing Constraint Logic". In: Berlin, Heidelberg, 2012. ISBN: 978-3-642-34032-1.

[103] Chang Liu, M. Hicks, and E. Shi. "Memory Trace Oblivious Program Execution". In: *2013 IEEE 26th Computer Security Foundations Symposium* (2013), pp. 51–65.

[104] Wen-Hao Liu, Min-Sheng Chang, and Ting-Chi Wang. "Floorplanning and Signal Assignment for Silicon Interposer-Based 3D ICs". In: *Proceedings of the 51st Annual Design Automation Conference*. DAC '14. San Francisco, CA, USA: Association for Computing Machinery, 2014, 1–6. ISBN: 9781450327305. DOI: 10.1145/2593069.2593142. URL: https://doi.org/10.1145/2593069.2593142.

[105] Xinxin Liu and Scott A Smolka. "Simple linear-time algorithms for minimal fixed points". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1998, pp. 53–66.

[106] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. "Trace-based verification of imperative programs with I/O". In: *Journal of Symbolic Computation* 46.2 (2011), pp. 95–118.

[107] Oded Maler, Dejan Nickovic, and Amir Pnueli. "Checking temporal properties of discrete, timed and continuous behaviors". In: *Pillars of computer science*. Springer, 2008, pp. 475–505.

[108] Oded Maler, Dejan Nickovic, and Amir Pnueli. "Checking Temporal Properties of Discrete, Timed and Continuous Behaviors". In: *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*. Ed. by Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 475–505.

[109] Oded Maler, Dejan Nickovic, and Amir Pnueli. "From MITL to timed automata". In: *International conference on formal modeling and analysis of timed systems*. Springer. 2006, pp. 274–289.

[110] 2020 Marie Lewis Posted on February 7. *NASA Shares Initial Findings from Boeing Starliner Orbital Flight Test Investigation*. URL: https://blogs.nasa.gov/commercialcrew/2020/02/07/nasa-shares-initial-findings-from-boeing-starliner-orbital-flight-test-investigation/ (visited on 04/05/2021).

[111] Bojan Markovic et al. "A high-linearity, 17 ps precision time-to-digital converter based on a single-stage vernier delay loop fine interpolation". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 60.3 (2013), pp. 557–569.

[112] R. Massoud, H. M. Le, and R. Drechsler. "Property-driven Timestamps Encoding for Timeprints-based Tracing and Monitoring". In: *FORMATS-19*. 2019.

[113] R. Massoud et al. "Semi-formal Cycle-Accurate Temporal Execution Traces Reconstruction". In: *FORMATS-17*. 2017.

[114] R. Massoud et al. "Temporal Tracing of On-Chip Signals using Timeprints". In: *Design Automation Conference DAC-19*. 2019. URL: https://doi.org/10.1145/3316781.3317920.

[115] R. Massoud et al. "Time-stamps for Hardware Simulation Models Accurate Time-back Annotation". In: *5th Workshop on Design Automation for Understanding Hardware Designs (DUHDe-18)*. 2018.

[116] Rehab Massoud, Hoang M. Le, and Rolf Drechsler. "Property-Driven Timestamps Encoding for Timeprints-Based Tracing and Monitoring". In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Étienne André and Mariëlle Stoelinga. Cham: Springer International Publishing, 2019, pp. 41–58. ISBN: 978-3-030-29662-9.

[117] John McDermid et al. *Safer Complex Systems*. URL: https://www.raeng.org.uk/publications/reports/safer-complex-systems (visited on 08/21/2021).

[118] Bojan Mihajlović, Željko Žilić, and Warren J Gross. "Architecture-aware real-time compression of execution traces". In: *ACM Transactions on Embedded Computing Systems (TECS)* 14.4 (2015), pp. 1–24.

[119] Aleksandar Milenković and Milena Milenković. "An efficient single-pass trace compression technique utilizing instruction streams". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 17.1 (2007), 2–es.

[120] S Mitra, S. A. Seshia, and N. Nicolici. "Post-Silicon Validation Opportunities, Challenges and Recent Advances". In: *DAC*. ACM, 2010.

[121] *Molly Problem*. 2020. URL: https://www.itu.int/en/ITU-T/focusgroups/ai4ad/Pages/MollyProblem.aspx (visited on 04/05/2021).

[122] Gordon E Moore et al. *Cramming more components onto integrated circuits*. McGraw-Hill 38 (8), 114-117.

[123] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. "R 2 U 2 : monitoring and diagnosis of security threats for unmanned aerial systems". In: 2015.

[124] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[125] A. Nassar, F. J. Kurdahi, and W. Elsharkasy. "NUVA: Architectural Support for Runtime Verication of Parametric Specications over Multicores". In: *CASES*. 2015.

[126] *National Transportation Safety Board, Washington, DC 20594 , Safety Recommendation Report*. URL: https://www.ntsb.gov/investigations/AccidentReports/Reports/ASR1901.pdf (visited on 04/05/2021).

[127] N.Dunn, T.Dingus, and S. Soccolich. *(2019). Understanding the Impact of Technology: Do Advanced Driver Assistance and Semi-Automated Vehicle Systems Lead to Improper Driving Behavior? (Technical Report). Washington, D.C. AAA Foundation for Traffic Safety.* 2019. URL: https://aaafoundation.org/understanding-the-impact-of-technology-do-advanced-driver-assistance-and-semi-automated-vehicle-systems-lead-to-improper-driving-behavior/ (visited on 04/05/2021).

[128] Dejan Ničković and Nir Piterman. "From MTL to deterministic timed automata". In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 2010, pp. 152–167.

[129] Peter Øhrstrøm and Per Hasle. "Modern temporal logic: The philosophical background". In: *Handbook of the History of Logic*. Vol. 7. Elsevier, 2006, pp. 447–498.

[130] Joël Ouaknine and James Worrell. "On the decidability of metric temporal logic". In: *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE. 2005, pp. 188–197.

[131] F. M. De Paula. "Backspace: Formal Analysis for Post-Silicon Debug Traces". In: *Phd. Thesis, University of British Colombia*. 2012.

[132] Maja Pešić, Dragan Bošnački, and Wil M. P. van der Aalst. "Enacting Declarative Languages Using LTL: Avoiding Errors and Improving Performance". In: *Model Checking Software*. Ed. by Jaco van de Pol and Michael Weber. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 146–161. ISBN: 978-3-642-16164-3.

[133] S. Pinisetty and et al. "Runtime enforcement of timed properties revisited". In: *Formal Methods in System Design* (2014). ISSN: 1572-8102.

[134] David A. Plaisted and Steven Greenbaum. "A Structure-Preserving Clause Form Translation". In: *J. Symb. Comput.* 2.3 (Sept. 1986), 293–304. ISSN: 0747-7171. DOI: 10.1016/S0747-7171(86)80028-1. URL: https://doi.org/10.1016/S0747-7171(86)80028-1.

[135] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE. 1977, pp. 46–57.

[136] Amir Pnueli and Zohar Manna. "The temporal logic of reactive and concurrent systems". In: *Springer* 16 (1992), p. 12.

[137] Sandesh Prabhakar, Rajamani Sethuram, and Michael S Hsiao. "Trace buffer-based silicon debug with lossless compression". In: *2011 24th International Conference on VLSI Design*. IEEE. 2011, pp. 358–363.

[138] *Preliminary Investigative Findings Boeing 737 MAX, March 20, 2020*. URL: https://transportation.house.gov/imo/media/doc/TI%20Preliminary%20Investigative%20Findings%20Boeing%20737%20MAX%20March%202020.pdf (visited on 04/05/2021).

[139] *Preliminary Report Released for Crash Involving Pedestrian, Uber Technologies, Inc., Test Vehicle*. URL: https://www.ntsb.gov/news/press-releases/Pages/NR20180524.aspx (visited on 04/05/2021).

[140] Jean-François Raskin and Pierre-Yves Schobbens. "The Logic of Event Clocks - Decidability, Complexity and Expressiveness". In: *Journal of Automata, Languages and Combinatorics* 4.3 (1999), pp. 247–286.

[141] Jan Reich, Marc Zeller, and Daniel Schneider. "Automated evidence analysis of safety arguments using digital dependability identities". In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2019, pp. 254–268.

[142] Jan Reich et al. "Engineering of Runtime Safety Monitors for Cyber-Physical Systems with Digital Dependability Identities". In: *Computer Safety, Reliability, and Security*. Ed. by António Casimiro et al. Cham: Springer International Publishing, 2020, pp. 3–17. ISBN: 978-3-030-54549-9.

[143] *RISCV Trace Specifications*. URL: https://github.com/riscv/riscv-trace-spec/blob/master/riscv-trace-spec.pdf (visited on 04/05/2021).

[144] Amer Samarah et al. "Automated coverage directed test generation using a cell-based genetic algorithm". In: *2006 IEEE International High Level Design Validation and Test Workshop*. IEEE. 2006, pp. 19–26.

[145] César Sánchez et al. "A survey of challenges for runtime verification from advanced application domains (beyond software)". In: *Formal Methods in System Design* 54.3 (2019), pp. 279–335.

[146] Christoph Schmittner and Zhendong Ma. "Towards a framework for alignment between automotive safety and security standards". In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2014, pp. 133–143.

[147] Daniel Schneider, Eric Armengaud, and Erwin Schoitsch. "Towards Trust Assurance and Certification in Cyber-Physical Systems". In: *Computer Safety, Reliability, and Security*. Ed. by Andrea Bondavalli, Andrea Ceccarelli, and Frank Ortmeier. Cham: Springer International Publishing, 2014, pp. 180–191. ISBN: 978-3-319-10557-4.

[148] Daniel Schneider and Mario Trapp. "Conditional safety certification of open adaptive systems". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 8.2 (2013), pp. 1–20.

[149] Daniel Schneider and Mario Trapp. "Engineering Conditional Safety Certificates for Open Adaptive Systems". In: *IFAC Proceedings Volumes* 46.22 (2013). 4th IFAC Workshop on Dependable Control of Discrete Systems, pp. 139–144. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20130904-3-UK-4041.00037.

[150] Minjun Seo and Fadi Kurdahi. "Efficient tracing methodology using automata processor". In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.5s (2019), pp. 1–18.

[151] Glenn Shafer. "A Mathematical Theory of Evidence". In: Princeton University Press, 1976. ISBN: 978-0691100425.

[152] H. Shojaei and A. Davoodi. "Trace Signal Selection to Enhance Timing and Logic Visibility in Post-Silicon Validation". In: *ICCAD*. IEEE, 2010, pp. 168–172.

[153] Alex Simpson and Niels Voorneveld. "Behavioural equivalence via modalities for algebraic effects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42.1 (2019), pp. 1–45.

[154] Eshan Singh, Clark Barrett, and Subhasish Mitra. "E-QED: electrical bug localization during post-silicon validation enabled by quick error detection and formal methods". In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 104–125.

[155] C. Sinz. "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints". In: *CP*. Vol. 3709. Lecture Notes in Computer Science. 2005, pp. 827–831.

[156] Carsten Sinz. "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints". In: *Principles and Practice of Constraint Programming - CP 2005*. 2005. ISBN: 978-3-540-32050-0.

[157] "SoC debug infrastructure". In: *https://opensocdebug.org/*. 2020.

[158] M. Soos, K. Nohl, and C. Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *SAT*. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009.

[159] M. Soos, K. Nohl, and C. Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *SAT*. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 244–257.

[160] *Summary of the FAA's Review of the Boeing 737 MAX*. 2020. URL: https://www.faa.gov/foia/electronic_reading_room/boeing_reading_room/media/737_RTS_Summary.pdf (visited on 04/05/2021).

[161] *System Navigator Probe*. URL: http://www.mips.com (visited on 04/05/2021).

[162] Greenhill Systems. 2020. URL: https://www.ghs.com/products/supertraceprobe.html (visited on 04/05/2021).

[163] *The 2020 Wilson Research Group's Functional Verification Study*. URL: https://blogs.sw.siemens.com/verificationhorizons/2020/10/27/prologue-the-2020-wilson-research-group-functional-verification-study/ (visited on 04/05/2021).

[164] Risto Tiusanen, Timo Malm, and Ari Ronkainen. "An overview of current safety requirements for autonomous machines – review of standards". In: *Open Engineering* 10.1 (2020), pp. 665–673. DOI: doi:10.1515/eng-2020-0074. URL: https://doi.org/10.1515/eng-2020-0074.

[165] *To Build Trust in AI*. 2020. URL: https://fipra.com/update/crucial-to-strike-right-balance-between-ethics-regulation-to-build-trust-in-ai-tech/ (visited on 04/05/2021).

[166] Mario Trapp and Daniel Schneider. "Safety Assurance of Open Adaptive Systems – A Survey". In: *Models@run.time: Foundations, Applications, and Roadmaps*. Ed. by Nelly Bencomo et al. Cham: Springer International Publishing, 2014, pp. 279–318. ISBN: 978-3-319-08915-7. DOI: 10.1007/978-3-319-08915-7_11. URL: https://doi.org/10.1007/978-3-319-08915-7_11.

[167] Grigori S Tseitin. "On the complexity of derivation in propositional calculus". In: *Automation of reasoning*. Springer, 1983, pp. 466–483.

[168] Dogan Ulus and Oded Maler. "Specifying timed patterns using temporal logic". In: *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*. 2018, pp. 167–176.

[169] *U.S. audit report cites 'weaknesses' in FAA certification of Boeing 737 MAX*. URL: https://www.reuters.com/article/us-boeing-737max-idUSKBN2AO2P6 (visited on 04/05/2021).

[170] Vladimir Uzelac and Aleksandar Milenković. "Hardware-Based Load Value Trace Filtering for On-the-Fly Debugging". In: *ACM Trans. Embed. Comput. Syst.* 12.2s (May 2013). ISSN: 1539-9087. DOI: 10.1145/2465787.2465799. URL: https://doi.org/10.1145/2465787.2465799.

[171] Jeroen Van Cleemput, Bjorn De Sutter, and Koen De Bosschere. "Adaptive compiler strategies for mitigating timing side channel attacks". In: *IEEE Transactions on Dependable and Secure Computing* (2017).

[172] A. Vardy. "Algorithmic Complexity in Coding Theory and the Minimum Distance Problem". In: *STOC*. ACM, 1997, pp. 92–109.

[173] Marcell Vazquez-Chanlatte et al. "Time-Series Learning Using Monotonic Logical Properties". In: *18th International Conference on Runtime Verification (RV)*. 2018, pp. 389–405.

[174] B. Vermeulen and K. Goossens. "Debugging Systems-on-Chip". In: Springer, New York, 2014.

[175] M. Mitchell Waldrop. 2016. URL: https://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338 (visited on 04/05/2021).

[176] D. Walter, H. Täubig, and C. Lüth. "Experiences in Applying Formal Verification in Robotics". In: *SAFECOMP*. Vol. 6351. Lecture Notes in Computer Science. Springer, 2010, pp. 347–360.

[177] Xiayang Wang, Fuqian Huang, and Haibo Chen. "DTrace: Fine-grained and efficient data integrity checking with hardware instruction tracing". In: *Cybersecurity* 2.1 (2019), p. 1.

[178] Taimour Wehbe, Vincent Mooney, and David Keezer. "Hardware-Based Run-Time Code Integrity in Embedded Devices". In: *Cryptography* 2.3 (2018), p. 20.

[179] Georg Weissenbacher and Sharad Malik. "Post-silicon fault localization with satisfiability solvers". In: *Post-Silicon Validation and Debug*. Springer, 2019, pp. 255–273.

[180] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica Cambridge University Press*. 1910.

[181] Bruce Wile, John Goss, and Wolfgang Roesner. *Comprehensive functional verification: The complete industry cycle*. Morgan Kaufmann, 2005.

[182] Qiang Xu and Xiao Liu. "On signal tracing in post-silicon validation". In: *15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2010, pp. 262–267.

[183] Joon-Sung Yang and Nur A. Touba. "Improved Trace Buffer Observation via Selective Data Capture Using 2-D Compaction for Post-Silicon Debug". In: *TVLSI*. 2013.

[184] JS. Yang and NA. Touba. "Enhancing Silicon Debug via Periodic Monitoring". In: *Proc. of Symposium on Defect and Fault Tolerance*. 2008.

[185] Sergio Yovine. "Model checking timed automata". In: *School organized by the European Educational Forum*. Springer. 1996, pp. 114–152.

[186] Lotfi A Zadeh. "Review of a mathematical theory of evidence". In: *AI magazine* 5.3 (1984), pp. 81–81.

[187] DG Zakharov. "On the model checking of sequential reactive systems". In: Citeseer. 2016.