Efficient Temporal Logic Runtime Monitoring for Tiny Systems

Rüdiger Ehlers[0000−0002−8315−1431]

Clausthal University of Technology, Clausthal-Zellerfeld, Germany
`ruediger.ehlers@tu-clausthal.de`

Abstract. While the theory and practice of runtime monitoring are overall well developed, in embedded systems, runtime monitoring is not as common as one would expect. Especially small-scale embedded systems, which are found in many household devices, are often somewhat safety-critical and could benefit from the possibility to detect software or hardware defects that cannot be uncovered with verification alone. While monitoring frameworks such as RTLola and Copilot are available to address this problem, employing them leads to a gap between verification and runtime monitoring by these frameworks having specialized specification languages for monitoring, and what can be expressed in them is incomparable to the capabilities of the temporal logics traditionally employed in formal verification.

This paper discusses how (linear) temporal logic runtime monitoring for small-scale embedded systems can be made more efficient and attractive to the embedded systems practitioner. This includes identifying why monitoring for traditional temporal logics is somewhat inefficient in software, how this problem can be addressed in a low-cost way, and how runtime monitors can become a more useful component of an embedded system. By providing a way to translate a specification to monitor code that also tracks the reason for a specification violation, the overall approach discussed in the paper makes spending developer time on writing a relatively precise specification of a system to build attractive, which also helps paving the way to make formal verification for small-scale embedded systems more common.

1 Introduction

Many real-world systems are too complex to verify directly. They have too many states, so that we cannot simply build a transition system representation of them and compare it against some specification. This problem is addressed by several different approaches in formal methods. They key activity in some of them is abstracting the design under concern to something more manageable, either by means of a carefully made abstraction, or by learning such an abstraction and using the result as a model.

Not all systems lend themselves to obtaining such an abstraction, however, and in such a case, we have to deal with a relatively complex system. This is especially the case when the environment of a system is not fully modeled, but the environment behavior is crucial for the correctness of a system. In such a case, we have to trade being able to detect every error (right away) against being able to analyze a complex system without abstraction, which means observing system behavior and *testing* or *monitoring* it for correctness. In this case, the system is used in a black-box or gray-box manner without the need for an abstraction. We can however still utilize a formal specification, so that we can reason formally about *an* execution of a system, but not its set of executions.

Testing and monitoring differ by scope and how a system is used. In the former case, a system (under test) is employed in a testing environment, which allows to focus on individual aspects of its operation and to observe the system's behavior in carefully crafted situations, such as boundary cases. Furthermore, in testing, relatively expensive analysis techniques can be used for analyzing the behavior of the system under test, and testing setups can be augmented with components for tracking the precise nature of a specification violation, which helps with finding the cause of a specification violation in case it occurs.

Monitoring, on the other hand, is a related but independent activity. Monitoring means to assess the correctness of the system's behavior by observing its behavior and raising an alarm whenever a violation of the specification is witnessed. Monitoring can be used in the scope of testing but is not bound to it – an efficient monitor can be deployed in the field along with the system whose correctness we are interested in, so that even violations of properties of interest that were not found during testing due to missing test scenarios can be uncovered. In the case of embedded systems, monitoring also allows to identify erroneous behavior due to hardware defects (e.g., in sensors or actuators) for which the system's behavior was not formally verified. The output of a monitor can furthermore be used to influence the system's operation itself, such as by the system shutting down in an orderly manner in case of a specification violation.

The possibilities to also detect specification violations due to unmodelled causes (such as hardware defects) and to allow the system to react to violations are unique properties of monitoring among the different approaches to proving the correctness of a system. Especially for embedded systems of a smaller scale, monitoring is attractive, as the interaction of such systems with the physical environment and their hardware-specific software implementations reduce the possible degree of test automation (when compared to pure software-based systems). Small-scale embedded systems do not have layers of abstraction that allow to test their software independent of the hardware. Many required properties of embedded systems can be represented using *temporal logics*, and approaches to temporal logic runtime monitoring have been developed that yield monitors that can be integrated into embedded systems.

For instance, Havelund and Rosu [21] showed how to translate a linear past time temporal logic formula into C code that monitors the satisfaction of the formula when calling the monitor update/step function whenever the propositions occurring in the formula receive new values. The C code is relatively cumbersome to execute due to having many bitwise operations that each take at least one clock cycle for a microcontroller. This can be avoided by having the monitor run on a *field-programmable gate array (FPGA)*, where bitwise operations can execute in parallel and when monitoring the input/output behavior of an existing system, the FPGA can simply be connected to input and output to perform the monitoring process [27]. This approach has the drawback that FPGAs are normally more expensive than the microcontrollers they monitor, which causes the approach to have a relatively high cost.

While most embedded systems are closed-source and hence statistics on the adoption of runtime monitoring in embedded systems are difficult to give, it can still be observed that both in research as well as at industry trade shows, runtime monitoring for small-scale embedded systems is seldom a topic that is widely discussed (unlike, e.g., security for embedded systems), despite many runtime monitoring approaches already being available. This is rather unfortunate, as it has been noted that a show-stopper to the wider adoption of formal methods in industry is the need to train engineers in writing and working with formal specifications [32]. With its unique properties, runtime monitoring could be a wedge in the door toward greater adoption of formal methods in the embedded systems industry, which would then also help with paving the way for proof-based formal methods (such as the combination of learning a system model from traces and verifying this model against the specification [36]).

In recent years, runtime monitoring frameworks such as Copilot [28] and RTLola [18] entered the scene, which take runtime monitoring to the next level by allowing to provide a stream-based specification of the properties to be monitored and having compilation workflows to either C code or FPGA implementations. They feature operations over stream data as first-class citizens, which makes expressing complex properties over data feasible. As such, they make monitoring more attractive, but come with the draw-back that the domain-specific monitoring languages are less expressive in the temporal behavior of systems than classical temporal logics such as linear temporal logic (LTL, [29]). Also, employing monitoring-specific specification languages causes a divide between runtime monitoring and formal verification (before the deployment of a system), which is undesirable for establishing more rigorous specification engineering practices in industry. Arguably, even the existing work on monitoring linear past time temporal logic formulas can be considered to be part of such a division, as formal verification is frequently done with temporal operators that look into the future.

So why is it that runtime monitoring with classical (future-time) temporal logics appears to be under-explored in industry given its benefits? We argue that there are still two problems that need to be solved to make runtime monitoring for low-cost embedded systems useful. On the one hand, monitors are surprisingly *costly*: the amount of computation time of a microcontroller spent monitoring in software can easily grow larger than the computation time needed for the core functionality. Conversely, when offloading the monitoring task to an FPGA, this extra chip can easily be more expensive than the microcontroller for the core functionality. On the other hand, for a monitor to be useful, detecting violations needs to be useful. Without diagnostic information on *how* and *why* a violation occurred, a reported error is hardly useful for improving the design of a system, which is often the reason for applying formal methods in the first place.

We show in the following how both problems can at least be partially addressed in a way suitable for very small embedded systems with computing hardware costs of less than 2 Euros. Starting from linear temporal logic as specification formalism, we summarize some previously known concepts for the computation of an online runtime monitor for embedded systems and show what makes the resulting monitors difficult to apply on small-scale embedded systems. To address these issues, we present a novel microcontroller temporal logic runtime monitoring component that is easy to integrate into future microcontrollers for safety-critical systems, and for which a prototype has already been produced in silicon. To address the need for the monitor supporting the debugging of real-world systems, we also show how the problem of automatically capturing debug information with temporal logic runtime monitors can be addressed in a way that is reasonable under very tight storage space constraints for trace information, as commonly the case in tiny embedded systems.

2 Related Work

With the focus of this paper on showing why temporal logic monitoring is difficult on small-scale embedded systems and discussing what can be done about it, the amount of related work is too big to discuss it in a comprehensive way. Rather, a (small) selection of existing results that are particularly related to the following discussion are mentioned below. Runtime monitoring is a core concept of the research field of *runtime verification*, for which an introductory textbook exists [3], as well as a dedicated conference series (RV).

Temporal logics such as linear temporal logic (LTL, [29]) are well-established for the specification of desired system behavior. LTL reasons over infinite traces, while in runtime monitoring, we can only observe *finite prefixes* of a trace. Apart from defining special LTL semantics for runtime monitoring that reason over finite traces [20,6], one can solve this problem by defining monitoring as the problem of continuously checking if the trace of the system observed so far can be *extended* to one that satisfies the specification. Doing so lets the LTL semantics for model checking and runtime monitoring coincide. We call a prefix trace of a system a *bad prefix* if it cannot be extended to a trace satisfying the specification. Monitoring in this semantics amounts to checking if a bad prefix has been observed, and hence only the *safety hull* of a property under concern can be monitored, with relatively few exceptions [16].

Kupferman and Vardi described how to monitor the safety hull of an LTL property, where the monitor is a finite-state machine of size at most doubly-exponential in the length of the LTL property [22]. The monitor is structured in a way amenable to symbolic implementations, which enables, for instance, a compact encoding of the monitor as a circuit. Such an encoding can be compiled to a circuit for monitoring and this circuit can be implemented in a *field-programmable gate array* (FPGA). Due to the fact that FPGAs offer highly parallel computing in real-time, they are used in many monitoring approaches (e.g., [27,8,30]). The application of FPGAs for monitoring requires that the FPGA is physically connected to the signals to be monitored and at the same time using FPGAs adds to the cost of a system. Also, when the monitored signals change too fast for the synthesized FPGA implementation to process, only sampling-based runtime monitoring [10] can be applied.

For embedded systems that employ microcontrollers, the cost of an FPGA for monitoring can easily exceed the cost of the system to be monitored. Deploying a monitor on the microcontroller itself is a more reasonable alternative. It involves *instrumenting* [11] the microcontroller code to be monitored so that the monitor gets informed of events of relevance to the satisfaction of the specification.

For complex embedded systems that run a real-time operating system, evaluating the stream of events for whether it represents a bad prefix can be done in a separate thread that is regularly scheduled by the real-time operating system [26]. Not all microcontrollers in embedded systems run real-time operating systems, though. Especially when their purpose does not require communication between different software components and when they need to react quickly to events, scheduling real-time tasks can complicate a design rather than simplifying it. Rather, many simpler embedded systems are programmed "bare metal", which gives the system engineer full control over the microcontroller timing. This observation advocates for integrating the monitor code directly into the monitored code itself so that it runs *synchronously* with the code to be executed [4].

Either way, monitoring needs to be efficient as the overhead of runtime monitoring for temporal logics can be substantial. Symbolic implementations of automata for the evaluation of logical formulas, as used in FPGA-based monitoring, are difficult to translate to efficient microcontroller code as a lot of "bit fiddling" code needs to be used then (see, e.g., [21]), which leads to a high number of clock cycles of overhead to the instrumented instructions.

The detection of a specification violation can be used in multiple ways. Apart from switching the monitored system to a *fail-safe mode*, detecting a violation is the starting point to finding out *why* the system behaved incorrectly. For instance, Wang et al. [39] presented an approach based on causality analysis, in which the component responsible for the violation within a bigger system is identified. Finding this component has also been studied outside of the area of runtime verification (see, e.g., [5] and the references therein).

In a sense, approaches that identify components that behave incorrectly are however only the second step to analyzing specification violations. Especially for more complex specifications, finding out *how* a specification was violated is a useful first step. This can for instance be done by highlighting which parts of the trace contribute to the violation of the specification. Beer et al. [9] employ causality to find the important time point/proposition combinations in counter-examples. They show that determining the minimal number of combinations needed to explain a violation is NP-hard, but give a polynomial-time heuristic. The approach is however not applicable to online monitoring as it requires the complete trace for the analysis. Ferrère et al. [19] provide an alternative approach that is not based on causality, but is also suitable for metric time logic and the explanation of violations in infinite (lasso-shaped) traces. The development of a corresponding online monitoring approach was left for future work.

The idea to perform runtime monitoring in hardware in order to reduce the computational cost of runtime monitoring is not restricted to the use of FPGAs [33,27]. Walters et al. [38] discuss this idea for supervising a microcontroller, however without a temporal logic specification. Reinmacher et al. [31] propose the development of a programmable runtime verification component, but focus on describing the concept rather than refining it to an implementation of such a component.

3 Linear Temporal Logic Runtime Monitoring on Small-scale Systems – Step By Step

To illustrate why a straight-forward application of classical temporal logic runtime monitoring results yields monitors that are relatively inefficient, we illustrate how they work on an example. The example is then used as motivation for discussing refinements of the approach.

Full-length monitor code for the example in the following is available from [17]. We consider a couple of monitor code variants and report on code sizes and computation times for a 32-bit F446RE microcontroller by ST Microelectronics, which features an ARM Cortex M4 processor core, 512 kilobytes of program memory (Flash) and 128 kilobytes of RAM. While it is not actually a tiny system, its processor core has a useful extra feature, namely a clock cycle counter, which we employ for comparing the computation times of different monitors. Computation times for the monitor are taken on an example trace of length 7.

Linear temporal logic: We start with a specification in Linear Temporal Logic (LTL, [29]), which is a logic over infinite words $w \in (2^{\mathsf{AP}})^\omega$ of which each character assigns values to a finite set of atomic propositions AP. A word either satisfies an LTL formula or not. In the latter case, we say that the formula is *violated*. Linear temporal logic extends Boolean logic by the temporal operators G ("globally"), F ("eventually"), X ("next"), $\mathcal{U}$ ("until"), and $\mathcal{R}$ ("release"). We refer the reader to [29,22] for a formal definition of LTL.

For the example in the following, we consider monitoring a simple traffic light at an intersection, which is safety-critical and has temporal requirements. The traffic light has remotely controlled override signals for ambulance vans. To keep the example simple, we assume that the system is clocked with a clock rate of one execution step per second. For every direction, the traffic light has red, yellow and green light bulbs, and we assume that the signals shown on opposite sides of the intersection are always the same. The traffic light controller has to satisfy four properties:

1. Whenever one of the traffic lights shows yellow, it shows red next. Whenever a traffic light shows red+yellow, green follows next.
2. After a traffic light shows green, it has to show yellow or green next. Similarly, after a red light in any direction, the respective traffic light has to show red or red+yellow next.
3. Only in one direction at a time, the traffic lights may have a yellow or green light bulb lit.
4. In each direction, an ambulance van override signal can be issued, and if it keeps being issued for 10 seconds, the traffic light for the respective direction has to show green after the 10 seconds and stay green until the signal is turned off again.

The specification has the interesting property that no system can satisfy it along all possible executions. Ambulance van override signals can be given for two directions at the same time, and then the traffic light controller would need to give two green signals in different directions at the same time, which violates the third property. However, this case may not occur in practice, for instance by the receiver for the ambulance override signals only allowing one such signal at a time, and unlike in formal verification, we do not have to model this environment aspect to avoid that the employed verification approach spuriously detects the specification to be violated.

To formalize the specification in linear temporal logic, we first need to define a set of atomic propositions for the system. For simplicity, we assume a four-way intersection with traffic lights that always have the same bulbs lit in opposite directions, without separate signaling for cars that wish to turn left. We use $r_1, r_2, y_1, y_2, g_1, g_2$ as the propositions for the red, yellow, and green signals for the two relevant directions, respectively, and the two additional propositions a_1, a_2 for the ambulance override signals in these directions. The specification parts from above can be encoded into LTL as follows:

1. $\bigwedge_{i \in \{1,2\}} \mathsf{G}(\neg r_i \wedge y_i \rightarrow \mathsf{X}(\neg y_i \wedge r_i)) \wedge \mathsf{G}(r_i \wedge y_i \rightarrow \mathsf{X} g_i)$
2. $\bigwedge_{i \in \{1,2\}} \mathsf{G}(g_i \rightarrow \mathsf{X}(g_i \vee y_i)) \wedge \mathsf{G}(r_i \rightarrow \mathsf{X}(r_i \vee \neg r_i \wedge y_i))$
3. $\mathsf{G}\neg((y_1 \vee g_1) \wedge (y_2 \vee g_2))$
4. $\bigwedge_{i \in \{1,2\}} \mathsf{G}(a_i \rightarrow \underbrace{\mathsf{X}(\neg a_i \vee \mathsf{X}(\neg a_i \vee}_{\text{10 times}} \ldots (g_i \mathcal{U} \neg a_i) \ldots)))$

Note that not all aspects of the system's operation have been formalized, such as that if a traffic light has its green bulb lit, its other light bulbs have to be switched off. For runtime monitoring, which we want to formalize next, it is not necessary to include all such aspects.

Runtime Monitoring of LTL: Runtime monitoring is the process of observing the execution of a (reactive) system and continuously checking whether the trace so far already violates a given specification. Early guaranteed satisfaction is not considered in this paper. *Reactive systems*, which continuously interact with their environment, do not terminate and their execution can be written as an infinite *trace* in which each *trace element* provides information about the current input/output and the state of the system. For many monitoring tasks, the trace can be abstracted into a form in which each trace element is an assignment to a set of Boolean variables AP, which enables the use of LTL to specify desired system properties. Linear temporal logic is defined on traces of infinite length, and hence whether a specification is violated is not directly defined for a finite trace prefix that can be observed at runtime. To avoid introducing a special semantics for LTL over finite traces, it is common to monitor for *bad prefixes* of an trace. Formally, given an infinite word $w = w_0 w_1 \ldots \in (2^{\mathsf{AP}})^\omega$, a bad prefix $w_0 w_1 \ldots w_i$ of w (for some $i \in \mathbb{N}$) has the property that for every $w' \in (2^{\mathsf{AP}})^\omega$, we have that $w_0 \ldots w_i w'$ violates the specification.

For the traffic light specification that we formalized above, an example bad prefix is $\{r_1, r_2\}\{r_1, y_2\}\{r_1, g_2\}\{g_1, r_2\}$, as here, a sudden switch from green to red occurs for the second direction, which violates specification part number 2 above.

So how can we construct software components of embedded systems from LTL specifications that allow us to detect bad prefixes of the system's execution? Kupferman and Vardi [22] describe an automata-theoretic way for doing so that is based on *Büchi automata*.

Büchi automata: A Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with a finite set of states Q, an alphabet Σ (which will be a set of valuations of an atomic proposition set AP in this paper), a transition relation $\delta \subseteq Q \times \Sigma \times Q$, a set of initial states $Q_0 \subseteq Q$, and a set of accepting states F. Büchi automata operate on infinite words $w = w_0 w_1 \ldots \in \Sigma^\omega$. We say that an infinite sequence $\pi = \pi_0 \pi_1 \ldots \in Q^\omega$ is a run of $\mathcal{A}$ on w if $\pi_0 \in Q_0$ and for all $i \in \mathbb{N}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta$. We say that π is accepting if some state in F occurs infinitely often along π, and the word w is *accepted* by $\mathcal{A}$ if there exists some accepting run for w and $\mathcal{A}$. The words accepted by an automaton form its *language*. We extend the notion of a run of a Büchi automaton to finite words in the straight-forward way, but note that it does not make sense to talk about the acceptance of such a run.

It is known how to translate a linear temporal logic formula to a Büchi automaton of size exponential in the size of the LTL formula [37]. In this context, the set of words satisfying the LTL property is the language of the Büchi automaton. For the specification given above (where we take the conjunction of the specification parts as the overall specification), we can obtain an equivalent Büchi automaton with 2904 states when using the LTL-to-Büchi translation that is part of the `spot` framework [13] (of which we use version 2.12 in this paper).

Runtime monitoring with Büchi automata – first version: A Büchi automaton for a specification can be used to detect bad prefixes of the specification. For a first approach, we first need to prune all states from a Büchi automaton $\mathcal{A}$ that cannot occur along an accepting run of a word [12], which is a standard optimization step of LTL-to-Büchi translators anyway [34]. Afterwards, we have that a prefix word $w = w_0 w_1 \ldots w_n$ is a bad prefix for the language of a Büchi automaton $\mathcal{A}$ if and only if there is no finite run of $\mathcal{A}$ for w [2]. To see this, consider that there is such a finite run π. Then it can be extended to an accepting run for some infinite extension of w (as π can be the beginning of an accepting run as otherwise the state that is last in π would have been pruned away). On the other hand, if there is no such finite run π, then there is no infinite run for any infinite extension of w, which makes w a bad prefix.

This observation allows to perform runtime monitoring in a simple way: while observing the behavior of the system, a monitor can read the proposition valuations step-by-step and keep track of in which Büchi automaton states a run for the prefix trace observed so far can be. Once this state set becomes empty, a bad prefix has been observed, and the monitor can flag an alarm. We can compile a Büchi automaton to monitoring code that computes the next state set from the previous state set and the atomic propositions. To do so effectively, we group transitions by their predecessor and successor states and compile the set of characters for which some transition in the group can be taken into some symbolic expression.

For our running example, the Büchi automaton for the conjunction of the four properties has 2904 states. Figure 1 shows a part of the resulting monitor code for our example, which overall has 98303 source code lines. The source code file is too large to compile to ARM with the GNU C compiler version 7.2.1 (which runs out of memory on a 32 GB RAM computer), both with code optimization turned on or off.

Runtime monitoring with Büchi automata – deterministic approach: The program code shown in Figure 1 shows a problem with the simple approach to runtime monitoring described above, namely that the generated code is quite inefficient. Since code blocks for different predecessor states of the automaton's transitions need to be executed for each step of the monitor, the computation time of a monitor step is quite long.

A simple way to address this problem is to treat the pruned Büchi automaton as a non-deterministic safety automaton (so that every state is accepting) and to determinize this automaton [35,7]. Every state in the determinized automaton is then labeled by the set of states in the original automaton in which a run can be in. Whenever the state labeled by the empty set is reached in a run of the determinized automaton, this constitutes having read a bad prefix.

In our running example, determinization (and performing exact automaton minimization of the result) actually decreases the the number of states to 2816. For tracking the state, only a single integer variable needs to be stored in the microcontroller's RAM. The GNU C compiler again fails to compile the resulting monitor code due to running out of memory when code optimization is turned on. Without code optimization, the resulting code size for our example is 2434380 bytes, which is too large for small and medium-sized microcontrollers.

```
/* State storage information */
uint8_t inState0 = 1;
uint8_t inState1 = 0;
...
uint8_t inState2903 = 0;

/* Monitor step/update function */
int monitor(uint8_t r1,uint8_t y1,uint8_t g1,uint8_t r2,uint8_t y2,uint8_t g2,
  uint8_t a1,uint8_t a2) {
  uint8_t nextState0 = 0;
  ...
  uint8_t nextState2903 = 0;
  if (inState0){
    if (!g1&&!y1&&!g2&&!y2&&!a1&&!a2&&!r1&&!r2) nextState0 = 1;
    if (!g1&&!y1&&!g2&&!y2&&!a1&&!a2&&!r1&&r2) nextState1 = 1;
  ...
  if (inState2903){
    if (g1&&!y1&&!g2&&!y2&&!a1&&!a2&&r1&&r2) nextState75 = 1;
    if (g1&&!y1&&!g2&&!y2&&a1&&!a2&&r1&&r2) nextState2679 = 1;
  }
  inState0 = nextState0;
  ...
  inState2903 = nextState2903;
  if (inState0) return 0;
  ...
  if (inState2903) return 0;
  return 1; /* Reporting a violation */
}
```

Fig. 1. Some excerpts of a runtime monitor code based on a single Büchi automaton

Runtime monitoring with Büchi automata – fragmenting the specification: A way to generate smaller monitoring code is to build four separate monitors for each of the four properties in the running example, and monitoring them separately. This means that we build non-deterministic Büchi automata for each of them and generate the same type of code as for the first monitor version, once for each automaton. If and when at runtime, any component monitor reports a violation, the joint monitor reports a violation.

This approach leads to more compact monitor code, requiring only 27024 bytes of code size, which now fits the program memory of the example microcontroller. The four Büchi automata built for this monitor have 147 states overall. On the short example trace, we find that the code needs 2907 clock cycles on average per monitor step for its execution.

Runtime monitoring with universal automata: The observation from the previous variant, namely that the monitor became smaller after translating the specification parts to automata separately, leads to the question of how far the idea can be pushed. Is there perhaps a way to decompose the specification into many very small conjuncts? On an automata-theoretic level, this question has a positive answer in the form of *universal automata* [23]. In a nutshell, the idea in universal automata over infinite words is that *every* infinite run of the automaton for a word needs to be accepting for the word to be accepted by the automaton.

In fact, every LTL property can be translated to an equivalent *universal co-Büchi* automaton for the specification. The co-Büchi acceptance condition in this context defines that runs for which states in the automaton's F set

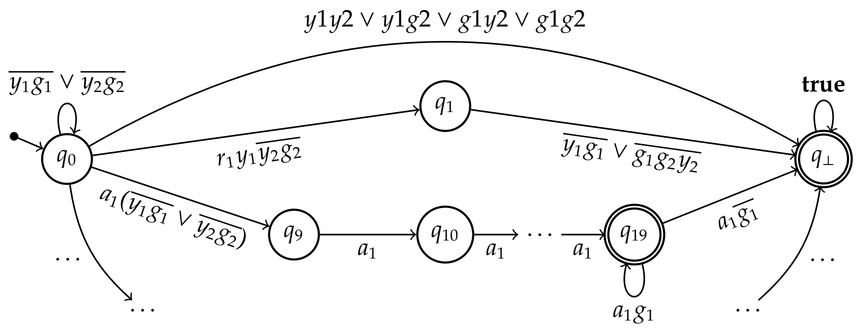
$y1y2 \vee y1g2 \vee g1y2 \vee g1g2$
$\overline{y_1 g_1} \vee \overline{y_2 g_2}$
true
q_0
q_1
$q_\perp$
$r_1 y_1 \overline{y_2 g_2}$
$\overline{y_1 g_1} \vee \overline{g_1 g_2 y_2}$
$a_1(\overline{y_1 g_1} \vee \overline{y_2 g_2})$
q_9
q_{10}
q_{19}
a_1
a_1
a_1
$a_1 \overline{g_1}$
$a_1 g_1$
…

Fig. 2. A part of the universal co-Büchi automaton for the specification of the running example. Rejecting states are doubly-circled. It can be observed that some transition labels are needlessly complicated as the focus of the LTL-to-Büchi translator is on minimizing the number of states rather than the simplifying the transition labels.

(which are then called the *rejecting states*) are visited only finitely often are accepting. Universal co-Büchi automata for a specification consisting of many conjuncts often look more structured than non-deterministic Büchi automata for the same language, as the different states in such automata keep track of different requirements on the rest of the word in order for it to be accepted. A sub-class of universal co-Büchi word automata are *universal very weak automata*, which capture exactly those temporal properties that are expressible in LTL as well as computation tree logic (CTL) with universal path quantifiers, where in the latter case, we consider the satisfaction of the property along all paths of a computation tree [25]. In fact, the specification from above falls into this class, and there exists a specialized translation procedure from a subset of LTL into universal very weak automata [1].

Since when working with universal automata, the automaton rejects a word if *any* trace violates the specification, we need to modify what the monitor code does. We still need to track in which states a run of the automaton can be. However, to detect bad prefixes with universal automata, we need to check if for the respective current state combination Q', every suffix word has a rejecting run from *one* of the states in Q'. For a given universal automaton, finding these state combinations is a model checking problem that can be solved upfront when generating the monitor. However, when the specification is simple enough, these state combinations are often exactly the ones containing a state that has an empty language, i.e., one for which when changing the initial states of the automaton to only that state, the language of the automaton becomes empty. A simplification at the cost of completeness (i.e., we may miss violations in some cases) is to refine the monitoring problem to be detecting if some run of the universal automaton has already reached such a state with an empty language.

Our specification in the running example translates to a universal very weak automaton using `spot` 2.12 even without the specialized translation procedure from [1] and is simple enough so that detecting a bad prefix by checking if the state with the empty language has been reached is sound and complete. A part of the universal automaton is shown in Fig. 2, and the complete automaton has 30 states.

The resulting monitor code needs 3136 bytes of program memory and takes 549 clock cycles on average to execute per monitor step. We note that the program code is small enough to fit into the program memory of low-cost microcontrollers, which typically have 16-128 kilobytes of this memory type. A particular advantage of monitoring with universal automata is that the monitor code size does not grow superlinearly with the number of monitored properties. In case the main functionality of the microcontroller needs most of the program memory, we can prioritize the properties to monitor in order to fill the remaining memory with monitor code for a selection of the properties.

4 Keeping Track of Specification Violation Reasons

For the example in the previous section (for which all details and the translation procedures to monitors are available in [17]), we used results from the literature, namely on runtime monitoring using non-deterministic Büchi automata [12], deterministic automata [35], and on universal co-Büchi automata, where in the last of these cases, we adapted the monitoring approach from [12] to account for the complemented branching and acceptance conditions. The Python script for compiling the monitors in all considered variants only has 435 lines of code, showing that monitors of all considered types can easily be built.

While the monitors are already usable for embedded systems (when adding calls to the monitor step function in the main program code manually), they leave room for improvement in several directions, of which we want to address two in this paper.

First of all, what happens if there is a violation? The monitor detects this, and the embedded system implementation can then react to the violation, for instance by shutting down the system's service. For systems in a prototyping stage and systems that can report specification violations back to the manufacturer, recording the reason for a violation would also be useful, so that after a violation, debugging and continuous engineering activities can be supported with a suitable starting point. Recording complete execution traces is however out of question, as microcontroller memory is extremely limited.

We propose an alternative approach that makes use of the structure of universal very weak automata, as we have in the running example above. An automaton is said to be very weak (or *1-weak*) if all loops along the transitions in the automaton are self-loops [25]. For universal such automata, the different possible paths that runs can take through the automaton constitute different ways of violating the specification, where a path defines the states along a run without state repetitions. For instance, the path $q_0 \rightarrow q_1 \rightarrow q_\perp$ in the automaton in Figure 2 represents the case that after red+yellow is shown for the traffic light in direction 1, green is not shown right afterwards (where the actual transition conditions happen to be more complex in the automaton so that this path cannot be taken when certain other specification violations happen at the same time). Similarly, a path from q_0 to $q_\perp$ via $q_9 \rightarrow \ldots \rightarrow q_{19}$ constitutes a specification violation by the ambulance van override signal not working correctly. In this case, however, the number of trace characters between leaving q_0 and entering $q_\perp$ along a run is not bounded, as the run may stay in q_{19} arbitrarily long.

Given that an embedded system has little memory, which information about a trace should be recorded? Surely, the concrete path from q_0 to $q_\perp$ should be recorded in order to explain *how* the specification was violated. The fact that in universal very weak automata, there are only finitely many different paths makes this possible. But we can actually record more: whenever a run moves forward along a path in the automaton, it intuitively moves *closer* towards violating the property (if the respective run ever reaches $q_\perp$). The values of the propositions along such transitions closer to $q_\perp$ then provide evidence on *why* the specification violation occurred.

For instance, when recording the proposition values for a violation along the path $q_0 \rightarrow q_1 \rightarrow q_\perp$, the values of a_1 and a_2 tell the system engineer whether an ambulance van override signal was involved in the violation or not, which helps pinpointing the cause of the violation. Similarly, if a violation via the path $q_0 \rightarrow q_9 \rightarrow \ldots \rightarrow q_{19} \rightarrow q_\perp$ is found and the proposition values along a violating run are recorded whenever the run switches states, the recorded information shows the (observable) traffic light states before stabilizing the green light for direction 1, and additionally how this stabilization was broken.

On a technical level, recording the proposition valuations along a path can be implemented by augmenting the monitor code variables tracking for each automaton state if a run can be in the state with a trace recording buffer. The buffer information needs to contain the prefix path and the proposition valuations for this prefix path. Whenever there exist multiple prefix runs leading to a state, an arbitrary of these is used for the stored buffer information.

While the choice to track proposition valuations when switching states in the universal automaton is a heuristic, and hence not always the most useful trace parts are recorded in this way, this heuristic has the advantage that the needed buffer sizes can be computed upfront and are constant, and we track proposition valuations at well-defined time points that have a clear reason for being *potentially* interesting, without a need for the engineer to specify a violation analysis strategy. In the example specification from above, we would need 130 extra bytes of RAM for storing the trace information for all states as well as persistent storage of 12 bytes for storing the trace information of an actual specification violation for later retrieval by the application engineer. This is even small enough for the low amount of EEPROM memory that low-cost microcontrollers have (such as the 512 bytes of EEPROM of the low-power STM32L010RB microcontroller by ST Microelectronics).

Finally, we note that if the specification automaton is not actually very weak, we can record the proposition valuations of a run at the time points at which it switches between strongly connected components of the automaton. Furthermore, it needs to be noted that if the universal automaton has bad prefixes along which a state with an empty language is not reached for some run, no violation reason can be recorded for such a case.

5 A Specialized Microcontroller Component for Temporal Logic Runtime Monitoring

The monitor based on universal automata built in Section 3 needs 549 clock cycles on average per monitor step on the example trace. While this is substantially less than for the other monitor types that we looked at, the number of cycles is still relatively large, and for simple applications, the overall monitoring overhead can easily exceed the number of clock cycles used for the core functionality of the system. This is caused by the many Boolean operations performed on the proposition values, for which even an optimizing compiler can only cache a few. Each such Boolean operation needs at least one clock cycle (as simple microcontrollers cannot parallelize basic operations).

This efficiency problem can be addressed by *hardware support* for temporal logic monitoring. Microcontrollers often have many components apart from a processor core and memory, and they communicate via one or more shared busses, to which a runtime monitoring component could be connected. As a proof of concept, we implemented such a component. The component was also accepted for fabrication with the Google-sponsored *Open MPW* program [15], and we describe its main ideas in the following.

The starting point of our monitoring component is the observation that monitor program code, such as the one in Figure 1, can be represented as a *combinatorial circuit*, which takes as input the current proposition valuation and the previous values of the single-bit variables representing in which states a run of the universal automaton can be, and outputs the next values of these state variables (of which one represents if a violation has been observed so far). The circuit can be minimized based on the reachable state combinations in the automaton as *careset*, for instance with the approach by Lee et al. [24]. A number of flip-flops looping back the state variables then complement the combinatorial circuit to a runtime monitor.

In order to simulate the behavior of such a combinatorial circuit, we do not have to perform the Boolean operations in a sequential way. Rather, we can group multiple circuit gates together and compute their output values simulataneously by *table lookups*. Figure 3 depicts the idea. We split simulating the circuit into a sequence of table lookups, each using one or more input bits and producing one or more output bits. At the end of a sequence of table lookups, we obtain the new values for the state variables.

The runtime monitoring component, available under an open source license from `https://github.com/progirep/temporal_runtime_monitor_`

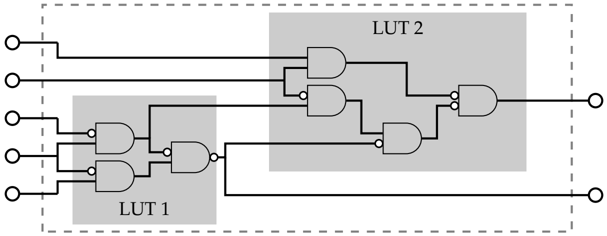

Fig. 3. Graphical depiction of the idea to map circuit elements to lookup tables (LUTs). The example circuit does not originate from an actual monitoring problem.

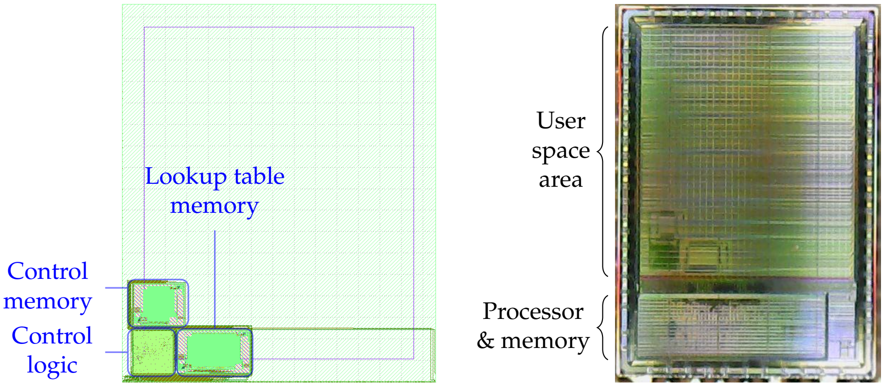

Fig. 4. The temporal logic runtime monitoring component in hardware. The left hand-side shows the utilization of the user space area on the Caravel SoC, where the runtime monitoring component only needs a small part of it. The right-hand side shows a photo of the manufactured system on chip, which includes the user space area at the top.

`for_caravel`, was designed to interface to the *Wishbone* bus of a *Caravel* System-on-chip (SoC) [14]. New proposition values can be provided to the component by a single 32-bit write access to the bus, and the component computes the result of the monitor step function while the processor core can already continue with its main functionality computation. The component has a 64-bit main state register that is continuously updated with each table lookup. Every table lookup takes 8 clock cycles, in which the following actions are performed:

1. A state register bit selection for the input to the lookup table as well as a lookup table starting address are read from a 1 kilobyte large *control memory* block.
2. The state register bits selected are extracted from the state register using a *bit extract* [40] operation.
3. Using the extracted bits as table row selection and the lookup table starting address, the table lookup is performed.

4. Another bit mask is read from the control memory, and the state register bits are compressed using another bit extract operation on the bit mask. The result of the table lookup from the previous step is then added to the state register content.

We implemented the component with 2 kilobyte lookup table memory, and it was manufactured in a 130 nanometer feature size IC production process. The right-hand side of Figure 4 shows the resulting chip, consisting of the SoC processor and memory at the bottom and a user space area for the added functionality of the SoC at the top. The user space area measures approximately 10 square millimeters, and only a small part of it was actually needed for the monitoring component. A design drawing of the user space area is also shown on the left-hand side of the figure.

A control register of the component stores information on the number of propositions used as well as the number of lookup tables. The component also allows write access to its memories to fill the them with the configuration needed to monitor for the specification of interest. An open-source compiler for encoding a circuit and a split of the circuit into lookup tables into the configuration encoding needed by the component is available at `https://github.com/progirep/monitor_compiler_for_caraval_monitor`. How to automatically optimize such a split based on a given monitoring circuit is left for future work. For the specification from the running example, a manual split into 7 lookup tables yields an encoding that needs 1504 bytes of lookup table memory and 144 bytes of control memory. As the monitoring component needs 8 clock cycles per lookup table (plus 8 cycles to filter the final state register content to those bits needed for its next step), a monitoring step of this component takes fewer than the 549 cycles needed by the software monitor based on a universal automaton.

We experimentally validated that the component actually works in silicon, except that the memory blocks, which were taken off-the-shelf, are unable to hold their values. When running the component with the randomly initialized but stable values of the memories, the results match those from a high-level simulation run on the memory contents read out from the component, indicating that when replacing the memory blocks or otherwise improving the stability of their operation, we obtain a working temporal logic runtime monitoring microcontroller component.

In its current version, the runtime monitoring component does not feature the specification violation information tracking approach from Section 4. The component is however not actually restricted to runtime monitoring, but can also be used for other computations that map nicely to simulating a sequential circuit.

6 Conclusion

In this paper, we gave a quick run-through of some existing results on compiling temporal logic specifications to runtime monitoring code. The focus was on using future-time temporal logic, as common in verification, while targeting tiny systems (i.e., embedded systems with a low-cost microcontroller). It should be noted that there is a large volume of related work that could not be discussed, such as for instance on offline runtime monitoring, multi-valued semantics for monitoring, automatic code instrumentation for monitoring, and runtime monitoring in a separate thread in more complex embedded systems.

In our run-through, we focused on closing the gap between formal verification and runtime monitoring by employing a specification formalism that is common in formal verification for runtime monitoring as well. We discussed two ways for making runtime monitoring more useful, hopefully encouraging system designers to write formal specifications more often. In particular, we showed how tiny systems can record information about a specification violation without the engineer having to declare which trace information is to be recorded. Furthermore, we described an in-silicon temporal logic runtime monitoring component that can be integrated into future microcontrollers for light-weight monitoring.

Multiple directions of future work can be identified from the discussions and results in this paper. For instance, general purpose tools for encoding computation that is easy to represent as a circuit to efficient program code to be executed by regular processors can make runtime monitoring on such processors more efficient. Then, automaton minimization approaches that reduce the sizes of transition labels of universal co-Büchi automata can also make monitors built from them more efficient to execute. For runtime monitoring in hardware, an approach for optimally partitioning a combinatorial circuit into a sequence of lookup tables is also needed. Finally, a monitoring approach that uses regular linear time temporal logic for specifications while adding a monitoring-specific formalism that allows the engineer to define which runtime information is most useful for debugging specification violations may make runtime monitoring more attractive for debugging the early versions of embedded systems in the field.

Acknowledgements

The author thanks Google for sponsoring the Open MPW program [15], in which the monitoring component described in Section 5 was manufactured, and Dr. Ivan Xavier Torres Cordova for taking the photo in Figure 4.

References

1. Adabala, K., Ehlers, R.: A fragment of linear temporal logic for universal very weak automata. In: 16th International Symposium on Automated Technology for Verification and Analysis (ATVA). Lecture Notes in Computer Science, vol. 11138, pp. 335–351. Springer (2018)
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed computing **2**(3), 117–126 (1987)
3. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457. Springer (2018)
4. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci and Falcone [3], pp. 1–33
5. Bartocci, E., Manjunath, N., Mariani, L., Mateis, C., Nickovic, D.: Automatic failure explanation in CPS models. In: 17th International Conference on Software Engineering and Formal Methods (SEFM). Lecture Notes in Computer Science, vol. 11724, pp. 69–86. Springer (2019)
6. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. **20**(3), 651–674 (2010)
7. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011)
8. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA stream-monitoring of real-time properties. ACM Trans. Embed. Comput. Syst. **18**(5s), 88:1–88:24 (2019)
9. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.J.: Explaining counterexamples using causality. Formal Methods Syst. Des. **40**(1), 20–40 (2012)
10. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Sampling-based runtime verification. In: 17th International Symposium on Formal Methods (FM). Lecture Notes in Computer Science, vol. 6664, pp. 88–102. Springer (2011)
11. Chen, F., Rosu, G.: Java-MOP: A monitoring oriented programming environment for Java. In: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 3440, pp. 546–550. Springer (2005)
12. d'Amorim, M., Rosu, G.: Efficient monitoring of omega-languages. In: Etessami, K., Rajamani, S.K. (eds.) 17th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 3576, pp. 364–378. Springer (2005). https://doi.org/10.1007/11513988_36
13. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From spot 2.0 to spot 2.10: What's new? In: Shoham, S., Vizel, Y. (eds.) 34th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer (2022)
14. Efabless Corporation: Caravel system on chip (2021), https://github.com/efabless/caravel
15. Efabless Corporation: The open MPW program (2022), https://efabless.com/open_shuttle_program
16. Ehlers, R., Finkbeiner, B.: Monitoring realizability. In: Second International Conference on Runtime Verification (RV). Lecture Notes in Computer Science, vol. 7186, pp. 427–441. Springer (2011)
17. Ehlers, R.: Runtime monitoring example data / TAP 2024 (2024). https://doi.org/10.5281/zenodo.12789827

18. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: Streamlab: Stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) 31st International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 11561, pp. 421–431. Springer (2019). `https://doi.org/10.1007/978-3-030-25540-4`
19. Ferrère, T., Maler, O., Nickovic, D.: Trace diagnostics using temporal implicants. In: 13th International Symposium on Automated Technology for Verification and Analysis (ATVA). Lecture Notes in Computer Science, vol. 9206, pp. 241–258. Springer (2015)
20. Havelund, K., Rosu, G.: Monitoring Java programs with Java PathExplorer. Electron. Notes Theor. Comput. Sci. **55**(2), 200–217 (2001)
21. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 2280, pp. 342–356. Springer (2002)
22. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods Syst. Des. **19**(3), 291–314 (2001)
23. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS). pp. 531–542. IEEE (2005). `https://doi.org/10.1109/SFCS.2005.66`
24. Lee, S.Y., Riener, H., De Micheli, G.: External don't cares in logic synthesis. In: Advanced Boolean Techniques: Selected Papers from the 15th International Workshop on Boolean Problems. pp. 33–47. Springer (2023). `https://doi.org/10.1007/978-3-031-28916-3_3`
25. Maidl, M.: The common fragment of CTL and LTL. In: 41st Annual Symposium on Foundations of Computer Science (FOCS). pp. 643–652 (2000)
26. Medhat, R., Kumar, D., Bonakdarpour, B., Fischmeister, S.: Sacrificing a little space can significantly improve monitoring of time-sensitive cyber-physical systems. In: ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS). pp. 115–126 (2014)
27. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Formal Methods Syst. Des. **51**(1), 31–61 (2017)
28. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Copilot: monitoring embedded systems. Innov. Syst. Softw. Eng. **9**(4), 235–255 (2013)
29. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57 (1977)
30. Reinbacher, T., Brauer, J., Horauer, M., Steininger, A., Kowalewski, S.: Past time LTL runtime verification for microcontroller binary code. In: 16th International Workshop on Formal Methods for Industrial Critical Systems (FMICS). Lecture Notes in Computer Science, vol. 6959, pp. 37–51. Springer (2011)
31. Reinbacher, T., Horauer, M., Steininger, A.: A runtime verification unit for microcontrollers. In: Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference. pp. 1–6 (2012)
32. Snook, C.F.: Exploring the barriers to formal specification. Ph.D. thesis, University of Southampton, UK (2001), `https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.268626`
33. Solet, D., Béchennec, J., Briday, M., Faucou, S., Pillement, S.: Hardware runtime verification of embedded software in sopc. In: 11th IEEE Symposium on Industrial Embedded Systems (SIES). pp. 171–176. IEEE (2016)

34. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: 12th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 1855, pp. 248–263. Springer (2000)
35. Tabakov, D., Rozier, K.Y., Vardi, M.Y.: Optimized temporal monitors for systemc. Formal Methods Syst. Des. **41**(3), 236–268 (2012)
36. Vaandrager, F.W.: Model learning. Commun. ACM **60**(2), 86–95 (2017)
37. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
38. Walters, G., King, E., Kessinger, R., Fryer, R.: Processor design and implementation for real-time testing of embedded systems. In: 17th DASC. AIAA/IEEE/SAE Digital Avionics Systems Conference. vol. 1, pp. B44–1. IEEE (1998)
39. Wang, S., Geoffroy, Y., Gößler, G., Sokolsky, O., Lee, I.: A hybrid approach to causality analysis. In: 6th International Conference on Runtime Verification (RV). Lecture Notes in Computer Science, vol. 9333, pp. 250–265. Springer (2015)
40. Wolf, C.X.: Reference hardware implementations of bit extract/deposit instructions (2017), `https://github.com/cliffordwolf/bextdep`