Sampling-based Decentralized Monitoring for Networked Embedded Systems

Ezio Bartocci
Institute of Computer Engineering
Vienna University of Technology
Vienna, Austria
ezio.bartocci@tuwien.ac.at

Decentralized monitoring (DM) refers to a monitoring technique, where each component must infer, based on a set of partial observations if the global property is satisfied. Our work is inspired by the theoretical results presented by Baurer and Falcone at FM 2012 [7], where the authors introduced an algorithm for distributing and monitoring LTL formulae, such that satisfaction or violation of specifications can be detected by local monitors alone. However, their work is based on the main assumption that neither the computation nor communication take time, hence it does not take into account how to set a sampling time among the components such that their local traces are consistent. In this work we provide a timed model in UPPAAL and we show a case study on a networked embedded systems board.

1 Introduction

The majority of all computing devices produced nowadays, are embedded systems employed to monitor and control physical processes: cars, airplanes, automotive highway systems, air traffic management, etc.. In all these scenarios, computing and communicating devices, sensors monitoring the physical processes and the actuators controlling the physical substratum are distributed and interconnected together in dedicated networks. In order to verify the correct behavior of these systems at runtime, the user often needs to monitor the emergent behavior of these autonomous systems perceiving them as monolithic system, where the global behavior is the result of all the local behaviors. The property to be observed is usually specified in terms of linear-time temporal logic [32] (LTL) formulae or as a finite state machine accepting the language of all the traces satisfying the property of interest. The observation of the system can follow two different approaches. The first is the *centralized* observation, where all the events generated by the local components (i.e. sensors values) must be sent to a central dedicated component that collects the local traces, orders them in a global trace and monitors the property of interest. In many real-world applications, where both the communication and the number of components need to be kept minimal, this approach is not feasible for practical and economical reason. An alternative method is the *decentralized* monitoring, where each components must infer, based on a set of partial observations if the global property is satisfied. Our work is inspired by the theoretical results presented by Baurer at al. at FM 2012 [7], introducing an algorithm for distributing and monitoring LTL formulae, such that satisfaction or violation of specifications can be detected by local monitors alone. In their paper the monitoring is performed using a technique also known as formula progression [35, 7, 3], where the LTL formula is rewritten into a new formula expressing what needs to be satisfied by the current observation and a new formula which has to be satisfied by the trace in the future. In the decentralized setting, the progression is performed by each component equipped with a rewriting engine. In this case the monitoring may involve the exchange, with the other components, of messages containing the rewritten LTL formula with past obligations on the events not directly observable by the local component. Unfortunately, we found that their very elegant theoretical results are hard to implement in real-time embedded systems for their main assumption in which neither the computation nor communication take time. For example, the sampling time with which the events are observed must be consistent among the components and during the monitoring. This time depends both on the communication media, the size of the messages exchanged, the worst-time execution of the formula progression. In this work we try to address these problems by providing a timed model in UPPAAL and we show a case study on a networked embedded systems board.

The paper is organized as follows: in Section 2 we discuss the related works and in Section 3 we introduce the background material. In Section 4 we present a timed model in UPPAAL for sample-based monitoring and in Section 5 we show a case study. The conclusion is in Section 6.

2 Related Work

Runtime verification (also called *monitoring*) [8, 24] is a lightweight yet powerful formal technique used to check whether the current execution of a program satisfies or violates a property of interest. This technique differs from the classical and more expensive *model checking* [16, 34] that aims instead to verify the correctness of the property exhaustively for all the possible program behaviors. Monitoring is generally used when the system model is too big to handle with *model checking* due to the state-explosion problem, or when the system model is not available, or it is a *black-box* where only the ouputs are observable. Furthermore, runtime verification can also be used to trigger some system recovery actions when a safety property is violated. If the system under scrutiny is distributed, multiple and decentralized monitoring processes [7, 22, 36, 39, 40, 41] can be employed to check if during the execution a global property is satisfied or not. In [36, 37] the authors describe an efficient decentralized monitoring algorithm, based on a variant of past time linear temporal logic, that monitors a distributed program's execution to check for violations of safety properties. However, their work does not deal with time constraints and does not address real-time applications running on networked embedded systems. Baur and Falcone propose in [7] an algorithm for decentralized LTL monitoring in synchronous systems based on formula progression/rewriting, but also in that work, neither the overhead of monitoring and the computation time are taken into account. In the last years, several techniques have been developed to control the overhead [14, 38, 6, 25] of monitoring. The majority of these techniques involve the use of *event-triggered* monitors, where the monitor is invoked whenever a new event is triggered by the system, making the overhead unpredictable. Our approach is based on *synchronous sampling* and it is similar to the one introduced in [12], but extended for the case of networked embedded systems, where the local monitors take samples from the local program variables and the sensors to analyze if a property of interest is satisfied or not.

3 Background

In the *decentralized monitoring* setting considered in this paper, we assume a set $\mathscr{C} = \{C_1, C_2, \cdots, C_n\}$ of components communicating on a serial communication BUS as Fig. 1 shows. Each component C_i is equipped with a monitor M_i that can observe the set of events Σ_i. $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \cdots \cup \Sigma_n$, is the set of all events. If each event can be observed only by a single component, we assume that $\Sigma_i \cap \Sigma_j = \emptyset$ for all $i, j \leq n$ with $i \neq j$. The property to be monitored is specified in a LTL [32] formula φ over a set of propositions AP, such that $\Sigma = 2^{AP}$. We denote with $\tau_i(m)$ the (m+1)-th event in the *local trace* τ_i

observed by the monitor M_i and with $\tau = (\tau_1, \tau_2, \cdots, \tau_n)$ the *global trace* such that $\tau = \tau_1(0) \cup \tau_2(0) \cup \cdots \tau_n(0) \cdots \tau_1(t) \cup \tau_2(t) \cup \cdots \tau_n(t)$. Each component is an embedded computing device that can access the values of a set of external sensors (i.e. the external temperature or the button pressure) and can control through a program a set of actuators (i. e. fan speed or the temperature of the heater). Each monitor observes the change of the local sensors and the program values together with the events that may receive from the other components through a serial communication BUS, enabling the exchange of multicast messages.

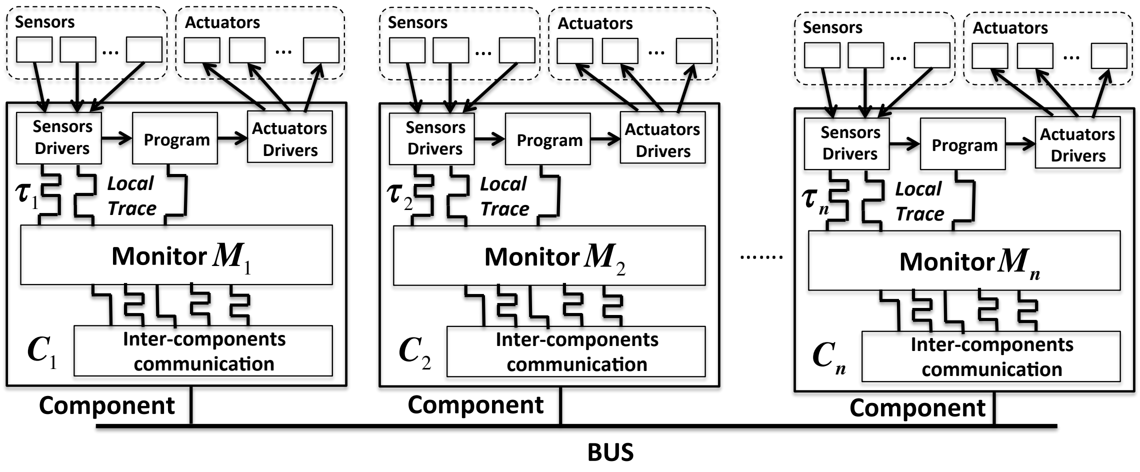

Figure 1: Decentralized monitoring setting for a distributed embedded systems.

Definition 3.1 (Linear Temporal Logic (LTL) Syntax [32]) *The syntax for an LTL formula is described by the following grammar:*

$$\varphi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi$$

where $p \in AP$. A LTL formula has atomic propositions p, logical connectives $\neg$ $\wedge$, temporal operators **X** *(next),* **U** *(until). As usual, we introduce shorthands by defining the following derivative logical and temporal operators:*

$\psi \vee \varphi$	$\Leftrightarrow$	$\neg(\neg\psi \wedge \neg\varphi)$	*or* $(\vee)$
$\psi \rightarrow \varphi$	$\Leftrightarrow$	$\neg\psi \vee \varphi$	*implies* $(\rightarrow)$
$\psi \leftrightarrow \varphi$	$\Leftrightarrow$	$(\psi \rightarrow \varphi) \wedge (\varphi \rightarrow \psi)$	*equivalent* $(\leftrightarrow)$
$\Box\psi$	$\Leftrightarrow$	$\neg(\bot\, \mathbf{U} \neg\psi)$	*always* $(\Box)$
$\Diamond\psi$	$\Leftrightarrow$	$\top\, \mathbf{U} \psi$	*eventually* $(\Diamond)$

Definition 3.2 (Linear Temporal Logic (LTL) Semantics [32]) *Let be $\tau = \tau_1(0) \cup \tau_2(0) \cup \cdots \tau_n(0) \cdots \tau_1(t) \cup \tau_2(t) \cup \cdots \tau_n(t) \cdots \in \Sigma^\omega$ (global trace) an infinite word with $i \in \mathbb{N}$ being a position corresponding to a particular time step. Then the semantics of an LTL formula is defined inductively as follows:*

$$\begin{array}{lcl}
\tau, i \models \top \textit{ holds} & \textit{(is true)}. & \\
\tau, i \models \bot & \Leftrightarrow & \tau, i \not\models \top \\
\tau, i \models p & \Leftrightarrow & p \in \tau_1(i) \cup \tau_2(i) \cup \cdots \tau_n(i) \\
\tau, i \models \neg\varphi & \Leftrightarrow & \tau, i \not\models \varphi \\
\tau, i \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow & \tau, i \models \varphi_1 \textit{ and } \tau, i \models \varphi_2 \\
\tau, i \models \mathbf{X}\varphi & \Leftrightarrow & \tau, i+1 \models \varphi \\
\tau, i \models \varphi_1 \mathbf{U} \varphi_2 & \Leftrightarrow & \exists k \geq i\ \tau, k \models \varphi_2 \textit{ and } \forall i \leq l < k\ \tau, l \models \varphi_1
\end{array}$$

Moreover, $\tau \models \varphi$ *holds* $\Leftrightarrow \tau, 0 \models \varphi$.

We denote $\mathscr{L}(\varphi) = \{w \in \Sigma^\omega | w \models \varphi\}$ as the language generated by an LTL-formula φ and corresponding to a set of models of a LTL-formula φ. The languages generated by two formulae φ and ψ are the same $\mathscr{L}(\psi) = \mathscr{L}(\varphi)$ iff $\varphi \equiv \psi$. A common technique to verify the correctness of a property is to generate a monitor from a LTL-formula. Such a monitor can be then executed in parallel with the application to be verified at runtime (*online synchronous monitoring*) or can be used after the program execution to check a finite set of recorded executions (*offline monitoring*). There are two main approaches to generate synchronous monitors. The first method relies on the generation of *automata-based* monitors. In particular, there are several papers [19, 20, 42] describing how to build a reduced nondeterministic Büchi automaton [13] able to recognize infinite words of the language $\mathscr{L}(\varphi)$ of a LTL formula φ. A Büchi automaton can be then turned to a monitor [17, 21] in the form of a deterministic finite state machine (DFSM). Generally, the process of converting a LTL formula into a monitor is expensive and the size of the Büchi automata generated can be $2^{O(|\varphi|)}$ [20]. However, once the monitor is generated, its execution can be very efficient. In particular, Rosu et. at. showed in [17] how to build particular DFSMs called *binary transition tree finite state machines* (BTT-FSM) that perform a transition from a state to another state of the monitor by evaluating an optimal number of atomic propositions.

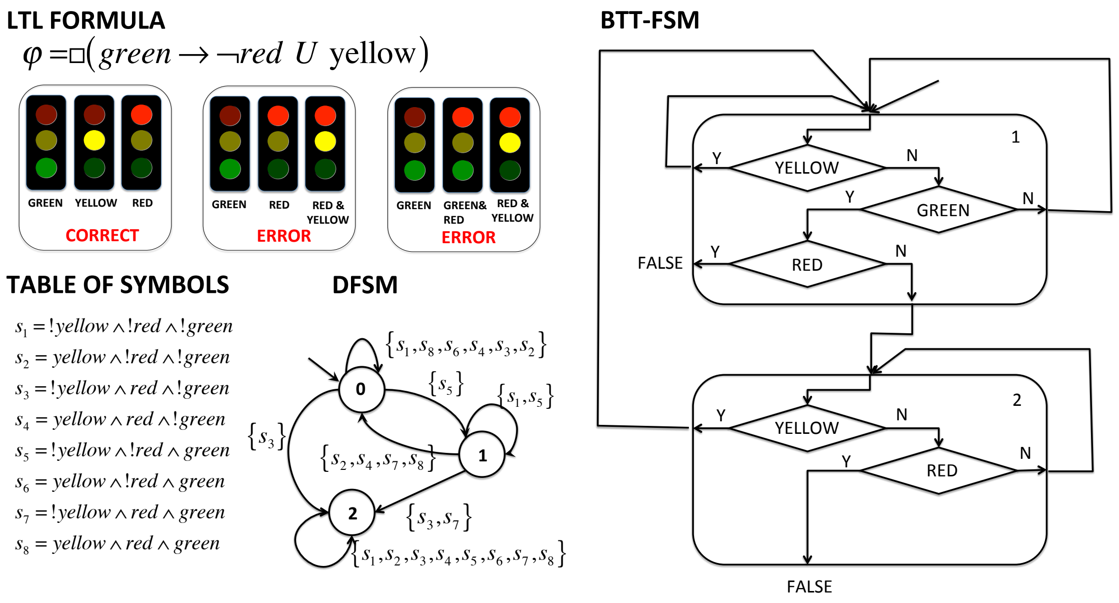
LTL FORMULA
$\varphi = \Box(green \rightarrow \neg red\ U\ \text{yellow})$
GREEN
YELLOW
RED
CORRECT
GREEN
RED
RED & YELLOW
ERROR
GREEN
GREEN& RED
RED & YELLOW
ERROR
TABLE OF SYMBOLS
$s_1 = !yellow \wedge !red \wedge !green$
$s_2 = yellow \wedge !red \wedge !green$
$s_3 = !yellow \wedge red \wedge !green$
$s_4 = yellow \wedge red \wedge !green$
$s_5 = !yellow \wedge !red \wedge green$
$s_6 = yellow \wedge !red \wedge green$
$s_7 = !yellow \wedge red \wedge green$
$s_8 = yellow \wedge red \wedge green$
DFSM
0
1
2
$\{s_1, s_8, s_6, s_4, s_3, s_2\}$
$\{s_5\}$
$\{s_1, s_5\}$
$\{s_3\}$
$\{s_2, s_4, s_7, s_8\}$
$\{s_3, s_7\}$
$\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$
BTT-FSM
1
YELLOW
GREEN
RED
Y
N
FALSE
2
YELLOW
RED
Y
N
FALSE

Figure 2: Automata-based monitors to check the property of a traffic light that always when it is green, then it is not red until it is yellow.

An alternative monitoring approach is based on *formula rewriting* [35, 7] or *formula progression* [3]. The monitor in this case is a rewriting engine, that rewrites the current formula into a new formula expressing what needs to be satisfied by the current observed events and what are the future obligations to meet. The overhead required for monitoring with this approach is higher than by using a DFSM. On the other hand this method is more flexible, because does not require a process of translation from LTL formula to monitor and allows to change at runtime the formula to be monitored. In the following, we provide some basic definitions for the LTL rewriting function and the monitoring result.

Definition 3.3 (LTL rewriting function[7]) *Let S_{LTL} be the set of all the possible LTL formulae and $\phi, \phi_1, \phi_2 \in S_{LTL}$, $\sigma \in \Sigma$ an event, the LTL rewriting function $R : S_{LTL} \times \Sigma \to S_{LTL}$ is inductively defined as follows:*

$$\begin{array}{rcl} R(\top,\sigma) & = & \top \\ R(\bot,\sigma) & = & \bot \\ R(\neg\phi,\sigma) & = & \neg R(\phi,\sigma) \\ R(\mathbf{X}\phi,\sigma) & = & \phi \end{array} \qquad \begin{array}{rcl} R(p \in AP,\sigma) & = & \top, \textit{ if } p \in \sigma, \bot \textit{ otherwise} \\ R(\phi_1 \vee \phi_2,\sigma) & = & R(\phi_1,\sigma) \vee R(\phi_2,\sigma) \\ R(\phi_1 \mathbf{U} \phi_2,\sigma) & = & R(\phi_2,\sigma) \vee R(\phi_1,\sigma) \wedge \phi_1 \mathbf{U} \phi_2 \\ R(\Box\phi,\sigma) & = & R(\phi,\sigma) \wedge \Box\phi \\ R(\diamond\phi,\sigma) & = & R(\phi,\sigma) \vee \diamond\phi \end{array}$$

Definition 3.4 (Monitoring[7]) *Let $u \in \Sigma^*$ denote a finite word. The evaluation of the* satisfaction relation, *$\models_3 : \Sigma^* \times LTL \to \mathbb{B}_3$, with $\mathbb{B}_3 = \{\top, \bot, ?\}$ of a formula φ with respect to u is defined as:*

$$u \models_3 \varphi = \begin{cases} \top & \textit{if} \quad \forall \sigma \in \Sigma^w : u\sigma \models \varphi \\ \bot & \textit{if} \quad \forall \sigma \in \Sigma^w : u\sigma \not\models \varphi \\ ? & \textit{otherwise} \end{cases}$$

4 A Timed Model for Decentralized Monitoring

In this section, we propose a timed model for the decentralized monitoring using networks of timed automata [1]. This formal specification allows us to analyze, with tools like UPPAAL [11], the timing behavior of the system and to check important properties such as the synchronization of the sampling, the sampling time and granularity. A timed automaton is a finite-state machine enriched with clock variables using a dense-time model. For the sake of completeness, in the following we provide all necessary definitions.

Definition 4.1 (Timed Automaton (TA) [11]) *A timed automaton is a tuple $\mathscr{A} = (L, l_0, C, \Sigma, E, I)$ where:*

- *L is a finite set of locations,*
- *$l_0 \in L$ is the initial location,*
- *C is a finite set called the clocks of $\mathscr{A}$,*
- *Σ is a finite set called the alphabet or actions of $\mathscr{A}$,*
- *$E \subseteq L \times \Sigma \times B(C) \times 2^C \times L$ is a set of edges, called transitions of $\mathscr{A}$, where $B(C)$ is the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$,*
- *$I : L \to B(C)$ assigns invariants to locations.*

Definition 4.2 (Semantics of TA[11]) *Let $\mathscr{A} = (L, l_0, C, \Sigma, E, I)$ be a timed automaton. The semantics is defined as a labelled transition system $\langle S, s_0, \to \rangle$, where:*

- $S \subseteq L \times \mathbb{R}^C$ *is the set of states,*
- $s_0 = (l_0, u_0)$ *is the initial state,*
- $\rightarrow \subseteq S \times (\mathbb{R}_{\geq 0} \cup \Sigma) \times S$ *is the transition relation such that:*
  - $(l,u) \xrightarrow{d} (l, u+d)$ *if* $\forall d' : 0 \leq d' \leq d \Longrightarrow u + d' \in I(l)$,
  - $(l,u) \xrightarrow{a}$ *(l',u') if there exists* $e = (l, \sigma, g, r, l') \in E$ *s.t.* $u \in g$, $u' = [r \longmapsto 0]u$, *and* $u' \in I(l')$,

  where for $d \in \mathbb{R}_{\geq 0}$, $u + d$ *maps each clock x in C to the value* $u(x) + d$, *and* $[r \longmapsto 0]u$ *denotes the clock valuation which maps each clock in r to 0 and agrees with u over* $C \setminus r$.

A network of timed automata [11] is defined as a parallel composition of timed automata over a common set of clocks and actions, consisting of n timed automata $\mathscr{A}_i = (L_i, l_i^0, C, \Sigma, E_i.I_i), 1 \leq i \leq n$. A location vector is a vector $\bar{l} = (l_1, \cdots, l_n)$. The invariant functions are composed in a common function over location vectors $\bar{l} = \wedge_i I_i(l_i)$. Following the notation in [11], we denote with $\bar{l}[l_i'/l_i]$ the vector where the ith element l_i of $\bar{l}$ is replaced by l_i'.

Definition 4.3 (Semantics of a network of Timed Automata [11]) *Let* $\mathscr{A}_i = (L_i, l_i^0, C, \Sigma, E_i, I_i)$ *be a network of n timed automata. Let* $\bar{l}_0 = (l_1^0, \cdots, l_n^0)$ *be the initial location vector. The semantics is defined as a transition system* $\langle S, s_0, \rangle$, *where* $S = (L_1 \times \cdots \times L_n) \times \mathbb{R}^C$ *is the set of states,* $s_0 = (\bar{l}_0, u_0)$ *is the initial state, and* $\rightarrow \subseteq S \times S$ *is the transition relation defined by:*

- $(\bar{l}, u) \xrightarrow{d} (\bar{l}, u+d)$ *if* $\forall d' : 0 \leq d' \leq d \Longrightarrow u + d' \in I(\bar{l})$.
- $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l_i'/l_i], u')$ *if there exists* $l_i \xrightarrow{\tau g r} l_i'$ *s.t.* $u \in g$, $u' = [r \longmapsto 0]u$ *and* $u' \in I(\bar{l}[l_i'/l_i])$.
- $(\bar{l}, u) \underset{\rightarrow}{a} (\bar{l}[l_j'/l_j, l_i'/l_i], u')$ *if there exist* $l_i \xrightarrow{c?g_i r_i} l_i'$ *and* $l_j \xrightarrow{c!g_j r_j} l_j'$ *s.t.* $u \in (g_i \wedge g_j)$, $u' = [r_i \cup r_j \longmapsto 0]u$ *and* $u' \in I(\bar{l}[l_j'/l_j, l_i'/l_i])$.

The UPPAAL standard semantics presented in Definitions 4.1, 4.2 and 4.3 includes neither the use of *bounded integer variables*, nor the use of *broadcast channels*. Variables allow to keep low the number of locations to handle, while the semantics of the broadcast channel does not require to have receivers synchronized and so is never blocking. In our timed-model we employ both of these UPPAAL extensions and we refer the reader to [11] for further details.

Fig. 3 shows the timed model[1] chosen for each component C_i in Fig. 1. The model provides two different possible behaviors depending on the value $\{0,1\}$ of the parameter $fault_t$. This parameter enables/disables a fault-tolerance mechanism called N modular redundancy (NMR) in which $2k+1$ modules perform a process (in this case the monitoring) and the result is processed by a voting system to produce a single output. For example, in triple modular redundancy (TMR) if any one of the three systems fails, the other two systems can correct and mask the fault. The other important parameters in the model are the worst-case execution time (WCET) of the tasks involved in the process. WCET measures the maximum time length a task could take to execute on a specific hardware platform. In our setting we consider the following parameters:

- $wcet_l$ is the WCET to sample all the new local events to be monitored,
- $wcet_e$ is the WCET to send a message with the changed events from one node to the others (note that the communication is multicast),
- $wcet_m$ is the WCET to monitor the events,

[1]The UPPAAL model can be downloaded at `www.eziobartocci.com/has/decentralized_monitoring.xml`

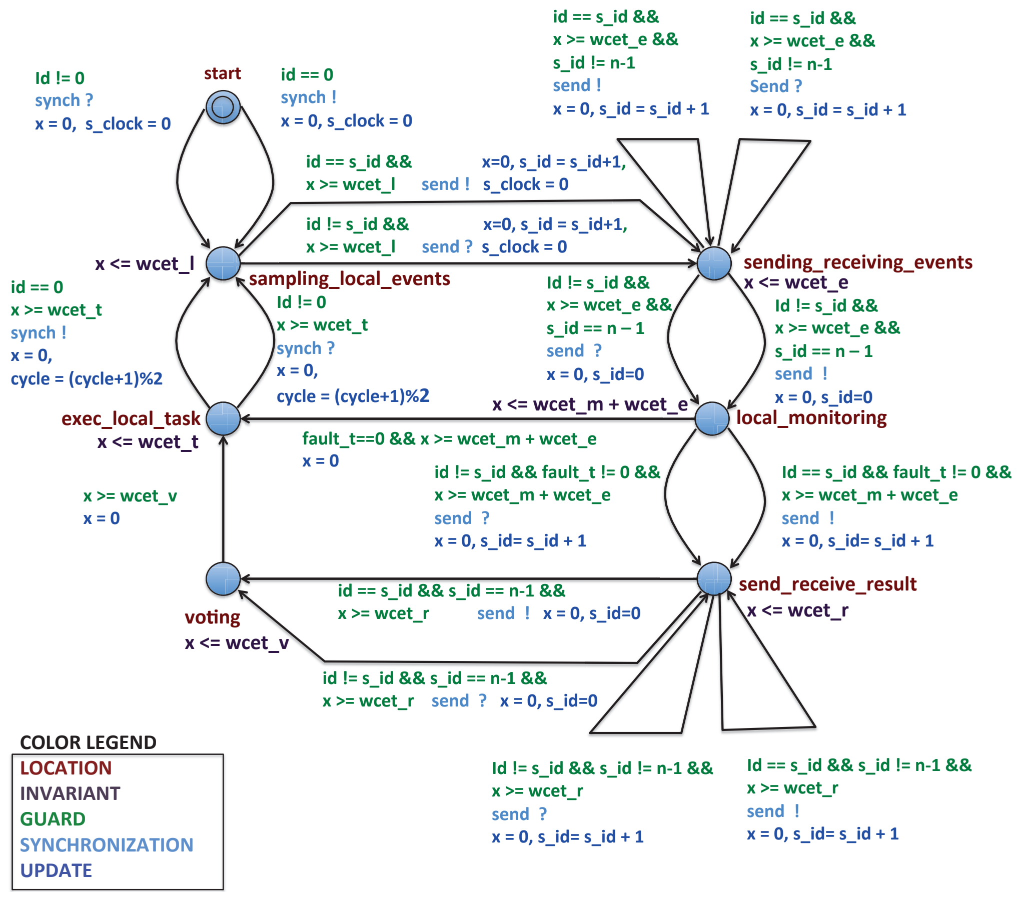

Figure 3: TA specification for each component involved in the decentralized monitoring.

- $wcet_r$ is the WCET of sending a message with the result from one node to the others,
- $wcet_v$is the WCET to perform the voting,
- $wcet_t$ is the WCET to execute a local task

Measuring the WCET is in the general case insoluble, because it is equivalent to the *halting problem*. However, in many particular cases (i.e. when the software does not contains infinite loops) is still possible to provide an over-approximation of such measure. The most common techniques to calculate the WCET are static analysis (by reasoning on the control graph, without executing the code) of the software or by runtime measuring the performances through the generation of appropriate test cases. All the WCET parameters are used to determine how much time each component C_i should stay in a particular location described in the timed model of Fig. 3. A clock x is used to keep track of the elapsed time in a location. This clock variable is reset when an enabled transition (representing an action) is taken and it is constrained with one of the WCET parameters mentioned before to determine the max time allowed in a particular location. Another clock variable s_{clock} keeps track of the time length elapsed between one sampling and the next one and it is used later to perform the analysis through model checking.
Two broadcast channels *synch* and *send* are used to realize the multicast communication. In a broadcast synchronization one sender with the actions *synch*! or *send*! can synchronize with an arbitrary number of receivers through the action *synch*? or *send*?. If a receiver in its current state has an enabled transition in which it can synchronize, it must do so. However, the broadcast sending is never blocking, so the

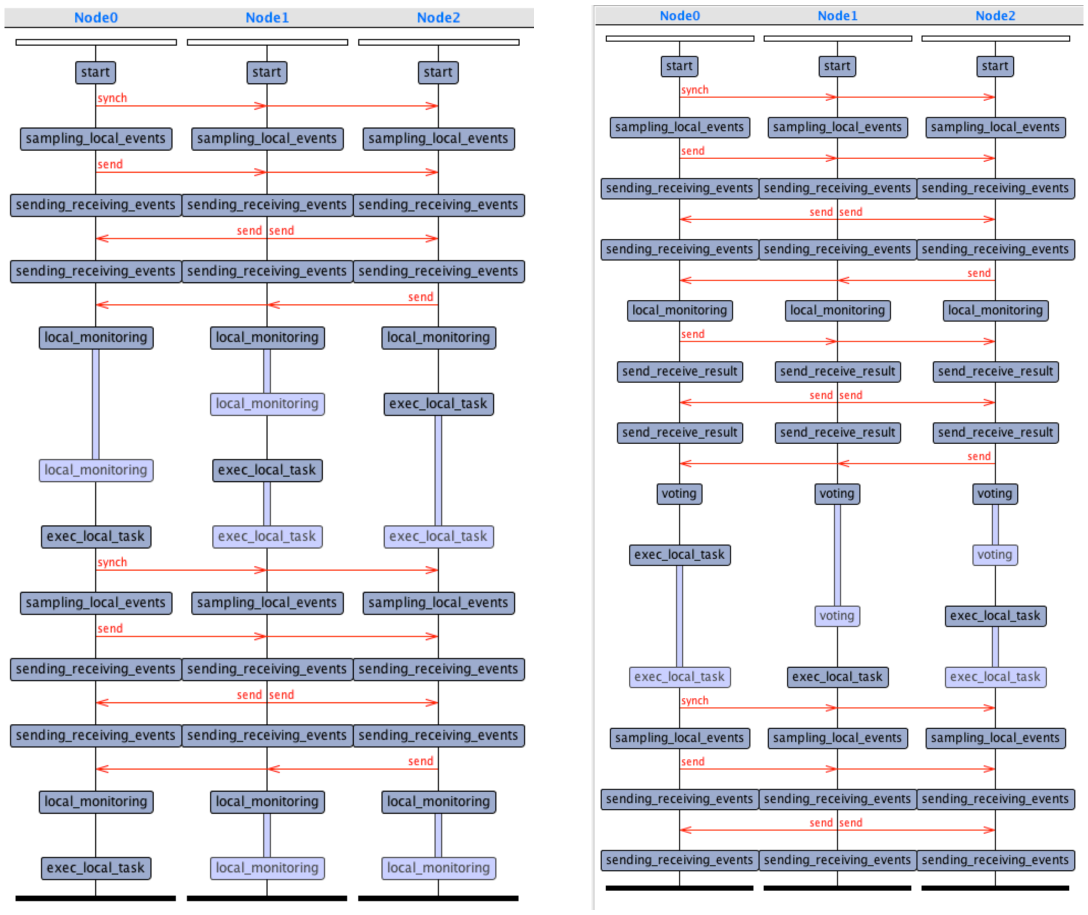

Figure 4: Two possible sequence (on the left $fault_t$=0, while on the right $fault_t$=1) diagrams generated using UPPAAL simulator for the model in Fig. 3.

sender can execute a synchronization action even if there are no receivers. A variable *cycle* is used for analysis purposes to mark the current cycle from the next and the previous one. Fig. 4 shows two possible execution traces of the system, one with the NMR enabled and one with NMR disabled. The timed automaton starts (*start* location) with a synchronization action *synch* sent always by the first component with $id == 0$ and received by the other components with $id! = 0$, respectively. Then in the *sampling_local_events* location within $wcet_l$ milliseconds all the local events in each nodes are sampled. The events that are changed in each node, are sent (in the location *sending_receiving_events*) to the others with n multicast messages, where n is the number of nodes. The order with which the nodes exchange their messages follows the order of their *id* (i.e. the node with lower *id* starts first). In the location *local_monitoring*, the local monitor processes the events and produces a result $\{\bot, \top, ?\}$. If the fault-tolerant mechanism is enabled ($fault_t$==1), the result is sent from each node to all the other nodes (in *send_receive_result*) and a voting mechanism will follow (*voting* location). A local task (i.e. displaying results, increase the heater temperature, etc..) can also be executed in the location *exec_local_task*, before the local sampling will start again the loop. In the following we shows some properties that are possible to be verified in the proposed timed model using UPPAAL tool.

Property 4.4 (Liveness) *When the timed automaton in Fig. 3 will enter the sampling_local_events location at the cycle* $= i$ *will then eventually enter the same location at cycle* $= (i+1)$ *mod* 2 *with* $i \in \{0,1\}$.

In UPPAAL this property can be expressed using the *leads to* or *response* form, written $\varphi \leadsto \psi$ which means whenever φ is satisfied, then ψ will be satisfied. It is possible to verify that the following liveness property $((\mathscr{C}_i.\text{sampling\_local\_events} \wedge cycle == 0) \leadsto (\mathscr{C}_i.\text{sampling\_local\_events} \wedge cycle == 1)) \wedge ((\mathscr{C}_i.\text{sampling\_local\_events} \wedge cycle == 1) \leadsto (\mathscr{C}_i.\text{sampling\_local\_events} \wedge cycle == 0))$ holds for each $1 \leq i \leq n$.

Property 4.5 (Synchronous sampling) *Given a network of n timed automata, there is not a reachable state, where one timed automaton is in the sampling_local_events location and the others in different locations at the same time. This means that the local events will be sampled by each component always synchronously.*

This property can be expressed in UPPAAL as the negation of a path formula (*exist eventually*) $\neg E\Diamond\varphi$: $\neg E\Diamond(\bigvee_{1\leq i\leq n}(\mathscr{C}_i.\text{sampling\_local\_events} \wedge (\bigvee_{j\neq i, 1\leq j\leq n} \neg\mathscr{C}_j.\text{sampling\_local\_events})))$

Proposition 4.6 (Sampling frequency) *Given a network of n components with the timed model shown in Fig. 3, the sampling frequency function* $f_s : \mathbb{N} \to \mathbb{R}$ *with which the local events are sampled is:*

$$f_s(n) = \frac{1}{wcet_l + n \cdot wcet_e + wcet_m + fault_t \cdot (n \cdot wcet_r + wcet_v) + wcet_t}$$

with $wcet_l$ *time units is also the time granularity and within this interval of time it is not possible to distinguish two different samples.*

In UPPAAL we can check that the sampling period is always constant, by verifying the following formula: $(\mathscr{C}_i.s_{clock} == 0) \leadsto (\mathscr{C}_i.sampling\_local\_events \wedge \mathscr{C}_i.s_{clock} == wcet_l + n \cdot wcet_e + wcet_m + fault_t \cdot (n \cdot wcet_r + wcet_v) + wcet_t)$

We verified the previous properties in UPPAAL by varying the number of components n from two to ten. However, we can generalize to an arbitrary number of nodes by making the following observations on the timed-model of Fig. 4 (here we consider only the case $fault_t = 0$, but the observations are similar for the case $fault_t = 1$):

1. **Start $\rightarrow$ sampling_local_events**. All n components are initialized in the *Start* location. The first transition is forced by the component $\mathscr{C}_0$ that synchronizes with a *synch* ! action all the other components $\mathscr{C}_i$ with $0 < i < n$ to switch, with a *synch* ? action (the only one enabled), into the new location *sampling_local_events* at the same time. There is no possibility that one component is in location *Start* and another is in location *sampling_local_events*.

2. **sampling_local_events$\rightarrow$ sending_receiving_events**. In this case all the components need to wait the same amount of time $wcet_l$ even if one finishes to sample the local events before another. After $wcet_l$ time there is only one action enabled for each component: *send* ! for $\mathscr{C}_0$ and *send* ? for $\mathscr{C}_i$ with $0 < i < n$. This step models $\mathscr{C}_0$ sending its local events update to all the other components.

3. **sending_receiving_events$\rightarrow$ sending_receiving_events**. A sequence of $n-2$ broadcast synchronization actions *send* ! and *send* ? will be enabled after waiting $wcet_e$ time each step. Incrementing the variable s_{id} from one to $n-2$ will distinguish the sender component $\mathscr{C}_{i=s_{id}}$ performing the synchronization action *send* ! from the receiver components $\mathscr{C}_{i\neq s_{id}}$ performing the action *send* ?. This sequence of synchronizations costs $(n-2)\cdot wcet_e$ time.

4. **sending_receiving_events$\rightarrow$ local_monitoring**. This transition is enabled only when $s_{id} = n-1$ and corresponds to the last component $\mathscr{C}_{i=n-1}$ sending its local events update to the other components after all having waited $wcet_e$ time.

5. **local_monitoring$\rightarrow$ exec_local_tasks**. This transition is performed by all the components without synchronization. Each component should wait in the location *local_monitoring* exactly $wcet_m + wcet_e$ time and then switch to the location *exec_local_tasks*.

6. **exec_local_tasks$\rightarrow$ sampling_local_events**. This case is similar to 1. The time spent in the location *exec_local_tasks* for each component is $wcet_t$. This transition makes sure that the *liveness* Property 4.4 holds.

By summing up the times spent in each location for a complete cycle, it is easy to show that also the Proposition 4.6 holds.

5 Case Study

The model presented in the previous section has been implemented on an hardware platform (designed in our lab) hosts with four independent micro-controllers (ATMega128 produced by Amtel) nodes connected to a real-time network. Each node is equipped with different peripheral devices as Fig. 5 shows. A shared communication BUS is included for Real-Time data transfer between the four nodes.
We chose the Carrier Sense Multiple Access (CSMA) with collision detection as our low level communication protocol among the nodes. This avoids that simultaneous messages are sent from the nodes and the messages do not require a fixed message length like in other protocols (like TTP [26]). It is possible to determine the WCET of the communication by analyzing the max length of the exchanged messages as Table 1 shows. The max length of the messages depends usually on the max number of atomic propositions that can change at runtime in one component.
We have adopted both the *automata-based* and the *formula progression monitoring* approaches. In the automata-based approach, we have used LTL3 tools[2] to generate the DFSM from a LTL formula and then coded the resulted state-machine in C, while to measure the monitoring overhead is possible to

[2]`http://ltl3tools.sourceforge.net`

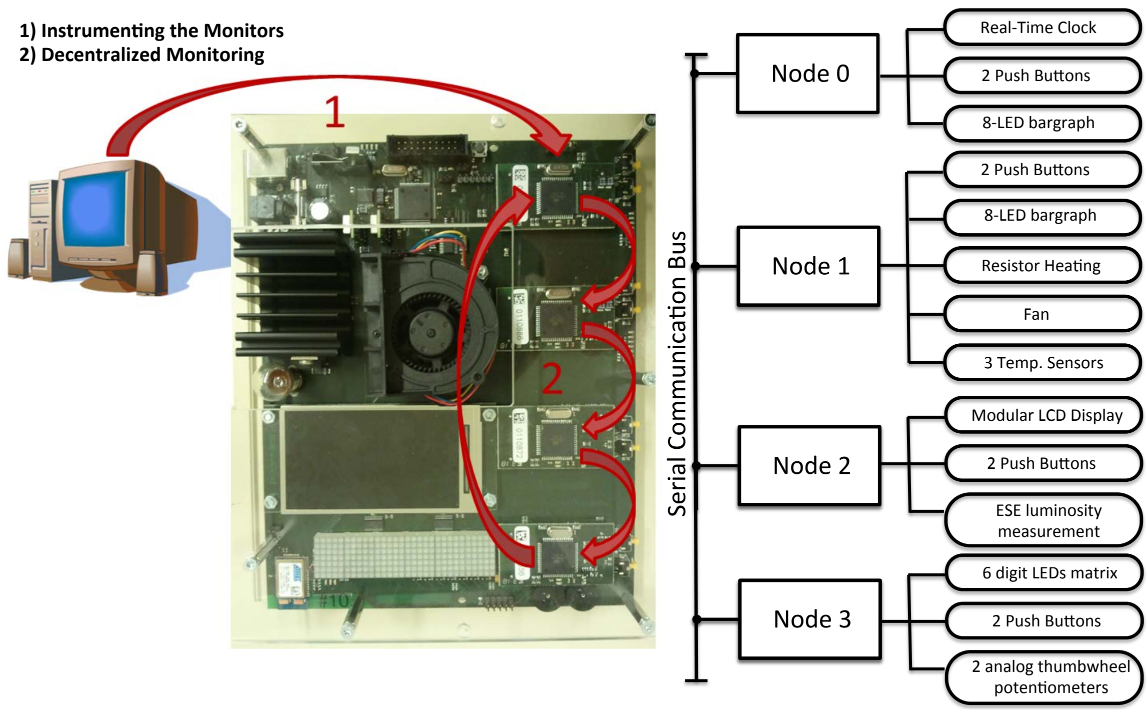

Figure 5: Networked Embedded Systems.

use a static analyzer for AMTEL micro-controller like Bound-T[3] . Concerning the formula-progression monitoring technique, even if we have imposed some limitations on the length of the formula (max 64 symbols) and on the number of next temporal operators allowed, the only way to measure the WCET is by measuring the elapsed time directly on the components.

Example 5.1 *We have implemented a simple heating control, where a resistor controlled by the node 1 heats up to 30 degrees and a fan is activated unless one of the two safety buttons controlled by node 0 are not pressed. We can specify the correct behavior using the following formula:*

$$\Box((!b_0 \vee !b_1) \wedge ((t > 30) \rightarrow (fan_{on})))$$

The size for the automata-based monitor is only of two states as Figure 6 shows. The monitor based on

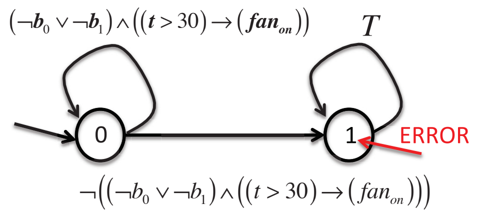
$(\neg b_0 \vee \neg b_1) \wedge ((t > 30) \rightarrow (fan_{on}))$
T
0
1
ERROR
$\neg((\neg b_0 \vee \neg b_1) \wedge ((t > 30) \rightarrow (fan_{on})))$

Figure 6: Automata-based monitor.

formula progression $R(\Box((!b_0 \vee !b_1) \wedge ((t > 30) \rightarrow (fan_{on}))), \sigma)$ will rewrite the formula as follows:

[3]`http://www.bound-t.com`

Property	Value	Description
max length of the token msg	66 bytes	1 byte for CSMA + 1 byte for the message length (4 bits) and token (4 bits) + 64 bytes for the data.
max length of the result msg	4 bytes	1 CSMA byte + 1 byte for the message length (4 bits) and token (4 bits) + 2 bytes for the data.
max length of the synch msg	1 bytes	1 CSMA byte
max num. bytes sent in one round	281 bytes	synch + 4 * token msg + 4 * result msg
max bits sent in one round	2810 bits	a byte sent contains 8 data bits, 1 start bit and 1 bit stop
Baud rate	4800 bit/s	
Worst Case Time for communication	0.585 sec	

Table 1: The Worst Case execution depends on the number of micro-controllers used, the max length of a token message exchanged, the max length of the result message and the baud rate of the BUS.

$$R((!b_0 \vee !b_1) \wedge ((t > 30) \rightarrow (fan_{on})), \sigma) \wedge \Box((!b_0 \vee !b_1) \wedge ((t > 30) \rightarrow (fan_{on})))$$

Hence, if the term $(!b_0 \vee !b_1) \wedge ((t > 30) \rightarrow (fan_{on}))$ is true, given the events in σ, then the resulting formula is:

$$\top \wedge \Box((!b_0 \vee !b_1) \wedge ((t > 30) \rightarrow (fan_{on})) = \Box((!b_0 \vee !b_1) \wedge ((t > 30) \rightarrow (fan_{on}))$$

otherwise, then the resulting formula is:

$$\bot \wedge \Box((!b_0 \vee !b_1) \wedge ((t > 30) \rightarrow (fan_{on})) = \bot$$

We have measured the $wcet_m$ time for the formula-progression monitoring of this example (that is also an upper bound for the automata-based monitor) counting the max number of CPU cycles needed with a prescaler (that divides the clock frequency) value set to 8. Considering that the clock speed of the micro-controller is 16 MHz, we have obtained that the rewriting worst case execution time for the formula is 65415 cycles $\times (1/(16MHz/8)) = 0.03237$ sec.

6 Conclusion

The synchronous decentralized monitoring of a networked embedded system requires some important assumptions about the synchronization mechanisms and the minimum sampling time to guarantee the time consistency among the monitored local traces. In this work we provide a possible timed model in UPPAAL for a sampling-based decentralized monitoring and we verify some important properties such as the *liveness*, the *synchronous sampling and the frequency*. We then provide a case study where we implement this timed model in our networked embedded systems testbed. Currently, we plan to extend our work in two directions. First, we would like to monitor properties expressed in more sophisticated temporal logics dealing with dense-time such as Metric Interval Temporal Logic (MITL) [2]. Secondly, since the synchronous communication becomes very computational expensive when the number of components increases, we plan to provide an asynchronous decentralized monitoring model, based on the Lamport's notion of global time [27].

7 Acknowledgement

We would like to thank the students Stephan Brugger, Dominik Macher and Daniel Schachinger that contribute in the implementation of the case study.

References

[1] R. Alur & D. L. Dill (1994): *A Theory of Timed Automata*. *Theor. Comput. Sci.* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[2] R. Alur, T. Feder & T. A. Henzinger (1996): *The Benefits of Relaxing Punctuality*. *Journal of ACM* 43(1), pp. 116–146, doi:10.1145/227595.227602.

[3] F. Bacchus & F. Kabanza (1998): *Planning for temporally extended goals*. *Annals of Mathematics and Artificial Intelligence* 22(1–2), pp. 5–27, doi:10.1023/A:1018985923441.

[4] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G.J. Pace, G. Rosu, O. Sokolsky & N. Tillmann (2010): *Preface*. In: *Proc. of RV 2010, the First International Conference on Runtime Verification, St. Julians, Malta, November 1-4, 2010*, *Lecture Notes in Computer Science* 6418, Springer, doi:10.1007/978-3-642-16612-9.

[5] H. Barringer, D. Rydeheard & K. Havelund (2010): *Rule Systems for Run-Time monitoring: From Eagle to RuleR*. *Journal of Logic and Computation* 20(3), pp. 675–706, doi:10.1093/logcom/exn076.

[6] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok & J. Seyster (2012): *Adaptive Runtime Verification*. In: *Proc. of RV 2012, the third International Conference on Runtime Verification, September, 2012 Istanbul, Turkey*, *Lecture Notes in Computer Science* 7687, Springer, pp. 168–182, doi:10.1007/978-3-642-35632-2_18.

[7] A. Bauer & Y. Falcone (2012): *Decentralised LTL monitoring*. In: *FM 2012: Formal Methods*, 7436, Springer Berlin Heidelberg, pp. 85–100, doi:10.1007/978-3-642-32759-9_10.

[8] A. Bauer, M. Leucker & C. Schallhart (2006): *Monitoring of real-time properties*. In: *Proc. of FSTTCS, the 26th Conference on Foundations of Software Technology and Theoretical Computer Science*, *Lecture Notes in Computer Science* 4337, Springer-Verlag, Berlin, Heidelberg, doi:10.1007/11813040_37.

[9] A. Bauer, M. Leucker & C. Schallhart (2010): *Comparing LTL semantics for runtime verification*. *Journal of Logic and Computation* 20(3), pp. 651–674, doi:10.1093/logcom/exn075.

[10] A. Bauer, M. Leucker & C. Schallhart (2011): *Runtime verification for LTL and TLTL*. *ACM Transactions on Software Engineering and Methodology* 20(4), doi:10.1145/2000799.2000800.

[11] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi & M. Hendriks (2006): *UPPAAL 4.0*. In: *Proc. of QEST 2006, the Third International Conference on the Quantitative Evaluation of Systems, Riverside, California, USA*, IEEE Computer Society, pp. 125–126, doi:10.1109/QEST.2006.59.

[12] B. Bonakdarpour, S. Navabpour & S. Fischmeister (2011): *Sampling-Based Runtime Verification*. In: *Proc. FM 2011: Formal Methods, the 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011*, *Lecture Notes in Computer Science* 6664, Springer, pp. 88–102, doi:10.1007/978-3-642-21437-0_9.

[13] J. R. Büchi (1990): *On a decision method in restricted second order arithmetic*. In: *The Collected Works of J. Richard Büchi*, Springer New York, pp. 425–435, doi:10.1007/978-1-4613-8928-6_23.

[14] S. Callanan, D. J. Dean, M. Gorbovitski, R. Grosu, J. Seyster, S. A. Smolka, S. D. Stoller & E. Zadok (2008): *Software monitoring with bounded overhead*. In: *Proc. of IPDPS 2008, the 22nd IEEE International Symposium on Parallel and Distributed Processing, Miami, Florida USA, April 14-18, 2008*, IEEE, pp. 1–8, doi:10.1109/IPDPS.2008.4536433.

[15] F. Cassez (2012): *The Complexity of Codiagnosability for Discrete Event and Timed Systems*. *IEEE Transactions on Automatic Control* 57(7), pp. 1752–1764, doi:10.1109/TAC.2012.2183169.

[16] E. M. Clarke & E. Emerson (1982): *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic*. In Dexter Kozen, editor: *Logics of Programs*, *Lecture Notes in Computer Science* 131, Springer Berlin / Heidelberg, pp. 52–71, doi:10.1007/BFb0025774.

[17] M. d'Amorim & G. Rosu (2005): *Efficient Monitoring of ω-Languages*. In: *Proc. of CAV 2005, the 17th International Conference on Computer Aided Verification, Edinburgh, Scotland, UK, July 6-10, 2005*, *Lecture Notes in Computer Science* 3576, Springer, pp. 364–378, doi:10.1007/11513988_36.

[18] M. B. Dwyer, G. S. Avrunin & J. C. Corbett (1999): *Patterns in property specifications for finite-state verification*. In: *Proc. of ICSE '99, the 21st international conference on Software engineering, Los Angeles, California, USA*, ACM, pp. 411–420, doi:10.1145/302405.302672.

[19] K. Etessami & G. J. Holzmann (2000): *Optimizing Büchi Automata*. In: *Proc. of CONCUR 2000 - Concurrency Theory, the 11th International Conference University Park, PA, USA, August 2225, 2000*, *Lecture Notes in Computer Science* 1877, Springer, pp. 153–168, doi:10.1007/3-540-44618-4_13.

[20] P Gastin & D. Oddoux (2003): *LTL with Past and Two-Way Very-Weak Alternating Automata*. In: *Proc. of MFCS 2003, the 28th International Symposium in Mathematical Foundations of Computer Science,Bratislava, Slovakia, August 25-29, 2003*, *Lecture Notes in Computer Science* 2747, Springer, pp. 439–448, doi:10.1007/978-3-540-45138-9_38.

[21] M. Geilen (2001): *On the Construction of Monitors for Temporal Logic Properties*. *Electr. Notes Theor. Comput. Sci.* 55(2), pp. 181–199, doi:10.1016/S1571-0661(04)00252-X.

[22] A. Genon, T. Massart & C. Meuter (2006): *Monitoring Distributed Controllers: When an efficient LTL algorithm on Sequences Is Needed to Model-Check Traces*. In: *Proc. of FM 2006: Formal Methods, the 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006*, *Lecture Notes in Computer Science* 4085, pp. 557–572, doi:10.1007/11813040_37.

[23] M. Gunzert & A. Naegele (1999): *Component-based development and verification of safety critical software for a brake-by-wire system with synchronous software components*. PDSE '99, pp. 134–145, doi:10.1109/PDSE.1999.779745.

[24] K. Havelund & G. Rosu (2002): *Runtime Verification, RV 2002: Preface*. *Electr. Notes Theor. Comput. Sci.* 70(4), pp. 201–202, doi:10.1016/S1571-0661(05)80585-7.

[25] K. Kalajdzic, E. Bartocci, S. A. Smolka, Scott Stoller & G. Grosu (2013): *Runtime Verification with Particle Filtering*. In: *Proc. of RV 2013, the fourth International Conference on Runtime Verification, INRIA Rennes, France, 24-27 September, 2013*, Lecture Notes in Computer Science, Springer, p. To Appear.

[26] H. Kopetz, M. Holzmann & W. Elmenreich (2001): *A universal smart transducer interface: TTP/A*. *Comput. Syst. Sci. Eng.* 16(2), pp. 71–77, doi:10.1109/ISORC.2000.839507.

[27] L. Lamport (1978): *Time, Clocks and the Ordering of Events in a Distributed System*. *Communications of the ACM* 21(7), pp. 558–565, doi:10.1145/359545.359563.

[28] O. Lichtenstein, A. Pnueli & L. Zuck (1985): *The glory of the past*. *Lecture Notes in Computer Science* 193, pp. 196–218, doi:10.1007/3-540-15648-8_16.

[29] M. Lukasiewycz, M. Glaß, J. Teich & P. Milbredt (2009): *FlexRay schedule optimization of the static segment*. In: *Proc. of the CODES+ISSS '09, the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, Grenoble, France*, ACM, pp. 363–372, doi:10.1145/1629435.1629485.

[30] N. Markey (2003): *Temporal logic with past is exponentially more succinct*. *EATCS Bulletin* 79, pp. 122–128.

[31] S.P. Miller, M.W. Whalen & D.D. Cofer (2010): *Software model checking takes off*. Communications of the ACM 53(2), pp. 58–64, doi:10.1145/1646353.1646372.

[32] A. Pnueli (1977): *The temporal logic of programs*. Proc. 18th IEEE Symposium on Foundations of Computer Science, pp. 46–57, doi:10.1109/SFCS.1977.32.

[33] T. Pop, P. Pop, P. Eles, Z. Peng & A. Andrei (2008): *Timing analysis of the FlexRay communication protocol*. Real-Time Systems 39(1–3), pp. 205–235, doi:10.1109/ECRTS.2006.31.

[34] J.P. Queille & J. Sifakis (1982): *Specification and verification of concurrent systems in CESAR*. In: Proc. of the 5th Colloquium on International Symposium on Programming, Springer-Verlag, pp. 337–351, doi:10.1007/3-540-11494-7_22.

[35] G. Rosu & K. Havelund (2005): *Rewriting-Based Techniques for Runtime Verification*. Automated Software Engineering 12(2), pp. 151–197, doi:10.1007/s10515-005-6205-y.

[36] K. Sen, A. Vardhan, G. Agha & G. Rosu (2006): *Decentralized runtime analysis of multithreaded applications*. doi:10.1109/IPDPS.2006.1639591.

[37] K. Sen, A. Vardhan, G. Agha & G. Rou (2004): *Efficient decentralized monitoring of safety in distributed systems*. 26, pp. 418–427, doi:10.1109/ICSE.2004.1317464.

[38] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka & E. Zadok (2011): *Runtime Verification with State Estimation*. In: Proc. of RV 2011, the Second international conference on Runtime verification, San Francisco, CA, USA, Lecture Notes in Computer Science 7186, Springer-Verlag, pp. 193–207, doi:10.1007/978-3-642-29860-8_15.

[39] S. Tripakis (2005): *Decentralized observation problems*. pp. 6–11, doi:10.1109/CDC.2005.1582122.

[40] Y. Wang, T. Yoo & S. Lafortune (2007): *Diagnosis of Discrete Event Systems Using Decentralized Architectures*. Discrete Event Dynamic Systems 17(2), pp. 233–263, doi:10.1007/s10626-006-0006-8.

[41] Y. Wang, T.-S. Yoo & S. Lafortune (2007): *Diagnosis of Discrete Event Systems Using Decentralized Architectures*. Discrete Event Dynamic Systems: Theory and Applications 17(2), pp. 233–263, doi:10.1007/s10626-006-0006-8.

[42] P. Wolper (2001): *Constructing Automata from Temporal Logic Formulas: A Tutorial*. In: Lectures on formal methods and performance analysis, Lecture Notes in Computer Science 2090, Springer-Verlag New York, Inc., pp. 261–277, doi:10.1007/3-540-44667-2_7.