Property-driven Timestamps Encoding for Timeprints-based Tracing and Monitoring*

Rehab Massoud[1], Hoang M. Le[1], and Rolf Drechsler[1,2]

[1] University of Bremen, Bremen 28359, Germany
massoud,hle,drechsler@informatik.uni-bremen.de
[2] German Research Center of Artificial Intelligence DFKI GmbH, Bremen, Germany

Abstract. Timeprints are temporal regularly-logged signatures, describing a signal's temporal behavior. They have been recently used in on-chip signals tracing and temporal properties checking. Timeprints are generated by aggregations of encoded timestamps marking where signal changes took place. This paper describes different timestamps encoding mechanisms, and shows how some system's temporal properties can be used to create more efficient timestamps. The efficiency of a timestamps-encoding is introduced in terms of the number of collisions in the timeprints-reconstruction solution space. We show how using property-based timestamps encoding reduces the number of such collisions, leading to better chances capturing unexpected behaviors.

Keywords: Timeprints · Timestamps-encoding · Trace-cycles.

1 Introduction

In Real-Time (RT) and Cyber-Physical Systems (CPS), non-intrusive cycle-accurate execution tracing is at the top of designers and operators wish-list – if it is affordable. The barrier to accurate traces' logging is: first that these traces are generally infinite, and generated with very high rates; if we get enough ports to log them, storing and processing them are still inherently problematic. Practically, it is always possible to erase old traces that are not needed anymore. But still, the speed of today's ICs – reaching several Gigahertz – result in unmanageably huge logs quickly, making traces very tricky to store, even for few seconds. Second, the ports' capabilities are also limited by the number of pins that can be assigned in a chip and by the pads physical characteristics. Digital logging speeds can not exceed the maximum on-chip clock, so the amount of bits that can be logged per clock-cycle is constrained by the available logging pins. Similarly, it is hard to store useful duration of operation's data on on-chip trace buffers, due to their huge amount. Many systems-specific work-around techniques exist to provide accurate logs with relative efficiency, like [3,2,12,1], but they are very customized, and hence cannot be extended to any generic on-chip signal. Except for [12], all these methods are strictly limited to design-time,

* This work is supported by the DAAD, University of Bremen (SyDe graduate school and CRDF) and the BMBF grant SELFIE (grant no. 01IW16001).

as they require physical tracers and/or debuggers to be attached to the chip, and still incur huge logs per second, which makes continuously capturing normal operation periods non-achievable.

Timeprints have been introduced in [8] as a light-weight check-trace, logged all over the execution for a specific on-chip signal. Timeprints could be generated for a single or multiple signals, as per the designers/auditors choice. Each logged timeprint summarizes information about signals temporal behavior during a trace-cycle. Although timeprints do not have explicitly all the details, they still contain enough data to check exactly and accurately what took place on-chip in many cases. To obtain timeprints, the tracing task is divided into consecutive trace-cycles; and one *timeprint* is logged at the end of each cycle. Timeprints contain information about the exact timings of where the traced signal changed its value, and are generated by encoded timestamps aggregations at the designated change instances. The exact timestamps aggregation into a timeprint is a form of lossy compression, where the exact instances are embedded and need to be retrieved by a *reconstruction* process.

The timestamps-encoding used in generating the timeprints, contributes strongly to the uniqueness/ambiguity of the reconstruction, when retrieving the original timings from a timeprint. Some details about how they affect the tracing (trace-size and reconstruction effort) will be presented next in section 2. There we explain how timeprints are generated and the rule of timestamps encoding in generating them. Then, we give an overview of how temporal properties are used in the reconstruction. In this paper, we explore how these properties can be used to obtain more efficient timestamps' encoding. So, after the background, we give a formulation of the problem of timestamps-encoding generation, in section 3. After the formulation, we present the proposed timestamps-encoding generation algorithms (with and without properties) in section 4. The efficiency measures suggested to compare the different timestamps encodings is then introduced in section 5; and applied to a sample experiment in section 6 for illustrating the effect of using some properties. Finally the paper is concluded in the last section.

2 Background on Timeprints

Run-time verification and monitoring been thoroughly considered in the literature; a recent overview can be found in [4], for example. While Run-time Verification (RV) is capable of checking the temporal properties on-line while the system is in operation; it is limited to those specifications known and formalized at design time. Although some parameterized run-time verification techniques exist, like [10], they are still limited in the sense that they operate on a pre-configuration that is fixed a priori to the monitoring and tracing task itself. This means that after the events have already taken place, the trace at hand (if any) would contain data logged according to a pre-configuration. So, if the trace did not contain enough data about the root-cause of the encountered problem, and a new criteria or a specification that is suspected but was not configured to be traced before hand; there is going to be no way of checking the previous trace already at hand for those newly suspected properties. Of course the new

configuration can be implemented in future, where it can capture the suspected case, but there might be no guarantee that the captured suspected case is actually the case that took place in the past, and not just another bug/bad-case. In general if the problem is inconsistent –i.e. sporadic–, there could be no way to capture it again. The availability of a trace that keeps some evidence about executions that took place, and contains non-specified properties would help greatly in identifying such sporadic problems' root-cause.

Timeprints aim at providing specification-independent tracing [8], to enable checking a wide range of temporal properties; including those unknown at design-time. Despite such independence, timeprints can still make use of the known traced signal's temporal properties in the sense described in this paper without much affecting their capabilities to detect/debug unspecified behaviors.

Specifications of behaviors and temporal execution traces can be expressed using different available forms of temporal logic. Temporal properties of systems have been a topic for study from both the perspective of specification efficient pre-description for monitoring, like in [9,7] and from specifications mining and learning perspective, as in [6] and [13]. Here, we limit our focus to the temporal properties over a single trace-cycles for simplicity. Expressing them over generic periods is a subject of our next upcoming work. This focus is also reasonable for a first properties-based timestamps codes generation attempt. It also renders the properties description into a very intuitive and simple task. The drawback of course is the need for a translation layer between system level properties (over generic periods) and trace-cycle properties.

In the next subsection, we describe how timeprints are generated. Then, the temporal properties that could be utilized to obtained better timestamps encoding are defined.

2.1 Timeprint Generation

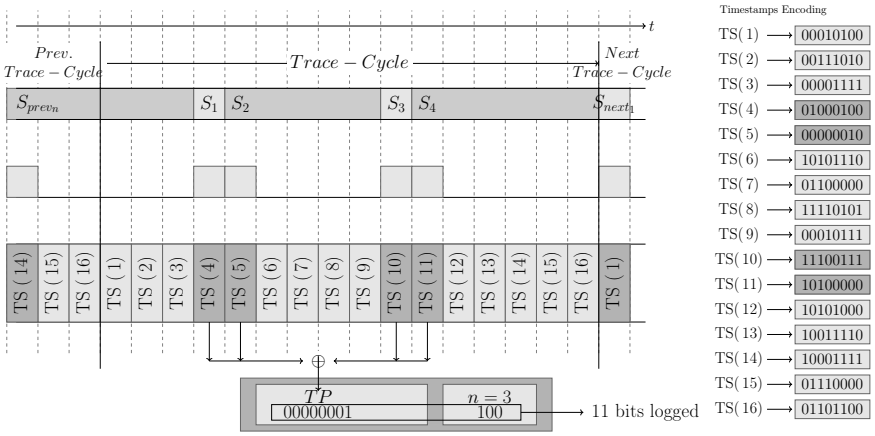

Fig. 1: An intermediate trace-cycle, with its respective timeprint

To generate a trace of timeprints of a signal, first: the continuous signal execution trace is divided into trace-cycles; where the first trace-cycle would start at reset or at a defined-check point, and the value of the timeprints is initialized to 0. A trace-cycle length m is defined before the system's deployment; i.e. before tracing starts. A sample intermediate trace-cycle is depicted in Fig. 1. Within a trace-cycle, an encoded time-stamp is assigned to every clock-cycle; for the i^{th} clock-cycle the corresponding code is denoted by $TS(i)$ in the figure. An example of a possible timestamps encoding is shown at the right of Fig. 1. A typical timestamps-encoding would contain m timestamps-codes, each of bit-width b_i, that can be fixed as in the figure. As the traced signal changes its value, a change marker triggers the aggregation of the corresponding timestamp code to the *timeprint TP*. In the example the aggregation function is XOR. So TP at the bottom of Fig. 1, is the result of XORing $TS(4)$, $TS(5)$, $TS(10)$, and $TS(11)$. At the end of each trace-cycle, the timeprint value that exists in the timeprint's register is logged; and the tracing continues: XORing the codes, where changes happen. The given encoding in the example is generated by checking some randomly generated codes, for linear independence from each other. Details of the generation algorithm will be given in section 4.

In this paper we fix the aggregation function to XOR, the trace-cycle length to m and the timestamps encoding (code) bit-width to b.

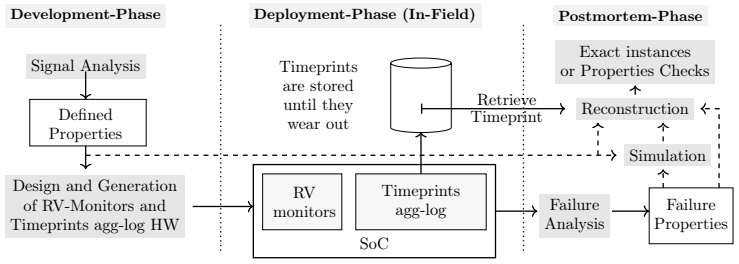

Fig. 2: Timeprints Life Cycle, from [8].

The decision about the trace-cycle characteristics (trace-cycle length, timeprints's width and timestamps encoding) happens at the the signal analysis phase, as at the top left of Figure 2. The signal analysis is also expected to result in the system's defined properties, from which the decision about which run-time monitors are going to be implemented is taken. A timeprint is an aggregation of timestamps that summarizes the temporal behavior. That is logged by aggregation hardware on-chip (within some System on Chip SoC) or attached to the ports/pins where the tracing is needed. During deployment, a change in the signal values triggers the corresponding timestamp aggregation into the timeprints. At the end of each trace-cycle, the fixed size timeprint is logged; together with the number of changes counted in the trace-cycle. The number of bits needed to log the number of changes is $\lceil log(m-1) \rceil$. This keeps the amount

of logging small and constant over time $(b + \lceil log(m-1) \rceil)$.[3] If a problem happened for which its root-cause need to be analyzed or for which an accurate trace is required, the relevant timeprints are retrieved. The failure analysis results in what we call *Failure Properties*, which expresses the visible problematic behavior of the system. To retrieve the accurate timing, (at postmortem) we retrieve exact instances of events from the timeprint via a *Reconstruction*, as in Figure 2. The reconstruction might use simulation to help aligning the timeprints to the system's visible behavior. All the optional paths are marked by dashed lines.

2.2 Temporal Properties

Timeprints are considered abstractions of the exact temporal execution; but the details lost by a timeprint's abstraction, are retrievable in most of the cases. We do not compromise accuracy during the aggregation process, as most traditional abstractions. Rather, we overlook data that are already known (verified) and hence can be used in the reconstruction. We describe those in terms of *Temporal Properties*, and add them to the reconstruction to decrease the ambiguity. For example, the details that can be retrieved by simulation and alignment to the timeprints trace are not considered lost, because simulation's input can be used in the reconstruction.

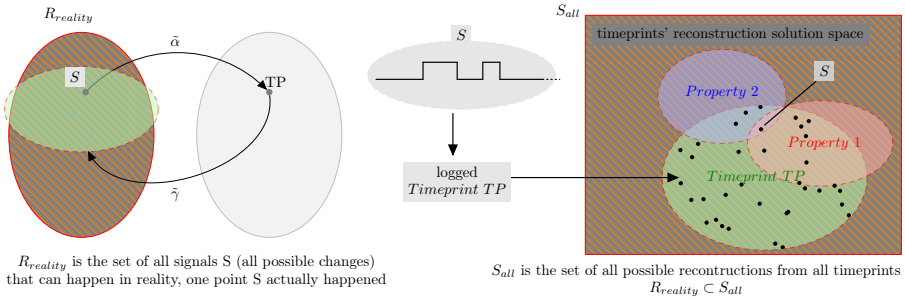

Fig. 3: Timeprints as Abstractions

Figure 3 shows the idea behind retrieval of the accurate timing via the reconstruction process; with the help of temporal properties. At the left, the timeprints reconstruction without properties is depicted. A point $S \in R_{Reality}$ corresponds to a specific timing changes in the traced signal that took place on-chip. We call each such a point *Signal* S. This signal would cause the aggregation of some encoded timestamps into the corresponding abstraction, or *Timeprint* TP. This

[3] If the signal change rate is known to be below certain limit the number of bits needed to describe the number of changes can still be less than $log(m-1)$. This $log(m-1)$ bits already covers the case of m changes as we aggregate timeprints recursively, i.e. the last timeprint of a trace-cycle is the initial value of the timeprint for the new trace-cycle; hence, if m=0 and the timeprint value changed, it means m changes took place, and if m=0 and the timeprint value is the same, then there has been zero changes in that trace-cycle.

aggregation and logging of the timeprint and the number of changes can be seen as a function $\tilde{\alpha}$ in a Galois insertion, and the reconstruction as $\tilde{\gamma}$, see [8] and the formulation section for more details.

Each point in the timeprints' reconstruction space (on the right of Fig. 3) corresponds to a possible accurate timing that could have led to the timeprint at hand. In the figure, one can see how the ambiguity (many possible accurate timings that could have led to the same timeprints) resulting from the reconstruction process is mitigated via properties. The exclusion of non-real solutions by properties-sets as in the figure corresponds to pruning the search space in the timeprints reconstruction space, as in Fig. 3. The number of solutions can be really huge for large trace-cycle sizes, if properties are not used [5]. This is why the properties usage proposed in [8], is essential to render the whole method acceptable. Ideally, as in Figure 3, the reconstruction which considers the properties ends up with a unique signal/timing (the intersection of the 3 sets). But this of course might not always be guaranteed; and defining metrics to judge the timeprints efficiency is under development.

In this paper, we suggest using the temporal properties, not only for reconstruction, but also for the generation of the timestamps encoding itself. For this purpose, we need to define briefly what do we mean by a property here. As the focus here is mainly about timing, properties in our context would be temporal properties that relates the timings of events happening within a trace-cycle. In general, if relations between events span more than one trace-cycle, they still can be mapped to some adjacent trace-cycles; so focusing on one trace-cycle does not limit the results presented here.

3 Formulation

The choice of the timestamps has influence on the ambiguity occurring within the logging procedure and thus on the time needed to reconstruct the original signal timings. Intuitively, a sparse choice of timestamps allows only for few possibilities to sum up to the timeprint. It decreases the number of the reconstructed solutions, making it easier to find all of them. However, we can only allow sparsity up to a certain extent as the number of logged bits would grow.

Ideally, we would choose an timestamp encoding that avoids ambiguity at all. This can be achieved by constructing an encoding $TS : [1..m] \to \mathbb{F}_2^b$, where $TS(1), \ldots, TS(m)$ are linearly independent vectors. Then, reconstructing from the logged timeprint would have a unique solution and it can be obtained quite fast. For example, an *one-hot encoding* would be of this type, and the timeprint would exactly correspond to the signal-changes themselves; i.e. zero solving time. However, choosing m linearly independent vectors requires that the dimension of $\mathbb{F}_2^b$ is m, hence $b = m$. But this means that the number of bits we need to log depends linearly on m, contradicting our goal to establish a space-efficient logging procedure.

The basic idea behind the timestamps encoding, is to achieve a trade-off in the choice of timestamps by requiring linear independence only up to a depth d. That means each subset of timestamps $T \subseteq TS([1..m])$ of size d is linearly

independent. As d grows, the number of solutions to the reconstruction problem decreases, but the number of logged bits b required increases. Computing TS with smallest b given m and d is still an open problem for future research. In this paper we give various algorithms, in the case $d = 4$ and approximate TS and b using a practical heuristic (see section 4).

In the next subsection, we give a formulation of the timestamps encoding (generation) problem before listing the algorithms.

3.1 Timestamps Generation Problem

The required timestamps encoding, as a target,is an ordered set of b-wide bitvectors of m elements. We denote this set be TS. An element of this set can be accessed by an index i, as $TS(i)$, where $TS(i)$ 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 $m \in \mathbb{N}$, which we simply call *signal*. A *signal* is a map $S : [1..m] \rightarrow \{0, 1\}$, where $S(i) = 1$ when a change takes place in the i-th clock-cycle. The logged timeprint TP, is the returned log entry (TP, k) from the aggregation function TP, where $TP = \sum_{i:S(i)=1} TS(i)$ and $k = |\{i \mid S(i) = 1\}|$, representing the number of changes in the signal. We enumerate all signals that represents all possible changes can happen in m-long trace-cycle by σ_m; hence all $S_i \in \sigma$.

Problem: Timestamps Generation 1 (TSG)
Input: trace-cycle length m, bit width $b \in N, log[m] < b < m$, property P.
Task: Find TS, 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 $TP(S_k) = \sum_{i:S_k(i)=1} TS(i)$.

Where a property P is a temporal property defined over S. Another variant of the TSG problem is the one that is limited in the choices of the possible timestamps to a predefined TS.

Problem: Timestamps Generation 2 (TSG)
Input: trace-cycle length m, bit width $b \in N, log[m] < b < m$, property P and input TS_{in}.
Task: Find $TS' \subseteq TS$, 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 $TP(S_k) = \sum_{i:S_k(i)=1} TS(i)$.

An example of the properties is linear independence of degree N; where the property P can be expressed as: that every signal S_i has exactly N ones; or $|S_i| = N$ has a unique timeprint TP. We denote linear independence of degree N by LI-N.

4 Timestamp Generation Algorithms

In this section, we introduce different practical approaches, which we used to tackle the timestamps encoding/generation problem.

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 encoded timestamps 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 the minimum (smallest) possible vectors (time-stamped codes) satisfying a set of conditions,
4) greedy algorithm: here an algorithm is presented for obtaining the set of fixed width b encoding timestamps, 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 timestamps 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 timestamps-encoding generation process.

4.1 SMT-based Time-stamps 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 timestamps-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 [11] 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:

$$\forall i, j, k, l, [TS_i \oplus TS_j \neq TS_k \oplus TS_l] \tag{1}$$

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

Similar conditions can be derived for higher N.

The resultant SMT instance is:

$$
\begin{aligned}
&(exists\ ((ts\_var\ (Array\ (\_BitVec3)(\_BitVec6))))\\
&(forall\ ((k\ (\_BitVec3))\\
&(l\ (\_BitVec3))(m\ (\_BitVec3))(n\ (_BitVec3)))\\
&(let((A1(and\ \ (not(=k\ \ l))\ \ (not(=n\ \ m))\\
&\qquad\qquad (=>(=k\ \ m)(not(=l\ \ n)))\\
&\qquad\qquad (=>(=l\ \ n)(not(=k\ \ m)))\\
&\qquad\qquad (=>(=k\ \ n)(not(=l\ \ m)))\\
&\qquad\qquad (=>(=l\ \ m)(not(=k\ \ n)))))\\
&(A2(not(=\ \ (bvxor\\
&(select\ \ ts\_var\ \ k)(select\ \ ts\_var\ \ l))\\
&\qquad\qquad (bvxor\\
&(select\ \ ts\_var\ \ m)(select\ \ ts\_var\ \ n))))))\\
&(=>A1\ \ A2))))
\end{aligned}
\tag{2}
$$

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 2). 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 GiB memory.

4.2 Random-based Time-stamps 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 1. 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.

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.

4.4 Greedy Algorithm

This algorithm is similar to the incremental algorithm. It starts from scratch and iterates over all possile timestamps in increasing order (i.e. treating them as integer values). Then, it greedly adds a new timestamp to the set of selected timestamps, if doing so does not violate the property under consideration (e.g. LI-4). Due to some optimizations such as look-ahead elimination of timestamps that are guaranted to violate the property, this algorithm is much faster than the incremental one. It also generates a maximum m timestamps that could be generated of width b, satisfying the property.

4.5 Properties Based Generation

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.

5 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 timestamps. 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 timestamps 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, [8], we can further increase m.

As in the formulation of the algorithms that encode the timestamps 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 timestamps, 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 timestamps 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 encoded timestamps 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.

5.1 Algorithms Run-time

In order to compare algorithms' run times (in Table 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 timestamps. Since the comparison is not fully possible due to how the 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 1, first rows gives six direct algorithms run-times. **Inc-index** in the first row is just using the index i of a timestamp $TS(i)$ as the coded timestamp. 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 timestamps 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 timestamps 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

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 1: Comparison between different Timestamps Encoding Algorithms, generating 1024 timestamps, or for an input of 1024 timestamps (in LI4-to-LI6)

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 timestamps 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 timestamps set. The generated sets are of smaller size because the timestamps that do not fulfill the LI6 condition are removed. Notice that the surviving timestamps from the Random-LI4 are much more than those surviving from other algorithms; but they also took much longer time to be generated.

5.2 Encoding with Properties

The tables below shows the number of reduced timestamps 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 timestamps (out of 1000) applying Pn are shown in table (a). Table (b)[4] shows the number of remaining timestamps 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 timestamps even for incremental codes.

		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

(a) Reduction in m for different P_n

	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

(b) The number of remaining timestamps in m after applying properties of different "constant delays (1,2,1,2) between 2 and 3 changes" consecutively

Table 2: Reduction in timestamps (a) and the number of remaining timestamps in reference to the original input set (b).

In table 2a, it could be seen how the greedy algorithm results still contains collisions when it comes to consecutive occurrences of changes.

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

6 Case-Study

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. 4. The car was equipped with three ultrasonic sensors and four servo motors for the brakes, one at each wheel[5].

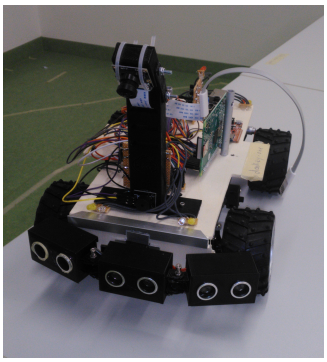

Fig. 4: Donkey Car

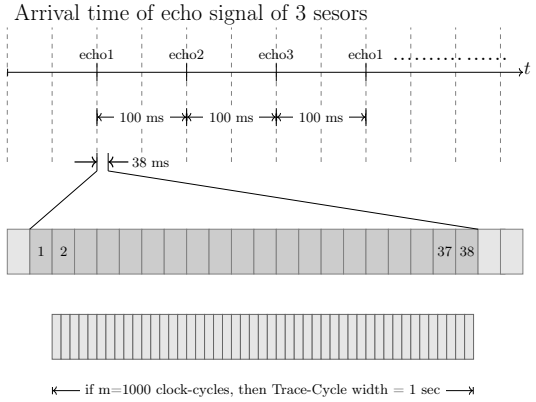

Fig. 5: Sensor data in a trace-cycle

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. 4, 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

[5] 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.

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. 5. 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 timestamps 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 timestamps-set size (trace-cycle length and timestamps 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∼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 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 timestamps saturates, and with keeping in memory removed timestamps-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 timestamps is checked at every stage, and is considered fulfilling the properties when all the properties-checks cannot remove any more timestamps 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 timestamps For trace-cycles of lengths from 200 maximum timestamps 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 timestamps fulfilling these properties is to use a list of randomly generated timestamps 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 timestamps 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).

7 Conclusion

We 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 timestamps 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 timestamps 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 co-inside, and avoided having them mapped to the same timeprint.

References

1. ARM CoreSight and ETM. http://www.arm.com (2018)
2. https://www.ghs.com/products/supertraceprobe.html (2018)
3. www2.lauterbach.com/pdf/main.pdf (2018)
4. Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Ničković, D., Sankaranarayanan, S.: Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications. Springer International Publishing (2018)
5. Chini, P., Massoud, R., Meyer, R., Saivasan, P.: Fast witness counting. CoRR **abs/1807.05777** (2018), http://arxiv.org/abs/1807.05777
6. Giantamidis, G., Tripakis, S.: Learning moore machines from input-output traces. In: FM 2016. Springer (2016)
7. Maler, O., Nickovic, D., Pnueli, A.: Checking Temporal Properties of Discrete, Timed and Continuous Behaviors, pp. 475–505. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
8. Massoud, R., Le, H.M., Chini, P., Saivasan, P., Meyer, R., Drechsler, R.: Temporal tracing of on-chip signals using timeprints. In: Design Automation Conference DAC-19 (2019), https://doi.org/10.1145/3316781.3317920
9. Mehrabian, M., Khayatian, M., Shrivastava, A., Eidson, J.C., Derler, P., Andrade, H.A., Li-Baboud, Y.S., Griffor, E., Weiss, M., Stanton, K.: Timestamp temporal logic (TTL) for testing the timing of cyber-physical systems. ACM Trans. Embed. Comput. Syst. **16**(5s), 169:1–169:20 (Sep 2017). https://doi.org/10.1145/3126510, http://doi.acm.org/10.1145/3126510
10. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R 2 u 2 : monitoring and diagnosis of security threats for unmanned aerial systems (2015)
11. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
12. Park, S.B., Hong, T., Mitra, S.: Post-silicon bug localization in processors using instruction footprint recording and analysis (ifra). TCADIC (2009)
13. Vazquez-Chanlatte, M., Deshmukh, J.V., Jin, X., Seshia, S.A.: Logical clustering and learning for time-series data. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 305–325. Springer International Publishing, Cham (2017)