Efficient Tracing Methodology Using Automata Processor

MINJUN SEO and FADI KURDAHI, Center for Embedded Cyber-physical Systems, University of California, USA

Tracing or trace interface has been used in various ways to find system defects or bugs. As embedded systems are increasingly used in safety-critical applications, tracing can provide useful information during system execution at runtime. Non-intrusive tracing that does not affect system performance has become especially important, but unfortunately, the biggest obstacle to this approach was the vast amount of real-time trace data, making it challenging to address complex requirements with relatively limited hardware implementations. Automata processors can be programmed with a memory-like structure of automata and have a structure specific to streaming data, large capacity, and parallel processing functions. This paper promotes the idea of high-level system-on-chip monitoring using automata processors. We used a safety-critical pacemaker application in the experiments, described timed automata (TA)-based requirements, and tested intentionally injected 4,000 random failures. The TA model converted for Automata Processor to monitor system, correctness, and safety properties achieved 100% failure detection rate in the experiment, and the detected failure is reported as fast enough to allow enough extent for failure recovery.

CCS Concepts: • **Software and its engineering** → *Software verification*; Dynamic analysis; • **Computer systems organization** → Embedded hardware;

Additional Key Words and Phrases: Tracing, tracing methodology, runtime verification

ACM Reference format:
Minjun Seo and Fadi Kurdahi. 2019. Efficient Tracing Methodology Using Automata Processor. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 80 (October 2019), 18 pages.
https://doi.org/10.1145/3358200

1 INTRODUCTION

At the software development stage, verification/debug takes about 75% of the effort [15]. Despite these efforts, the deployed software released still contains bugs due to the fact that not all use cases may have been tested, and also because of the environmental and unpredictable nature of embedded systems, which can cause hardware failures. In a Cyber-Physical System, these bugs can sometimes affect system safety, which can result in life-threatening situations, so on-chip verification becomes necessary. If we can continue to monitor and verify software after deployment, we can detect when a safety-critical application fails in a timely fashion and take appropriate action. Thus, the use of tracing (or trace interface) for continuous software verification has emerged.

This work was partially supported by NSF grant CCF-1704859.
This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2019" of the camera-ready version of your paper.
Authors' addresses: M. Seo and F. Kurdahi, Center for Embedded Cyber-physical Systems, University of California, Irvine, California, USA, 92617; emails: {minjun.seo, kurdahi}@uci.edu.

1539-9087/2019/10-ART80 $15.00
https://doi.org/10.1145/3358200

Tracing typically produces information about the state of a pipeline in the microprocessor or information pertaining to instructions that have passed through the pipeline such as program counter, opcode, ALU results, memory access, and so on. Trace interfaces are either exposed to the outside or internal only. Externally exposed interfaces allow other devices to process or transfer information to external trace storage. When used internally, it usually stores tracing in the trace buffer, and the debug unit uses this information, halting microprocessor.

Due to the usefulness and real-time nature of the trace information, various methodologies have been developed to deal with trace interface directly. That is, another piece of hardware directly connected to the trace interface can directly verify trace data. This non-intrusive approach eliminates or minimizes software instrumentation and enables fast failure detection. Software instrumentation in embedded systems is likely to cause many problems in real-time performance, which can lead to abnormal behaviors of responsiveness and scheduling, mainly due to timing changes. Therefore, non-intrusive tracing can perform system verification without such changes of software behaviors with no performance penalty.

Separate independent hardware is suitable for handling tracing, but with constraints. First, it is a hardware speed problem when defining requirements. The information from tracing is instantaneous data and does not show any long-term view. Second, it is a size issue in hardware design. To perform runtime verification, it is necessary to maintain high-capacity requirements data. Because of this, the hardware that handles tracing internally requires a significant amount of high-speed memory. The cost of SRAM-level high-capacity/performance hardware also rises tremendously. Thus, we had to find a way to guarantee significant capacity, throughput, and parallelism.

Micron's commercialized automata processors can solve many of these problems. First, it provides DRAM-level high capacity. The current implementation provides a capacity of 512 MB, which enables the programming of a large number of automata. Second is fast processing speed. Memory with a DDR3 level clock is characterized by high-speed processing once the symbol is processed. Another essential feature is parallelism. Several independent automata running in parallel can be a crucial factor when monitoring the system.

In this paper, we propose a Trace Abstraction Layer (TAL) and show a methodology that can perform verification using Automata Processors. TAL's well-defined and independent layers make it possible to be used in a variety of ways to meet goals and circumstances. This methodology is illustrated through a pacemaker implementation, a life/safety-critical example. As far as we know, there are no standards in existence today for automata processor-based programmable non-intrusive runtime verification methodologies. This paper addresses the topic.

2 RELATED WORK

2.1 Trace-based Methods

Trace-based method for monitoring system execution uses instantaneous data of large volumes of real-time data. CoreSight [25], ChipScope [40], SignalTap [10], Intel Trace Hub [35], and RTNI [14] allow designers to specify a set of signals that can be traced at runtime without affecting the system performance.

Intel Trace Hub (TH) mainly controls tracing in a software way, which is performed by Software Trace Hub (STH), of how to store the trace data in a specified Memory Storage Unit (MSU), and the TH does not provide a way to perform runtime software verification.

ARM CoreSight supports a wide range of tracing/monitoring, including all cores and internal buses, but must be supplemented by software interactions. For example, to obtain a software execution trace of a specific address, we must specify the address as a trigger, and then the processor must stop and continue this analysis with the external device, which is an obstacle to perform runtime software verification without interruption or timing changes.

ChipScope is a non-intrusive technique to monitor components in an FPGA design using offline configurable triggers and trace filters. While ChipScope is suitable for hardware debugging purposes, it cannot be easily utilized for online verification of complex system requirements.

MED is a debug methodology for embedded systems, which integrates on-chip instrumentation (OCI) to support configurable triggers and traces. The collected traces from the OCI components are then combined and made accessible through a traditional off-chip JTAG interface. However, similar to other scan-based interfaces, the system must be partially or entirely halted to access the collected information, meaning real-time monitoring is not possible.

The real-time non-intrusive (RTNI) approach utilizes a trace-based approach for tracing specific signals within a microprocessor focused on testing, debug, and validation of real-time systems.

However, those trace methods typically require external hardware or dedicated processors to access the traced signals and only allow a small subset of signals to be traced.

2.2 Event-based Methods

Event-based methods (e.g., ARBD [29], MAMon [12], BusMOP [28], SOF [21], NIRM [33], DiaSys [37], Watanabe et al. [39], Zouh et al. [43], and RTAD [27]) uses hardware to detect events, rather than tracing all execution data. Thus, event-based monitoring requires analyzing a relatively small amount of data and results in simpler implementations that enable on-chip verification.

ARBD and MAMon monitor the system execution using on-chip probes to detect events and off-chip hardware to analyze detected events. ARBD uses an assertion-based analyzer implemented in an FPGA, whereas MAMon uses an external workstation for analysis. As these approaches use off-chip interconnections, the number of events that can be analyzed is limited by the interfaceâs bandwidth.

BusMOP is a more general monitoring and verification approach for commercial off-the-shelf (COTS) components. BusMOP enables designers to define which events are monitored and verify requirements based on those events. Requirements are defined using linear temporal logic. However, BusMOP is limited to only those events that can be observed from the system bus. Monitoring bus not only limits which events can be analyzed but also imposes delays in event detection in the presence to bus contention, which makes precise timing verification difficult in many circumstances.

SOF is an event-based detection and collection framework that integrates distributed event detectors for both hardware and software with pipelined interfaces for efficiently collecting events at a centralized location. The SOF approach primarily focuses on non-intrusive event detection and assumes event analysis can be performed using a secondary on-chip processor but does not consider the problem of runtime verification itself.

NIRM is non-intrusive runtime monitoring methodology with compact representation, which is a hierarchical runtime monitoring graph (RMG), of requirements. While the compact representation can reduce hardware area resources, there is still significant hardware used for requirement storage to support programmability, which is expensive to thousands of states for the requirement.

DiaSys proposes a methodology to transfer the self-contained diagnosis events to a host PC by reducing the bandwidth of trace data. The host PC supports complex software debugging tasks through the events. However, this DiaSys only focuses on offline analysis and is not suitable for real-time analysis of the execution environment.

Watanabe et al. reports a development workflow and presents practical design considerations. Requirements (contracts) are carefully written by verification experts in signal temporal logic (STL) and are used for reasoning the continuous behavior of a system over time, showing two advanced driver-assistance systems (ADAS). The approach is a software-based assertion check technique, which has over 1-second overhead that probably will not be accepted for safety-critical systems.

Zhou et al. propose a hardware-accelerated assertion checking unit (ACU) that is guided by software. Temporal assertions are stored in ACU hardware registers, and atomic proposition processing unit (APPU) continuously check whether input for a hardware component satisfies properties defined, showing hardware-assisted cryptographic system as an example. The approach reduces processor utilization via hardware-based assertion checking, but still requires the intervention of software and laboratory-level post analysis.

RTAD non-intrusively infers anomalous branches on a target application. In the offline learning stage, RTAD trains and optimizes machine learning (ML) model with dynamic simulations. In online inference stage, the ML model from the learning stage is programmed, and the GPU-inspired inference engine monitors and checks the target application execution through ARM CoreSight trace. RTAD is only concerned with branches, and ML-based detection might have false positives, and software intervention is required to handle many more factors such as timing, memory addresses together in ARM CoreSight-based trace.

The bigger problem is that there is no consistency and no standard among these interfaces (types of tracing data, width, and connectivity), which requires different implementations of the hardware to verify/monitor software execution.

2.3 Requirements for Runtime Monitoring

For runtime monitoring and verification, there are various models to specify requirements. Formal models, which use a mathematical framework, such as timed automata (TA) [4] and linear temporal logic (LTL) [30] are used to define a system.

Linear Temporal Logic (LTL) defines system states and temporal properties of the states, which can be verified at runtime to ensure that the properties remain true during execution. Real-time extensions to LTL, such as metric interval temporal logic (MITL), have been proposed to specify real-time requirements to support the concept of specific time intervals for LTLs.

Timed automata (TA) is another well-known formal modeling method that specifies how a system functions and models time constraints for transitions between system states. TA models are typically used in model checking techniques to evaluate properties such as reachability, safety, and liveness property. However, the TA model is limited its effectiveness for modeling system-level interactions, including interactions between hardware and software components.

Despite strong verification capabilities, formal verification methods such as TA and LTL have been mainly used for design-time verification or in a hardware form that cannot be synthesized and modified in advance. Also, runtime checkers using formal properties mostly use instrumentation such as STARVOORS framework [3] and CLARA [8], which is not suitable for time-sensitive systems such as embedded systems. This paper introduces programmable on-chip hardware runtime monitor as a part of runtime verification.

2.4 Automata Processor

Automata processor (AP) [11] is a hardware implementation of a non-deterministic finite state machine (NFAs) and has several additional features to aid in the use of existing NFAs. The AP is a programmable device and can perform high-speed, parallel, and reconfigurable data stream processing by exploiting low-level parallelism based on DRAM technologies. Such high-efficiency hardware is used for data mining [38], bioinformatics [31], machine learning [7, 34, 42], and network intrusion detection [32], and it was mainly used as an accelerator, achieving from 45 X to 3978 X speedups.

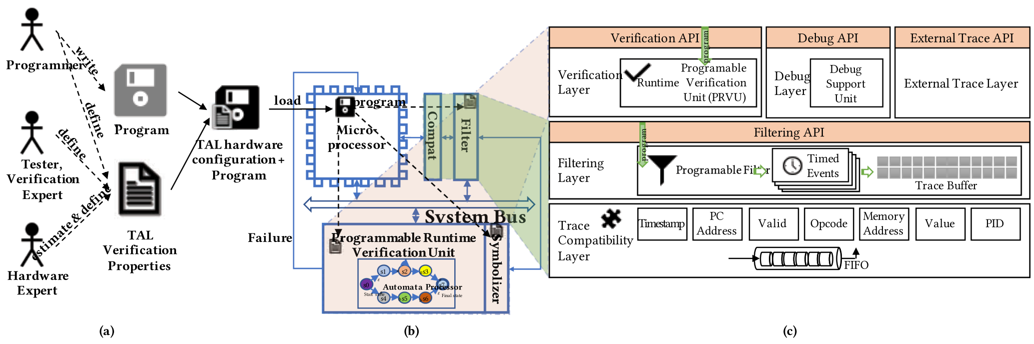

Fig. 1. Overview of a) Design-time Methodology and b) Runtime Methodology of Universal Tracing Methodology and c) Trace Abstraction Layer (TAL).

Runtime verification using the AP has not been studied, and as far as we know, this paper is the first attempt to achieve high-efficiency runtime verification using the AP. We believe that the AP's characteristics of high-speed parallel processing of streamed data are efficient in the processing of tracing.

3 UNIVERSAL TRACING METHODOLOGY

The universal tracing methodology shown in Figure 1 for monitoring and verification consists of: (a) a design-time methodology for developing software and constructing verification properties for runtime verification, (b) non-intrusive hardware for monitoring and verifying the properties at runtime within the deployed system. Figure 1(c) presents the Trace Abstraction Layer (TAL), consisting of several functionally/logically separated components which are: (1) compatibility layer, (2) filtering layer, and (3) verification/debug/external trace layer for universal tracing. Note that Figure 1(b) illustrates an instance of TAL as a hardware implementation, and the highlighted part of the instance points to the corresponding TAL layer.

3.1 Compatibility Layer

The compatibility layer at the lowest layer ensures compatibility with different processors. Existing microprocessors provide different trace interfaces for various vendors. Because these interfaces are not compatible with each other and the bits, types, and numbers of signals are different from each other, there is a problem that the hardware using trace interface is also incompatible across platforms. To ensure compatibility, we must first identify the essential elements of monitoring/verification.

Table 1 shows the characteristics and signal types of the trace interface of representative softcore processors [2, 20, 41]. The common signals of these processors are ***PC address*** and ***Opcode***, which is the minimum required set of signals for the compatibility layer. For runtime on-chip verification, let us take a closer look at the necessary elements in the hardware. Consider the well-known formal verification methods, timed automata (TA):

- Finite set C (clock of a timed automaton)
- Clock reset

Other notable software flow elements are: basic block, jump target, entry and exit block, and execution order pair.

The key to verification TA is the time factor, which is a ***time_tag***. A basic block can be represented by a pair of addresses, which is expressed as a pair $BB = (Addr1, Addr2)$. Jump target, entry

Table 1. Types of Interface and Trace Signals for Representative Softcore Processors and Common Trace Signals

Microprocessor	Interface	Trace signals	Comment
Leon3	Internal	Multi_cycle, Time_tag, LoadStore_parameter, Program_Counter, Instruction_Trap, Error_Mode, Opcode	32-bit time tag (cycle)
Microblaze	External	Trace_Instr_Valid, Trace_Instruction, Trace_PC, Trace_Reg_Write, Trace_Reg_Addr, Trace_MSR_Reg_11, Trace_MSR_Reg_15, Trace_PID_Reg, Trace_New_Reg_Value, Trace_Exception_Taken, … (***omitted due to page limitation***)	I$, D$ information
OpenRISC	External	traceport_exec_valid_o, traceport_exec_pc_o, traceport_exec_jb_o, traceport_exec_jal_o, traceport_exec_jr_o, traceport_exec_jbtarget_o, traceport_exec_insn_o, traceport_exec_wbdata_o, traceport_exec_wbreg_o, traceport_exec_wben_o,	mor1kx implementation
Common trace signal		Program Counter (PC), Opcode	

Table 2. Verification Operators and Corresponding Trace Signals

Verification operators	Corresponding Trace Signal
Temporal operators, clock	Time tag
Basic block, entry and exit block, execution order pair	PC address
Variables in verification	Memory address and value

block, exit block can also be represented by PC addresses. It is also required to check variable values to evaluate certain conditions such as checking whether the logic holds a value globally or checking the transition condition in the TA. Table 2 represents verification operators and corresponding trace signals discussed above.

For platforms that do not support a specific signal, the compatibility layer generates such signals. For example, MicroBlaze does not support ***time_tag***, so an internal counter in the compatibility layer can provide a counter, performing time counting. Also, when a memory operation is being performed, the Leon3 does not directly indicate memory address and value, but addresses and values can be obtained indirectly from ***Opcode*** and ***LoadStore*** parameters. The information generated or obtained from this layer is passed to the upper layer, the filtering layer.

3.2 Filtering Layer

Since tracing is a cycle-by-cycle method, too much information can be flooded into verification hardware. For example, in the case of a 100 MHz Leon3 running on FPGA, 12.8 GB trace data are produced per second. Therefore, the filtering layer focuses on filtering this data and delivering only the necessary information to the next layer. The information is referred to as an event with ***time_tag***.

Definition 3.1. Filter F is a set of a tuple $< E, P, V >$ where:

- E is an event, where each event represents a desirable trace to be collected.
- An event has a type $\in \{Addr, memWrite, memRead, period\}$
- P is a list of parameters for the event E (size of 2).
- V is the target value for the type.

The ***Addr*** event triggers when one specified address as a parameter is equal to the current PC address with an address (1st parameter) and operator (2nd parameter), ignoring target value ***V***. The ***memWrite*** and ***memRead*** events have two parameters: an address of a variable, and operator to compare with target value V, which are handled when the memory content of the address is being read or written. The period generates an event every ***V*** cycle. Figure 2 shows an example of defining events and setting filters. The events $E0 - E3$ will be handled when the addresses, which are **0x80002FFF, 0x80004000, 0x80004020, 0x80004258** respectively, are hit. The event E4 or E5

```
(E0(Addr), [0x80002FFF, -], -)              // address
(E1(Addr), [0x80004000, -], -)              // address
(E2(Addr), [0x80004020, -], -)              // address
(E3(Addr), [0x80004258, -], -)              // address
(E4(memRead), [0x40000000, “>”], 1000)      // value is gt
(E5(memWrite), [0x40000000, “>”], 1000)     // value is gt
(E10(period), [-, -], 1000)                 // every 1000 cycle
```

Fig. 2. Example programmable filter setting.

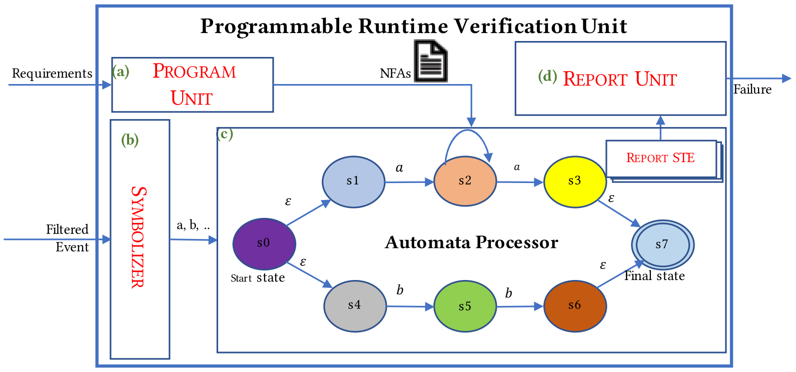

Fig. 3. Detailed view of programmable runtime verification unit (PRVU).

is handled when the content of memory location **0x40000000** is read or written and greater than value 1000, respectively. The event E10 generates an event every 1000 processor cycles. When an event defined in the filtering layer is detected in the trace interface, the trace is stored in the trace buffer with a timestamp and used by the upper layers. Note that trace data not related to the filter in this layer is discarded. The filter can be set via the filtering API shown in Figure 1.

3.3 Verification Layer

The verification layer provides a variety of primitives for non-intrusive monitoring. This layer contains the Programmable Runtime Verification Unit (PRVU), which can be reconfigured at the runtime. While typical formal verification methods perform symbolic analysis through exhaustive methods usually done offline, runtime monitors determine whether the property is satisfied at runtime, which is usually accompanied by instrumentation with its performance penalty. Our approach is based on non-intrusive runtime monitoring. In order to satisfy general formal verification and apply it to runtime verification, we need to define requirements, and a widely known and used timed automata (TA) was chosen for these requirements languages. Although PRVU can be implemented in various forms, this paper shows an implementation of PRVU through Automata Processor (AP) [11] with TA-based requirements for runtime verification.

The PRVU consists of the following functional blocks: a) Program Unit, b) Symbolizer, c) Report Unit, and d) Automata Processor as shown in Figure 3. The Program Unit (PU) programs AP with the requirements written in NFAs. The PU is a virtual/physical block in this diagram, either a software block that supports program loading via the AP SDK, or a hardware block that loads automata via DMA. The Symbolizer is a programmable unit that converts the filtered traces into symbols and delivers them to the AP. The AP performs runtime verification based on the programmed requirements, and if the defined STE is triggered, it passes it to the Report Unit (RU). The RU performs a failure report by reading the output event buffer of AP, which can be connected to the interrupt port of the microprocessor.

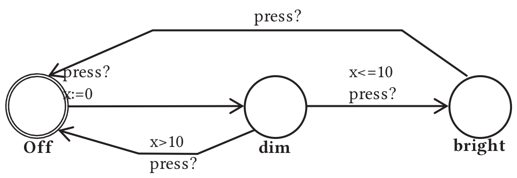

Fig. 4. Example of a timed automata model for a timer light switch.

3.4 Debug and External Trace Layer

The information stored in the trace buffer can be used for debugging. In this case, an ***Addr*** event is defined with breakpoint as an address through Filter API. The processor must be halted for debugging, which sends trace data (snapshots) from the breakpoint to the software debugger through the ***Debug API*** and Debug Support Unit (DSU). The sent snapshots are used for user interaction and debugging. The interaction can be done in cooperation with a software-level debugger. External traces simply transfer the contents of the trace buffer to hardware external to the microprocessor under the control of ***Trace API***. Note that this paper will only focus on the verification layer and its details.

4 TIMED AUTOMATA

We use Timed Automata (TA) model to specify requirements for runtime verification. Although the TA can be extracted automatically [23], this paper assumes that the system verification expert has well described the TA model for the application. The TA model defines a set of states to be observed at runtime, and this set can contain various valid execution orders, execution time constraints, bounds on variable values, and so on.

Definition 4.1. A timed automaton is a tuple $A = (\sum, L, L_0, C, F, E)$ where:

- $\sum$ is a finite set called the alphabet or action of A.
- L is a finite set consisting of the locations or states of A.
- C is a finite set called the clocks of A.
- $L_0 \subseteq$ is the set of start locations.
- $F \subseteq L$ is the set of accepting locations.
- $E \subseteq L \times \sum \times B(C) \times P(C) \times L$ is a set of edges, called transitions of A, where
  $-B(C)$ is the set of clock constraints related clocks from C
  $-P(C)$ is the powerset of C.

An edge (l, a, g, r, l') from E is a transition from location l to l' with action a, guard g and clock resets r.

Figure 4 is an illustrative example of a TA model, a timer light switch, where x represents the clock. As you can see in this example, the TA allows you to represent a state machine containing a time element. This paper also discusses how the TA model can be closely linked to the system.

Examples of TA models describing timing constraint, execution order, and memory value for runtime verification are shown in Figure 5. Figure 5(a) shows an example in which the start and end times of Task A are detected through the TA clock x, (b) shows the TA model that can verify the sequence between tasks, which shows a situation similar to a nested function call that waits for Task B to terminate execution in Task A, and (c) shows an example of verifying a specific value in a memory read or write, where a value of 1000 or greater is read or written to the variable a.

Currently, this paper assumes that the verification expert designs the TA model, but it is also possible to automate it with various techniques such as requirement mining [23].

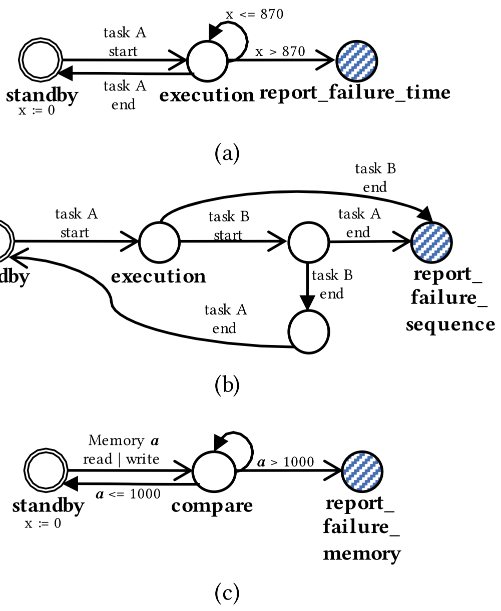

Fig. 5. Examples of timed automata model for runtime verification of (a) timing, (b) execution sequence, and (c) memory value.

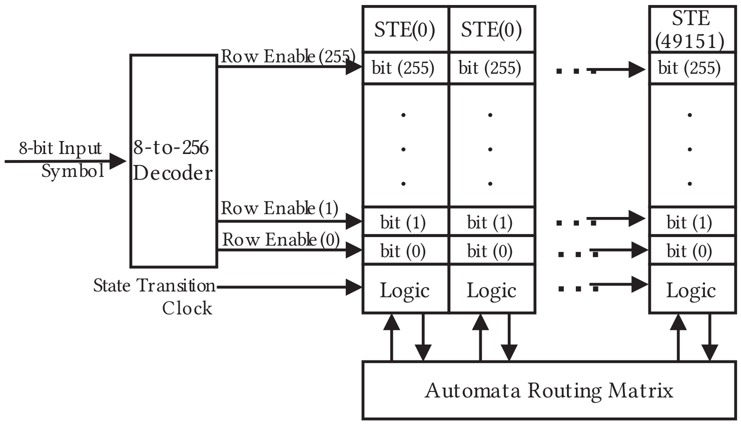

Fig. 6. The automata processor architecture.

5 AUTOMATA PROCESSOR FOR RUNTIME VERIFICATION

Automata Processor (AP) is an in-situ memory-based computational architecture that accelerates non-deterministic finite automata (NFAs). One of the critical features of the AP is that it supports the NFA, which has a much smaller size than the deterministic model.

In Figure 6, the AP receives 8-bit symbols and accesses the corresponding state transition element (STE). STE is the core component designed as a memory array that models the state of NFAs and has control and calculation logic. A single STE logic bit is provided to each column of the memory array used to activate the input and to the output decoder/driver. The value is set to (1) if the STE is active (0) or otherwise.

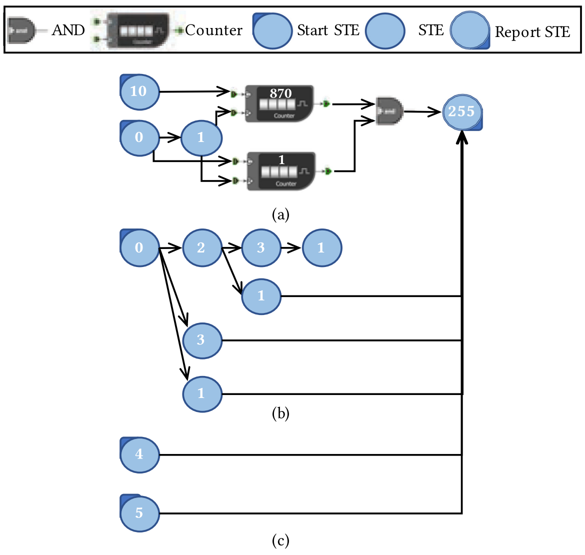

Fig. 7. AP model corresponding to the TA model in Figure 6.

The output is driven by the logical AND of the status bits in the relevant column of memory and the output and is output only if the selected bit is programmed to recognize the input symbol. A STE can match 8-bit custom symbols in clock cycles, and the STEs can activate each other through a reconfigurable routing network. Each STE is designed to recognize input data values and can be any character class for 8-bit symbols. These STEs are reconfigurable and can be reprogrammed to recognize new input data values.

Each STE has three states: inactive, active, and matched. Only the activated STE can be used for being matched with the following input symbols. Once a symbol is matched, the STE will accept the next input and match the programmed symbol match. The STE that indicates the starting state is called the starting STE. A STE that shows the acceptance status is called the reporting STE. Multiple start states are allowed to permit parallel execution of multiple NFAs.

The current generation AP supports 8-bit size symbols, and we use those for runtime verification and representing up to 256 different events. In Section 3.2, we have seen how to define programmable filters and use them to get refined traces. To use this in an AP, a kind of adapter is needed. That is, a mapping between a filtered event and a symbol is required, which is handled by the ***Symbolizer***.

Figure 7 shows the merged AP models corresponding to TA models (a), (b), and (c) in Figure 6 with a filter defined in Figure 2 and symbols represented in Table 3: For model (a) A total of two counter elements and a ***AND*** logic are used to verify one timing. The first counter tracks the timing constraint and the second checks whether the task is started or not. (b) To verify the execution sequence in model (b), all non-correct path events are considered failures. (c) For model (c), memory events use the value in the filter directly to determine the failure. Note that symbol 255 is used to report the failure. It can also be a model for explaining how the AP model supports parallelism. The activation of STE 0 and 1 affects the two automata simultaneously, which means that one event can update the state of two or more automata simultaneously.

Currently, the conversion from TA to AP model must be performed manually by verification exports. These manual works include the model transform as shown in Figure 7, and the type of

Table 3. Defining a Filter for Events and Mapping Information of the Symbolizer for AP model

Property	Filter	Symbol
Task A start	(E1(Addr), [0x80002FFF, -], -)	0
Task A end	(E2(Addr), [0x80004000, -], -)	1
Task B start	(E3(Addr), [0x80005000, -], -)	2
Task B end	(E4(Addr), [0x8000521F, -], -)	3
Memread(*a*) >1000	(E5(memread), [0x40000000, ">"], 1000)	4
Memwrite(*a*)>1000	(E6(memwrite), [0x40000000, ">"], 1000)	5
Periodic 1 ms	(E7(period), [-, -], 1000)	10
Report Failure	-	255

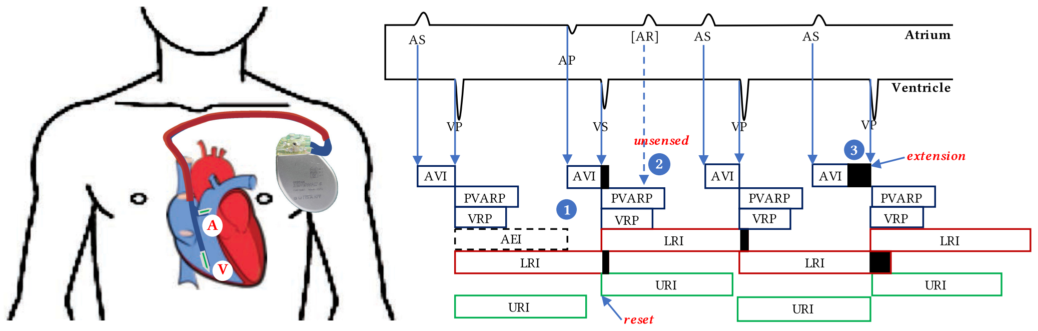

Fig. 8. Detailed timing diagram of a Pacemaker [17][18] (A: Atrium, V: Ventricle, S: Sensing, P: Pulsing).

reporting STE according to failure type must also be manually connected. The automation of this model transformation is beyond the scope of this paper.

6 EXPERIMENTAL RESULT

We have developed a system model framework based on gem5 [6] simulator's ARM processor-based full system simulation and automata processor simulator for a safe-critical embedded application modeling a network connected pacemaker [5]. At the same time, we also created an FPGA prototype to get hardware area overhead and power consumption. However, since it is impossible to create hardware with the same efficiency as an AP in FPGA, we use it only for approximate reference to extract usage of hardware resources.

In this work, a 1 GHz ARM processor with L1 cache (16 kB for data and 16 kB for instructions), L2 cache (shared 1 MB), and 1 GB of memory was used for gem5 simulation. The simulation on gem5 was performed with full-system simulation, and the experiment was performed assuming that the trace file collected after each simulation is the same as the real-time trace from the microprocessor. The RTEMS microkernel [1] was used as an OS for the purpose of providing accurate timers, implementing POSIX pthreads, and controlling synchronization.

Our embedded application is a network connected pacemaker supporting physician-configurable pacing configurations, communication of cardiac activity to a home monitoring device, and emergency physician notification for significant cardiac events. Figure 8 represents the detailed behaviors of a pacemaker. The Post Ventricular Atrial Refractory Period (PVARP) being initialized after ventricular event filters noise that causes undesired pacemaker behavior in the ventricular channels (marker 1). The Atrio-Ventricular Interval (AVI), which is the time between atrial event and ventricular events, is used to keep the appropriate delay between ventricular and atrial activity. The Ventricular Refractory Period (VRP) performs noise filtering

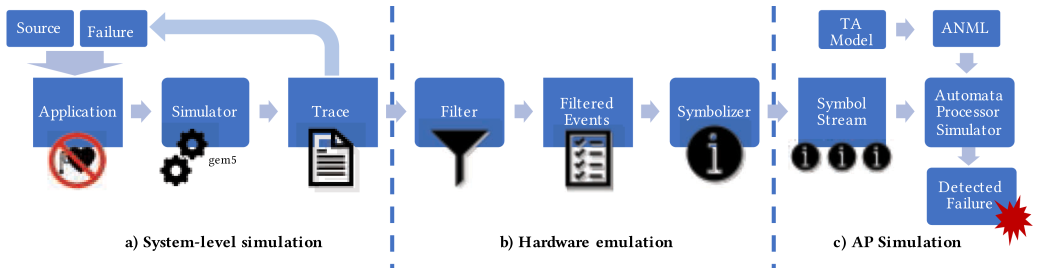

Fig. 9. Detailed steps of experimental setup.

to prevent undesired behavior from the atrial channel. The lowest rate interval (LRI) begins with ventricular pacing or sensing and counts the time until either atrial pacing (AP) or no atrial sense (AS) during atrial escape interval (AEI) (marker 1). In case that upper rate interval continues after AVI, ventricular pacing (VP) is delayed until the upper rate interval (URI) ends (marker 2). Besides, the pacemaker connected to the network, and it performs device setting by a physician and reports data to the physician.

Failure can be defined as the inability of a system to continue processing due to erroneous logic. We consider several common failure cases including timing failures [9], execution sequence failures [19], synchronization failures [22] that is a combination of timing and execution sequence, and memory value failure [24].

Figure 9 shows the overall process of this experiment consisting of a) system-level simulation, b) hardware emulation, and c) AP simulator stage. The system-level simulation is performed in the gem5 simulator after compiling the application source code with failures. The automated script generated the source codes with a total of 4000 failures, which is 1000 per each type of failure. To mimic the hardware trace interface, this experiment uses the trace file that the system created after the application is executed. In the hardware emulation step, the massive trace is filtered through a programmed filter, which generates filtered events. The symbolizer maps the filtered events and symbols. Finally, the mapped symbols are passed to VASim [36], an AP simulator, in the form of stream, and the AP simulator runs NFAs from automata network markup language (ANML) based on timed automata (TA) model through these symbols. If a failure is detected, it is reported through report STEs. In this experiment, the TAL approach achieved a 100 % failure detection rate for all types of failures.

The pacemaker implementation is written in C language, and each hardware component is configured as a thread to implement as much as hardware configuration, and this thread is controlled through synchronization mechanisms such as mutex, semaphore, and conditional variables. Table 4 shows the elements used for runtime verification in the pacemaker implementation and the required number of events for the verification.

A list of requirements is shown in Figure 10. System correctness, functional correctness, and safety property represent properties for runtime verification in different points of view. These properties can also affect other properties. For example, if the system correctness property does not satisfy the requirement, it affects the functional property and even the safety property. Likewise, if one of the functional correctness properties is violated, it also affects the safety property.

Based on this requirement, the automata representation for the AP system is shown in Figure 11. It was created with ANML definitions, took 0.23 seconds of compile time to generate 528 STEs, 3 reporting elements, 32 counters, and 31 logics. Among the resources of the AP, 32% symbols were used in 84 symbols, and about 1.25% of STEs, 4.1% of counter elements, 1.3% of Boolean logic elements, and less than 1% in reporting elements. The event-symbol mapping (32%) can be

Table 4. Functions and Variables Used in Runtime

	Function and offset	Variable	
	Execution order, Timing	Value	Update order
Name	`Check_buffer`	`AVI_ms`	`time_counter`
	`Event`	`VAI_ms`	`time_rate1`
	`Ventri_thread`		`time_rate2`
	`Atrial_thread`	`VRP_ms`	`heart_rate`
	`PVARP_thread`		`token`
	`VRP_thread`	`PVARP_ms`	
	`AVI_thread`	`heart_rate`	
	`VAI_thread`	`token`	
	`Sensed_Atrial`	`AVItime_recv`	
	`Sensed_Ventri`	`VAItime_recv`	
	`Doctor_config`		
	`Wait_for_connection`		
	`main`		
	`System_init`		
	`POSIX_Init`		
# Events	58	16	10

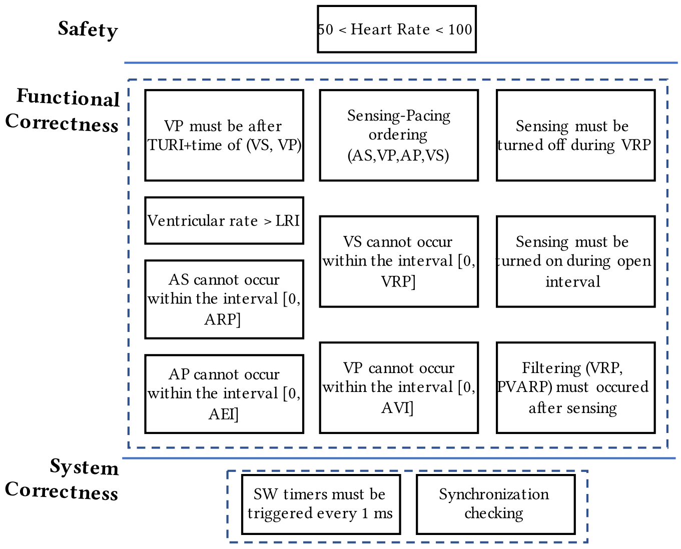
Safety
50 < Heart Rate < 100
Functional Correctness
VP must be after TURI+time of (VS, VP)
Sensing-Pacing ordering (AS,VP,AP,VS)
Sensing must be turned off during VRP
Ventricular rate > LRI
VS cannot occur within the interval [0, VRP]
Sensing must be turned on during open interval
AS cannot occur within the interval [0, ARP]
AP cannot occur within the interval [0, AEI]
VP cannot occur within the interval [0, AVI]
Filtering (VRP, PVARP) must occured after sensing
System Correctness
SW timers must be triggered every 1 ms
Synchronization checking

Fig. 10. Details of system, correctness, and safety properties in Pacemaker example.

regarded as a weak point by using a relatively small number of 256 symbols. However, this can be overcome by the symbol set conversion technique. For example, let us assume symbol 0 and 1 to be a conversion purpose symbol. When the 0 symbol is activated, the remaining 254 symbols can be used, and when symbol 1 is activated, another 254 symbols can be used. This is one of the techniques that can overcome the relatively insufficient number of symbols compared to relatively the larger number of STEs.

Failure reporting has three elements in the runtime verification system: the time that the failure occurs, type of the failure, and detection latency for post-failure analysis. The AP has 64 bits of metadata when reporting through reporting STEs, which includes reporting location and cycle information. In this experiment, a total of three reporting STEs are used, which are report timing failure, execution sequence failure, and memory value failure, respectively. This STE contains the type of failure and the cycle includes the failure time, which can be useful information for the

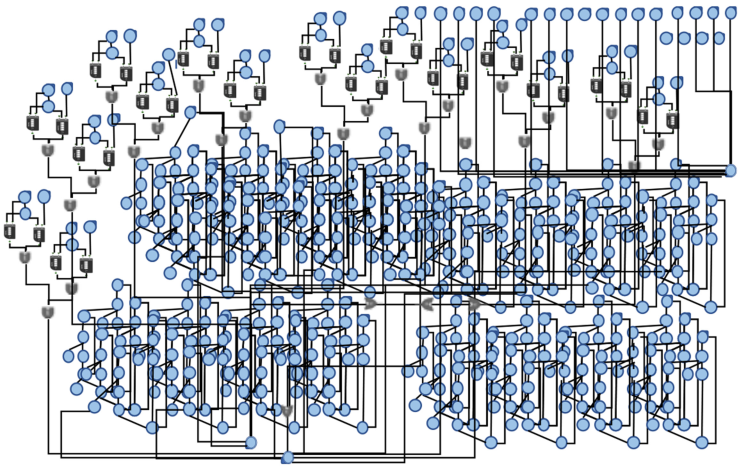

Fig. 11. Corresponding AP models extracted from requirements in Pacemaker example. *Note: symbol ID and detailed logic information are omitted due to the complexity of the model.*

post-failure analysis. Also, detection latency is significant in the case of medical devices because it is related to human life. Failure reporting considers only permanent failures and the failure should be reported to the microprocessor for further action in a similar manner to interrupts in a short period of time. The failure reporting time by Micron's reference document [16] is as follows:

$$\begin{aligned}1\text{ cycle (loading AP report L1)} + 2.5\text{ cycles (L1 export)} + 40\text{ cycles (L2 export to off-chip)}\\ = 43.5\text{ cycles} = 326\text{ ns}\end{aligned}$$

The AP-based failure reporting has a reasonable detection latency compared to a non-maskable interrupt (2 cycles), a general-purpose interrupt (20 cycles), and is several orders of magnitude faster than software-based runtime verification (913 700 cycles).

The current generation of APs has a clock speed of 133 MHz and quadruples the transfer rate using Double Data Rate Three (DDR3) technology with prefetch buffers. However, the clock for running automata still uses 133 MHz, which has a period of about 7.5 ns. Note that the AP is a memory-like processor using DDR3 interface, not a memory system. Therefore, the content stored in this AP is the application's requirements, not the application's data. The CPU used in this experiment is a 1 GHz ARM-compatible processor, and theoretically, it can output one event per 1 ns maximum. This provides sufficient resolution for most real-life applications. If the system requires events that occur more frequently than 7.5 ns, traces stored in the trace buffer within the filtering layer are processed sequentially. This requires a little more latency. Since the clock of the current generation AP problem limited to 133 Mhz, there is room for further improvement if another generation of APs come out such as DDR4 interface-based AP.

In Figure 12(a) and (b), we show a detailed analysis of the hardware utilization and (c) shows power overhead for 3 softcore processors as obtained from a Xilinx Vivado 2018.02 [13]. To measure overhead, we implemented a compatibility unit, a filter unit, and small size automata processors in FPGAs. Note that softcore processors are usually small in size for embedded systems. The area-optimized version of MicroBlaze is the smallest among the processors, which makes the overhead ratio look relatively high. On average, compatibility unit has 6% hardware resource overhead, and less than 1% power overhead. The filter unit supporting up to 128 events have 6% hardware resource overhead and 1% power overhead. Lastly, a lightweight FPGA version of the AP supporting

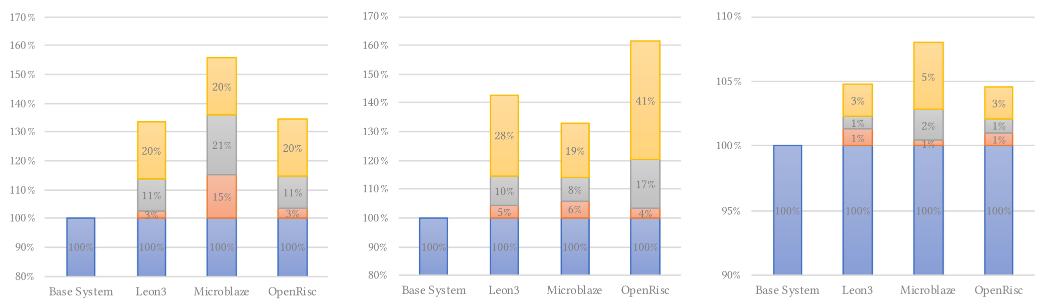

Fig. 12. Overhead of FPGA implementation for (a) look-up tables, (b) flip flops, and (c) power consumption.

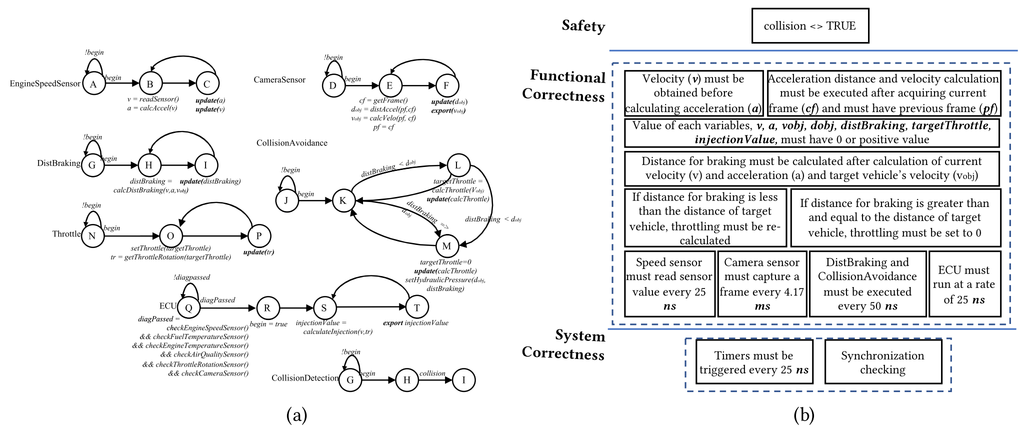
EngineSpeedSensor
CameraSensor
DistBraking
CollisionAvoidance
Throttle
ECU
CollisionDetection
Safety
collision <> TRUE
Functional Correctness
Velocity (v) must be obtained before calculating acceleration (a)
Acceleration distance and velocity calculation must be executed after acquiring current frame (cf) and must have previous frame (pf)
Value of each variables, v, a, vobj, dobj, distBraking, targetThrottle, injectionValue, must have 0 or positive value
Distance for braking must be calculated after calculation of current velocity (v) and acceleration (a) and target vehicle's velocity (vobj)
If distance for braking is less than the distance of target vehicle, throttling must be re-calculated
If distance for braking is greater than and equal to the distance of target vehicle, throttling must be set to 0
Speed sensor must read sensor value every 25 ns
Camera sensor must capture a frame every 4.17 ms
DistBraking and CollisionAvoidance must be executed every 50 ns
ECU must run at a rate of 25 ns
System Correctness
Timers must be triggered every 25 ns
Synchronization checking
(a)
(b)

Fig. 13. (a) state model and (b) properties for collision avoidance (CA) of autonomous driving application.

512 STEs and a verification unit containing a symbolizer have 24% hardware resource overhead and 4% power overhead.

A timed automata FPGA implementation [33] that only supports up to 17 states require 2845 LUTs and 2501 FFs, which are 1.8 times and 3.7 times larger than the filter unit + AP FPGA prototype with 1582 LUTs and 671 FFs Respectively. The TA implementation has relatively higher hardware resource usage to have a fixed number of programmable transition conditions, which might not be fully used in runtime verification. For safety-critical applications, this overhead is tolerable, especially when compared with the alternative of runtime verification in software implementation [33]. An optimized software runtime verification still incurs a 7% performance overhead and a 51% memory overhead [26].

To verify that the AP-based runtime verification is also suitable for other applications of the cyber-physical domain, we additionally considered a video-based collision avoidance (CA) system, shown in Figure 13. This implementation uses a single front-mounted camera to detect the location, distance, and speed of cars traveling in the same lane as the vehicle. Combined with vehicle speed, engine speed, and throttle position sensors, the system calculates the minimum safe following distance, and apply the brakes (and reduce throttle if needed) automatically if the distance is less than the minimum safe distance. The collision avoidance system consists of six state models, running periodically, and including EngineSpeedSensor, CameraSensor, DistBraking, CollisionAvoidance, Throttle, and ECU. Figure 13(a) shows the models and (b) shows correctness properties and a safety property. Table 5 represents details of elements for runtime

Table 5. Functions and Variables used in Runtime Verification for CA example

	Function and offset	Variable	
	Execution order, Timing	Value	Update order
Name	readSensor()	v	a
	calcAccel()	a	v
	getFrame()	pf	targetThrottle
	distAccel()	cf	tr
	calcVelo()	dobj	injectionValue
	calcDistBraking()	vobj	
	calcThrottle()	distBraking	
	setHydrauicPressure()	targetThrottle	
	checkEngineSpeedSensor()	tr	
	checkFuelTemperatureSensor()	begin	
	checkEngineTemperatureSensor()	injectionValue	
	checkAirQualitySensor()		
	checkThrottleRotationSensor()		
	checkCameraSensor()		
	main()		
# Events	62	22	10

Table 6. Result of Resource Utilization for two Cyber-Physical System Examples

	Pacemaker	Collision Avoidance
# Symbols	84 (32%)	66 (25.1%)
STEs	528 (1.25%)	244 (0.58%)
Boolean Logic Element	31 (1.3%)	28 (1.2%)
Counter Elements	32 (4.1%)	34 (4.36%)
Reporting Elements	3 (<1%)	3 (<1%)

verification. First, we create a TA model for CA state models. Second, we perform a model transform to convert the TA model to an AP model. Table 6 shows a result of resource usage in the automata processor with 1.48% without the number of symbols and 7% with the symbols in overall. As discussed earlier in this section, the symbol set replacement technique can overcome STE shortage problem to support a higher number of STEs. We show through two complete examples that various cyber-physical system applications can be verified at runtime through AP.

7 CONCLUSION

We presented a novel Automata Processor (AP)-based non-intrusive runtime verification methodology. Even though tracing can provide useful information during system execution at runtime, there are obstacles, which are the vast amount of real-time trace data, complex requirements of systems, and the limitation of hardware resources. The presented approach defines the Trace Abstraction Layer (TAL) according to the role/function in terms of tracing and enables efficient run time verification on Automata Processor. As a safety-critical example, we implemented a network-connected pacemaker application, showing how to extract the timed automata model from the requirements and convert it to an AP model. Using the application, we demonstrated 100% detection rate for various failure types. In addition to the experiment, we analyzed the problems that may occur from the AP implementation and examined the overhead in case of FPGA hardware implementation. Future work includes an automated approach of requirement mining, an efficient and automated transform algorithm from TA model to AP model, and complex verification via FPGA and AP combined environment.

REFERENCES

[1] [n.d.]. RTEMS Real Time Operating System (RTOS) | Real-Time and Real Free RTOS. https://www.rtems.org/.
[2] 2018. LEON/GRLIB guide GRLIB VHDL IP core library 2018 configuration and development guide configuration and development guide. (2018). www.cobham.com/gaisler.

[3] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. 2017. Verifying data-and control-oriented properties combining static and runtime verification: Theory and tools. *Formal Methods in System Design* 51, 1 (2017), 200–265.

[4] Rajeev Alur. 1999. Timed automata. In *International Conference on Computer Aided Verification*. Springer, 8–22.

[5] S. Serge Barold, Roland X. Stroobandt, and Alfons F Sinnaeve. 2008. *Cardiac Pacemakers Step by Step: An Illustrated Guide*. John Wiley & Sons.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[7] Chunkun Bo, Ke Wang, Yanjun Qi, and Kevin Skadron. 2015. String kernel testing acceleration using the micron automata processor. In *Workshop on Computer Architecture for Machine Learning*.

[8] Eric Bodden, Patrick Lam, and Laurie Hendren. 2010. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *International Conference on Runtime Verification*. Springer, 183–197.

[9] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. 2011. Sampling-based runtime verification. In *International Symposium on Formal Methods*. Springer, 88–102.

[10] Altera Corporation. 2008. with the SignalTap II Embedded Logic Analyzer. 24 pages.

[11] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098.

[12] Mohammed El Shobaki and Lennart Lindh. 2001. A hardware and software monitor for high-level system-on-chip verification. In *Proceedings of the IEEE 2001. 2nd International Symposium on Quality Electronic Design*. IEEE, 56–61.

[13] Tom Feist. 2012. Vivado design suite. *White Paper* 5 (2012), 30.

[14] Richard Fryer. 2005. FPGA based CPU instrumentation for hard real-time embedded system testing. *ACM SIGBED Review* 2, 2 (2005), 39–42.

[15] Brent Hailpern and Padmanabhan Santhanam. 2002. Software debugging, testing, and verification. *IBM Systems Journal* 41, 1 (2002), 4–12.

[16] Micron Inc. [n.d.]. Designing for the Micron D480 Automata Processor. http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html.

[17] Zhihao Jiang, Miroslav Pajic, Allison Connolly, Sanjay Dixit, and Rahul Mangharam. 2010. Real-time heart model for implantable cardiac device validation and verification. In *2010 22nd Euromicro Conference on Real-Time Systems*. IEEE, 239–248.

[18] Zhihao Jiang, Miroslav Pajic, and Rahul Mangharam. 2011. Cyber–physical modeling of implantable cardiac medical devices. *Proc. IEEE* 100, 1 (2011), 122–137.

[19] Mike Jones. 1997. What really happened on mars rover pathfinder. *The Risks Digest* 19, 49 (1997), 1–2.

[20] Damjan Lampret and Julius Baxter. [n.d.]. OpenRISC 1200 IP Core Specification (Preliminary Draft), 2014.

[21] Jong Chul Lee and Roman Lysecky. 2015. System-level observation framework for non-intrusive runtime monitoring of embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 20, 3 (2015), 42.

[22] Nancy G Leveson and Clark S Turner. 1993. An investigation of the Therac-25 accidents. *Computer* 26, 7 (1993), 18–41.

[23] Giovanni Liva, Muhammad Taimoor Khan, and Martin Pinzger. 2017. Extracting timed automata from Java methods. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 91–100.

[24] Hong Lu and Alessandro Forin. 2007. The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. (2007).

[25] R Mijat. 2010. Better trace for better software: Introducing the new arm coresight system trace macrocell and trace memory controller. *ARM, White Paper* (2010).

[26] Samaneh Navabpour, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2015. Time-triggered runtime verification of component-based multi-core systems. In *Runtime Verification*. Springer, 153–168.

[27] Hyunyoung Oh, Hayoon Yi, Hyeokjun Choe, Yeongpil Cho, Sungroh Yoon, and Yunheung Paek. 2019. Real-time anomalous branch behavior inference with a GPU-inspired engine for machine learning models. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 908–913.

[28] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Rosu. 2008. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *2008 Real-Time Systems Symposium*. IEEE, 481–491.

[29] Kevin Peterson and Yvon Savaria. 2004. Assertion-based on-line verification and debug environment for complex hardware systems. In *2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No. 04CH37512)*, Vol. 2. IEEE, II–685.

[30] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE, 46–57. DOI: https://doi.org/10.1109/SFCS.1977.32

[31] Indranil Roy and Srinivas Aluru. 2014. Finding motifs in biological sequences using the micron automata processor. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 415–424.

[32] Indranil Roy, Ankit Srivastava, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. 2016. High performance pattern matching using the automata processor. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1123–1132.

[33] Minjun Seo and Roman Lysecky. 2018. Non-intrusive in-situ requirements monitoring of embedded system. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23, 5 (2018), 58.

[34] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards machine learning on the automata processor. In *International Conference on High Performance Computing*. Springer, 200–218.

[35] Tullis and Michael L. 2015. *Intel ® Trace Hub Developer's Manual*. Technical Report. http://www.intel.com/products/processor.

[36] Jack Wadden and Kevin Skadron. 2016. VASim: An open virtual automata simulator for automata processing application and architecture research. *University of Virginia, Tech. Rep. CS2016-03* (2016).

[37] Philipp Wagner, Thomas Wild, and Andreas Herkersdorf. 2016. DiaSys: On-chip trace analysis for multi-processor system-on-chip. Springer, Cham, 197–209. DOI: https://doi.org/10.1007/978-3-319-30695-7_15

[38] Ke Wang, Yanjun Qi, Jeffrey J. Fox, Mircea R. Stan, and Kevin Skadron. 2015. Association rule mining with the micron automata processor. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 689–699.

[39] Kosuke Watanabe, Eunsuk Kang, Chung-Wei Lin, and Shinichi Shiraishi. 2018. INVITED: Runtime monitoring for safety of intelligent vehicles. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6. DOI: https://doi.org/10.1109/DAC.2018.8465912

[40] Xilinx. 2012. ChipScope Pro Software and Cores. , 5–226 pages. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/chipscope_pro_sw_cores_ug029.pdf.

[41] Xilinx. 2017. MicroBlaze processor reference guide. (2017). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug984-vivado-microblaze-ref.pdf.

[42] Keira Zhou, Jack Wadden, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. 2015. Regular expression acceleration on the micron automata processor: Brill tagging as a case study. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 355–360.

[43] Yumin Zhou, Sebastian Burg, Oliver Bringmann, and Wolfgang Rosenstiel. 2018. A software reconfigurable assertion checking unit for run-time error detection. In *2018 IEEE 23rd European Test Symposium (ETS)*. IEEE, 1–6. DOI: https://doi.org/10.1109/ETS.2018.8400691

Received April 2019; revised June 2019; accepted July 2019