Temporal Tracing of On-Chip Signals using Timeprints

Rehab Massoud*, Hoang M. Le†, Peter Chini‡, Prakash Saivasan*, Roland Meyer* and Rolf Drechsler*,†
*University of Bremen, †Technical University of Braunschweig, ‡DFKI GmbH, Germany

ABSTRACT
This paper introduces a new method to trace cycle-accurately the temporal behavior of on-chip signals while operating in-field. Current cycle-accurate schemes incur unacceptable amounts of data for logging, storage, and processing.

Our key idea to enable efficient yet cycle-accurate tracing, is to bring timing to the front as a main traced artifact. We split the signal tracing 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, we reconstruct the exact instances from a timeprint via a SAT query. The experiments demonstrate how unprecedented lightweight tracing can be applied, and how timeprints enable the verification of cycle-accurate properties and the detection of sporadic temperature effects.

ACM Reference Format:

1 INTRODUCTION
In late May 2018, the National Transportation Safety Board (NTSB) issued the preliminary report about the first instance of an autonomous car killing a pedestrian [3]. The report included information about the timing of when object identifications happened. The timings provided were very coarse; and if the delay that caused the crash was dependent on signals exchanged between modules, the exact timing of the signals’ firings would be crucial. Nowadays, reporting such events is carried out by the systems about themselves, as in space-telemetry or automotive-diagnostics and telematics. Accurate, independent and non-intrusive tracing of on-chip signals is needed to determine what happened exactly. This becomes even more vital for determining liability when signals are exchanged between systems provided by different suppliers.

In this paper, we target the scenario where a cycle-accurate post-mortem analysis is performed when an unexpected failure occurs during in-field deployment. On the cycle-accurate level, variant forms of Runtime Verification (RV) [9] and monitoring [10], 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 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: during deployment, RV can only cover a subset of the specified behaviors and none of the unspecified behaviors.

In summary, no existing approach can provide continuous cycle-accurate tracing that can cover unspecified behaviors during deployment. This is where the proposed method comes to serve. Our timeprints are designed to keep an independent, compressed, cycle-accurate temporal record/trace of what happened on chip. 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. Our contribution can be summarized as:

- a timeprints-based light-weight cycle-accurate tracing method, including a formulation and a reduction to SAT of the timeprint reconstruction problem, and
- an implementation of the approach using realistic experiments, showing how after-deployment cycle-accurate sporadic behavior tracing can be efficiently checked.
2 OVERVIEW

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 2 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. The Signal-Analysis step on the upper left side of Figure 3, shows this step at the design-phase. 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 2. 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 [10], or reason about combinations/history of them as in [11]. They can also be used in WCET analysis as in [12], and can even correct/enforce the behavior if the property is violated during run-time [13]. Due to on-chip space limitation, only a subset of the synthesizable monitors gets implemented. The darkest area in the Timeprints set in Figure 2, represents the definable properties which were undefined at design-time on the cycle-accurate level, and timeprints can enable tracing them.

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 Sections 3.3, 4.3 and 5.1. Like the RV monitors, the timeprints generator (timestamps aggregation and logging) is implemented in hardware, as in the middle of Figure 3. Light-weight timeprints are logged all the time during the execution, with constant rate and transmitted to some central database, see Figure 3.

The idea of timestamps aggregation was first introduced in [16], to compress the logged data into footprints. Instructions footprints-aided analysis was used in [18]. Periodic logging [24], analysis-based message-selection [8], and selective data capture and compaction [25], were all utilized to overcome the unacceptable accurate data logging requirements. These are all development-phase methods, that require on-chip trace buffers and/or many reruns.

When the system is running, hardware monitors (if used) will be checking the implemented defined properties satisfaction. Then, when an unexpected failure happens, or any doubts about certain point of operation in the past is raised, the respective timeprints are consulted in a postmortem-phase, as in the right if Figure 3.

To reconstruct the exact instances that can lead to the timeprint at hand, we encode the possible timestamps’ aggregations and the timeprint into an All-SAT problem. We then utilize the verified defined-properties about the execution in the SAT problem encoding to prune the huge search space. The properties already known to hold because the hardware monitors checking them indicate their satisfaction, can be encoded into the SAT-solver input. For example: missing deadlines, as a defined property, would be captured by RV; while smaller delays that might indicate a security threat [14], can skip such RV check. This RV check can still be used to prune the SAT-problem search space; then the reconstructed instances would show the slower delays. Dashed lines in Figure 3 indicate optional usage in the reconstruction step. Failure properties obtained from analyzing the failure that took place can also be encoded. This renders the reconstruction effort acceptable, and hence the whole method very practical.

Going back from a latest traced state, or check-point, using the formal description of the system, was used in [19]. But this formal reconstruction is neither time-accurate nor sporadic behavior preserving; because it requires several reruns. Formal reconstruction of scenarios that can lead to an existing failed state was also introduced in [6]; where a candidate simulation is provided as a possible failure scenario, but without any guarantee relating this scenario to what actually took place on-chip.

A timeprints’ trace enables reconstructing the exact instances, in a way that can show non-monitored delays, and/or give an evidence of a specific failure scenario or threat hypothesis. By encoding properties, the reconstruction process can directly check, potential simulation scenarios for plausibility or likelihood, as in the top-right of Figure 3. Timeprints can also in many cases prove or disprove exact properties about specific execution traces. This makes them serve as a potential witness, which can be consulted when needed, for transparent liability assignment. Timeprints are not meant to replace existing methods, rather they are inspected to reason about the details of cycle-accurate behavior when needed.

In the next section, we clarify by example the timeprints generation and reconstruction steps.

3 DIDACTIC EXAMPLE

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 \( S_1 \), sent by chip \( C_1 \) to chip \( C_2 \).
Thus, the amount of logged information depends linearly on the number of candidate signals from $2^b$ total, there are $2^{(b\cdot m)}$ change combinations that can lead to TP.

A common way to record the behavior of a signal like $S_i$ is to log the precise timings. This means we log a number with $\log(m)$ bits each time the value of $S_i$ changes. In Figure 4, 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.

### 3.1 Timeprint Logging

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 4, we identify the $i$-th clock-cycle by a timestamp $TS(i)$. 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 $S_i$ 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 4, we aggregate/add the timestamps $TS(4), TS(5), TS(10), \text{and } TS(11)$. 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 4, as an 8-bit vector. Note that modulo 2 addition is just bitwise XOR.

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 $k$.

### 3.2 Reconstruction Problem

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 4, there are more solutions besides the actual change-signal (we’ll call it for simplicity signal): $TS(4) + TS(5) + TS(10) + TS(11)$, for example, $TS(1) + TS(5) + TS(9)$, illustrated at the top middle of Figure 4. In total, there are 256 signals whose timestamps sum up to TP. The number of reconstructions (SAT-query solutions) 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 4. 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.

3.3 Temporal Properties
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 all 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. In our example, it was not an accident that we had at the end one possibility and we became sure of what accurately took place. 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 $i$.

To isolate the actual signal in the reconstruction problem, we utilize a couple of pairs (4.3 and 5.1.2 discuss timestamps encoding). However, sometimes we cannot exclude all the non-actual traces. For example, if one more cycle delay was allowed, the second possibility ($\alpha_2$ row, marked with $\square$) would have been a candidate.

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 4, 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.

4 FORMULATION AND ENCODING
We formalize the logging procedure, prove it to be 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 $\mathcal{P}(\text{Sig})$. The abstract domain is the powerset lattice over log entries, $\mathcal{P}(\text{Log})$. Fix an encoding $\text{TS} : [1..m] \rightarrow \mathbb{F}_2^m$. For the abstraction function, we apply the logging procedure to sets of signals. Let $\alpha_\text{TS} : \mathcal{P}(\text{Sig}) \rightarrow \mathcal{P}(\text{Log})$ be the lifting of $\alpha_\text{TS}$, defined by $\alpha_\text{TS}(F) = \bigcup_{S \in F} \tilde{\alpha}_\text{TS}(S)$, where $F \in \mathcal{P}(\text{Sig})$.

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

Both functions, $\alpha_\text{TS}$ and $\gamma_\text{TS}$ are monotonic by definition: We have that $\alpha_\text{TS}(F) \subseteq \alpha_\text{TS}(G)$ for $F \subseteq G$. Then $\gamma_\text{TS}$ is the lifting of $\gamma_\text{TS}$ to sets of signals, similar for $\gamma_\text{TS}$. Moreover, the functions establish a Galois insertion as shown in the next lemma. The proof follows from the above definitions.

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

4.2 Signal 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 $(TP, k) \in \text{Log}$ be the output of the logging procedure. Reconstruction all signals $S$ that get abstracted to $(TP, k)$ is the task of computing the preimage of $(TP, k)$ under the map $\alpha_\text{TS}$. We define the Signal Reconstruction problem as follows:

**Signal Reconstruction (SR)**

**Input:** Encoding $\text{TS} : [1..m] \rightarrow \mathbb{F}_2^m$, timeprint $TP \in \mathbb{F}_2^k$, $k \in \mathbb{N}$.

**Task:** Find all signals $S$ with $\tilde{\alpha}_\text{TS}(S) = (TP, k)$.

We can state an equivalent form of SR in terms of linear algebra. Let $A = [\text{TS}(1) \mid \ldots \mid \text{TS}(m)] \in \mathbb{F}_2^{m \times k}$ be the matrix consisting of all timestamps. SR is equivalent to finding all solutions $x \in \mathbb{F}_2^m$ of the system $Ax = TP$, where $x$ has exactly $k$ entries set to 1. Each solution $x$ represents a signal $S$ with $\tilde{\alpha}_\text{TS}(S) = (TP, k)$ and vice versa.

Variants of this problem are well-known in coding theory [22]. Moreover, SR is known to be NP-hard [7] and it is therefore unlikely that it can be solved in polynomial time.

To solve SR, we suggest a SAT-encoding. 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 [21].

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 $\text{TS}(i)$ 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 \( TS(i) \) 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 \( (m - 1) \) clauses, resulting in an intractable SAT query. We use the efficient and compact cardinality encoding proposed in [20], which introduces \( O(m \cdot k) \) additional variables and needs only \( O(m \cdot k) \) clauses to express the constraint.

4.3 Choice of Timestamps

The choice of the timestamps has influence on the ambiguity occurring within the logging procedure and thus on the time needed to solve SR. Intuitively, a sparse choice of timestamps 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 timestamp encoding that avoids ambiguity at all. This can be achieved by constructing an encoding \( TS : \{1, \ldots, m\} \rightarrow \mathbb{F}_2^d \), where \( TS(1), \ldots, TS(m) \) are linearly independent vectors. Then, the system \( Ax = TP \) has a unique solution and SR 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 \( \mathbb{F}_2^d \) 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 timestamps by requiring linear independence only up to a depth \( d \). That means each subset of timestamps \( T \subseteq TS(\{1, \ldots, m\}) \) of size \( d \) is linearly independent. As \( d \) grows, the number of solutions to SR decreases, but the number of logged bits \( b \) increases. Currently, we fix \( d = 4 \) and approximate \( TS \) and \( b \) using a practical heuristic. Computing an encoding with the smallest possible \( b \) is an open problem for future research.

5 APPLICATION

We first discuss timeprint design parameters, and how they affect the reconstruction time. Then two different experiments, showing how timeprints can be used, are presented.

5.1 Timeprint Design Parameters

5.1.1 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 number of changes \( k \) would increase, increasing the number of solutions, and the reconstruction time. Each row at Table 1 (to the left) 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.

5.1.2 Timestamp Encoding. In Random-constrained timestamp generation: timestamps are generated randomly, while checking for Linear Independence of depth 4 (LI-4). We use generic heuristics, which starts by smallest possible timestamp fulfilling LI-4; then we increment and check that the condition still holds. This encoding leads to smaller \( b \) and less average reconstruction times than the random-constrained encoding, see at the right Table 2.

5.1.3 Encoding Temporal Properties. As mentioned earlier, the additional encoding of temporal properties of a signal into SR 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, \( P_2 \): 2 consecutive timestamps would appear at least once, and \( D_k \): at least \( k \) changes happens before deadline \( D \).

Columns marked with c-SAT corresponds to solving time of SR (i.e. only using cardinality constraints c-cSAT). The columns marked with \( P_2 \) and \( D_k \) show the solving time under additional constraints imposed by \( P_2 \) and \( D_k \) 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, \( P_2 \) is less efficient than \( D_k \), because it is also a weaker property. Using \( P_2 \) and \( D_k \), together, reduces the solving time even more than \( D_k \) alone (column c + \( D_k + P_2 \)). 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. We can model properties defined in [15].

5.2 Experiments

5.2.1 CAN Bus Communication. 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 here, where timestamps are at the left:

<table>
<thead>
<tr>
<th>Time</th>
<th>Message</th>
<th>Data</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.257008s</td>
<td>GearBoxInfo(1020)</td>
<td>d</td>
<td>56</td>
</tr>
<tr>
<td>2.253525s</td>
<td>EngineData(100)</td>
<td>b</td>
<td>15</td>
</tr>
<tr>
<td>2.256312s</td>
<td>ABSdata(201)</td>
<td>d</td>
<td>5</td>
</tr>
<tr>
<td>2.260804s</td>
<td>IgnitionInfo(103)</td>
<td>d</td>
<td>6</td>
</tr>
</tbody>
</table>

If we name these messages as \( m_1, m_2, \ldots, m_4 \), 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):

<table>
<thead>
<tr>
<th>Time</th>
<th>Message</th>
<th>Data</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111111</td>
<td>GearBoxInfo(1020)</td>
<td>d</td>
<td>56</td>
</tr>
<tr>
<td>00000000</td>
<td>EngineData(100)</td>
<td>b</td>
<td>15</td>
</tr>
<tr>
<td>00000000</td>
<td>ABSdata(201)</td>
<td>d</td>
<td>5</td>
</tr>
<tr>
<td>00111111</td>
<td>IgnitionInfo(103)</td>
<td>d</td>
<td>6</td>
</tr>
</tbody>
</table>

The CAN bus idle state is 1, hence \( m_1 \)’s first bit 0 is the start bit, and then comes the ID \( (1020=011111111000000)=010000110011011111111111 \). The CAN bus idle state is 1, hence \( m_1 \)’s first bit 0 is the start bit, and then comes the ID \( (1020=011111111000000)=010000110011011111111111 \). The CAN bus idle state is 1, hence \( m_1 \)’s first bit 0 is the start bit, and then comes the ID \( (1020=011111111000000)=010000110011011111111111 \).

The CAN bus idle state is 1, hence \( m_1 \)’s first bit 0 is the start bit, and then comes the ID \( (1020=011111111000000)=010000110011011111111111 \). The CAN bus idle state is 1, hence \( m_1 \)’s first bit 0 is the start bit, and then comes the ID \( (1020=011111111000000)=010000110011011111111111 \). The CAN bus idle state is 1, hence \( m_1 \)’s first bit 0 is the start bit, and then comes the ID \( (1020=011111111000000)=010000110011011111111111 \). The CAN bus idle state is 1, hence \( m_1 \)’s first bit 0 is the start bit, and then comes the ID \( (1020=011111111000000)=010000110011011111111111 \).
6 CONCLUSION

The paper presented a novel Timeprints-based tracing methodology, which provides cycle-accurate temporal traces of on-chip signals execution. Being of extreme light-weight, Timeprints open the door for the first time, to a wide range of deployment-phase accurate timing-properties verification. Being consistent and cycle-accurate, provides a reliable transparent evidence for the actual temporal behaviors taking place on-chip, while operating in-field.

ACKNOWLEDGMENT

This work is supported by the DAAD, University of Bremen (SyDe graduate school and CRDF) and the BMBF grant SELFEI (grant no. 01IW16001). We also thank Khaled Naga (Avelabs) for his insights on CAN bus experiment.

REFERENCES