Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Automatic Generation of Runtime Monitors from Structured Natural Language Using Timed Automatons

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von:	Tim Jellinek geb. Sikatzki
geboren am:	27.05.1997
geboren in:	Werdohl
Gutachter/innen:	Prof. Lars Grunske
	Prof. Genaína Rodrigues
eingereicht am:	.... verteidigt am: ....

Contents

1 Introduction

Safety-critical systems, such as aircrafts and trains are systems whose failure can result in loss of life, significant property damage, or damage to the environment[21]. Unwanted behaviour, if not detected early, in the worst case can cause system failure. This makes safety-critical systems a perfect fit for runtime verification. The approach is used to verify the absence of specified unwanted behaviour in a system.[20, 30]. Runtime verification already established itself in the context of cyber-physical system[46, 27]. Though being an effective solution for behaviour monitoring, creating runtime verification monitors, which verify the intended properties of the system specification, can be a complex task. Most of the time, the user has to provide the system specification in the specification language that is used to setup their runtime monitors. This specification language can be based on regular expressions[36], rules[4], streams[24, 32, 11], LTL formulas and finite state machines[39]. Another design decision which needs to be taken is whether the system is monitored from the inside or outside. Both of these approaches have their own advantages and disadvantages. While having a built-in monitor inside your system can have advantages for the data collection, because system processes and monitoring processes can act on a schedule and synchronized clocks, we have to instrumentalize the system to work with the monitor. If not done correctly, this can lead to an altered behaviour of the monitored system, especially for cyber-physical systems, where computation overhead can lead to unwanted latencies[44]. Therefore, if a build-in monitor is planned, the system has to be designed with the additional resources, the monitor occupies, in mind. The alternative to build-in monitors are external monitors that do not have direct access to internal system values, but rely on communication between the system and the monitor to receive important system events. The advantage of those monitoring approaches is that they can be designed non-intrusive if the natural system communication allows for it. The system behaviour can then be observed without any interaction with the monitor. This approach leads to different challenges, because timing and communication in asynchronous systems are non-trivial tasks and can also require system instrumentation if it does not provide natural logging or communication output. In this thesis, we tackle these challenges to present an approach which provides access to runtime verification with low time and resource investment. The idea is to automatically generate ready-to-use runtime verification monitors with minimal effort and setup. We utilize the mapping from structured natural language to Dwyer's specification patterns[10] provided by Autili et al.[3] for specification accessibility and compatibility. Additionally, we use PSP-UPPAAL[41], a tool which provides a catalogue of UPPAAL observer for model checking, based on these specification patterns. In Section 2 we present the background of our thesis and explain concepts and approaches which will be relevant throughout the thesis. Then we talk about our system concept, the architecture and design decisions in Section 3. Afterwards, in Section 4 we propose our experiments and evaluation. Our runtime verification generation is evaluated on a variation of the self-adaptive BSN[13] which is implemented in Arduino[1]. We describe our experimentation setup and present research questions which we address throughout the evaluation. In Section 5 we then talk about challenges which occurred during the evaluation, before going over threats to the validity of our approach in Section 6. Finally, we present similar approaches and related ideas for runtime verification and monitor generation combined with their advantages and disadvantages in Section 7. Here we will also explain the differences in comparison to our approach and set our system into context, before then concluding the thesis and talking about future research that can be done to improve our system in Section 8.

[1]`https://github.com/carwehlm/masterthesis/tree/master/code/BSN`

2 Background

In this section we explain the basic concepts that are used within this thesis and talk about preceding research that our approach relies on.

2.1 Cyber-Physical Systems

Cyber-physical system (CPS) is a term for systems which use software and hardware components to interact with physical objects in the real world[17]. Nowadays, CPS can be found in different domains, such as transportation[26] and healthcare sectors[9]. Implementing cyber physical systems leads to multiple design challenges as Hu et al.[17] explains, two of them being:

Reliability and uncertainty Many CPS are inherently safety-critical systems[35], because a system failure can have direct impact on the physical world. Due to their interaction with an unpredictable environment cyber-physical systems have to take additional uncertainties into account. Common uncertainties are missing knowledge about the timing of inputs or about the state of the system[45].

Superdense Timing Superdense time revolves around the problem that "[...] time is continuous in the real world but must become discrete in the cyber world."[17, p.11]. Superdense time models are used to combine discrete untimed sequences and order them accordingly[17]. There are multiple different approaches that were proposed by different authors[28].

2.2 Runtime Verification

To counteract the effects of the uncertainties in CPS as described in Section 2.1, runtime verification is commonly used to achieve this goal. Runtime verification is an analysis approach which utilizes formal methods to verify the behaviour of a system during runtime. The approach itself can be split into two categories

- monitoring the system and its parameters
- analysis of monitored parameters with respect to specified properties

Monitoring There are multiple monitoring approaches that are used in runtime monitoring [35, 29, 39]. A runtime monitor is an executable unit which runs alongside the system. The goal of a runtime monitor is to observe the runtime behaviour of the monitored system and verify this behaviour against a specified property. Considering the design of a runtime monitor, multiple design decisions have to be taken into account. Monitors should have minimal impact on the behaviour of their system[5]. Bartocci et al.[5] give a good overview about different monitoring approaches which we will cover here briefly.

Offline and online monitoring Offline monitoring is mostly known as logging, where the verification of the specified property is done after the execution of the system. In this approach, important system events do not have to be evaluated during runtime, but are stored when they occur, which reduces the load on the running system and makes the approach less intrusive compared to online monitoring approaches. One of the disadvantages of offline monitoring is the delay in verification. Faulty behaviour can only be detected after the execution of the system, which makes it impossible to react to failures before the system finishes its execution. Online monitoring evaluates the properties during system execution. This makes it possible to implement error handling routines or manually adapt to faulty behaviour if it is detected. We divide online monitoring into asynchronous and synchronous online monitoring[5].

Asynchronous and synchronous online monitoring In online monitoring, the relation between the system and the monitor is relevant. In synchronous online monitoring systems, the monitor and the monitored system are dependant on each other. In these kind of systems, each component acts "in lock-step: every time the system generates an event, it waits for the monitor to process it before proceeding with its execution"[5, p.15]. One of the advantages of synchronous systems is that the monitor does not have to keep track of new incoming events while still evaluating the current one. The opposite of this is asynchronous online monitoring, where the monitor and system act independent from one another. The advantage of this approach is the less intrusive nature, where depending on the implementation, the system does not need to interact with the monitor in any way.

Analysis of monitored parameters After monitoring important system events, the analysis is also done by the monitor. For this, a specification language is used to define the property which is evaluated. Many specification languages are based on temporal logic to describe the intended behaviour of the system. This specification can then be used to generate executable monitors.

2.3 Temporal Logic

Temporal logic can be used to specify properties of system behaviours over time[37]. The branch of logic adds temporal operators to the propositional logic. Because of its ability to verify statements about processes and orders of event occurrences, it can be used in model checking as well as runtime verification[37]. Two of the most popular temporal logics are linear temporal logic (LTL) and computation tree logic (CTL)[14]. While linear temporal logic is used to specify properties over a single computation path, CTL can express properties over a tree of all possible computation paths.

LTL Linear temporal logic is able to process and verify single paths. These paths, for example, can be finite execution traces of a system, which were aggregated during its execution, but also infinite traces in a system model. Huth et al.[18, p.175] propose the following definition for the LTL syntax:

$$\begin{aligned}\phi ::= &\top|\bot|p|(\neg\phi)|(\phi \vee \phi)|(\phi \wedge \phi)|(\phi \rightarrow \phi)\\ &|(X\phi)|(F\phi)|(G\phi)|(\phi U\phi)|(\phi W\phi)|(\phi R\phi)\end{aligned}$$

In this context, ϕ is a LTL formula and p is an atomic proposition. Compared to propositional logic, LTL uses multiple temporal operators that perform on single paths, such as *finally*(**F**), *globally*(**G**) and *until*(**U**). With these operators we can express properties, for instance:

- $F\phi$ (ϕ eventually holds)
- $G\phi$ (ϕ always holds)
- $\phi U\omega$ (ϕ holds until ω holds)

Operators can also be nested to create more expressive properties. For instance, the property ϕ *repeatedly holds* can be expressed as $GF\phi$.

CTL The computation tree logic does not operate over a single path but rather over all possible paths. As the name suggests these possible paths can be expressed in a tree structure, therefore we talk about computation trees. In addition to the operators described in LTL, CTL also adds the quantifying operators **A** and **E** to the specification. **A** specifies that a property should hold in all possible paths, while **E** specifies that there exists at least one computation path where the property holds. For more information about the basics of CTL see [1]. CTL and LTL share an intersection of properties which can be expressed by both specification languages.

2.4 Specification Patterns

Dwyer's specification patterns[10] summarize a number of common system behaviour properties, which can adjusted and used for the model verification of a system. Every pattern provides formulas for every scope. The scope of a specification pattern describes context where the pattern is expected. The patterns are specified in either *LTL* or *CTL*. These patterns were extended by timed and untimed versions using Metric Temporal Logic (*MTL*) and Timed CTL (*TCTL*) formulas[22] and by probabilistic versions in Probabilistic LTL (*PLTL*) and Continuous Stochastic Logic (*CSL*)[15]. Autili et al.[3] aligned the basic patterns and their timed and probabilistic additions with structured natural language. The authors also introduced new patterns on top of Dwyer's specification patterns[2].

[2]An overview of all current patterns with their timed and probabilistic versions can be found here `http://ps-patterns.wikidot.com/`

2.5 UPPAAL

UPPAAL[42] is a tool which utilizes timed automatons to provide modelling as well as model checking features. A UPPAAL model can consist of multiple timed automatons. Each automaton is a structure of edges (transitions) and vertices (locations).

Locations Locations in UPPAAL are the vertices in automatons. Each location can possess an invariant and a label. The invariants are constraints which must be satisfied when the automaton wants to enter the location. The label can be used to identify a location and to verify specific queries of the system. Every automaton contains exactly one initial location which is the location, the automaton is in at the start of the system. Another feature UPPAAL offers are committed locations. Whenever an automaton enters a committed location, the rest of the system is locked until the committed location is left. No other transitions in other automatons can be fired during this time.

Transitions Transitions are the edges in UPPAAL automatons. A transition can possess guards, synchronisations and assignments. A guard is a constraint that has to be satisfied before the transition can be fired. Synchronizes are used to synchronize multiple edges using channels. Transitions can either be emitting or receiving. This is denoted via `!` (emitting) and `?` (receiving)[3]. All receiving transitions have to synchronize on emission. Assignments are variable assignments which will be executed when a transition is fired.

Specification of the system is done using timed computation tree logic (TCTL). TCTL[23] adds timed intervals to the logical operators used by CTL.

2.6 PSP-UPPAAL

In order to make model checking via temporal logic more accessible, the authors of PSP-UPPAAL[41, 43] utilize the TCTL formulas of the specification patterns we explained in Section 2.4 to provide ready-to-use specifications for systems that were designed as UPPAAL models. The authors built UPPAAL observer templates for each of the specification patterns if applicable. These UPPAAL observer can be integrated into the system to handle the verification of specification pattern properties.

2.7 Body Sensor Network

The system under test for this thesis will be the Body Sensor Network (BSN). The SA-BSN [13] is a self-adaptive version of the body sensor network. Self-adaptive systems have the advantage that they can adjust themselves during runtime to counteract uncertainties which can occur in cyber-physical systems as described in Section 2.1. The system monitors the patient's health status, while adapting itself to possible uncertainties. Figure 1 depicts an overview of the architecture of the BSN. The system

[3]`https://www.seas.upenn.edu/~lee/09cis480/lec-part-4-uppaal-input.pdf`

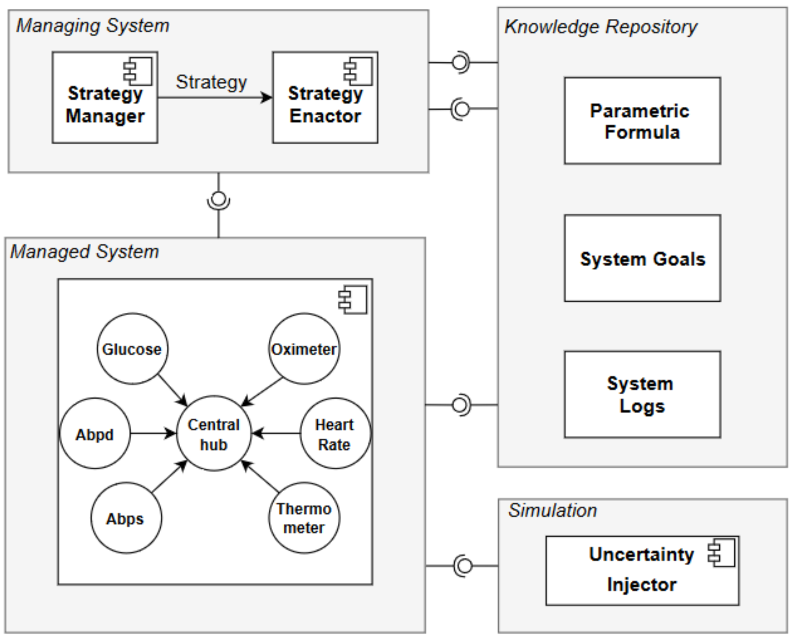


Figure 1: Architecture of the self-adaptive Body Sensor Network. Taken from Gil et al.[13]

itself consists of multiple sensor nodes, each measuring separate vital parameters, such as the heart rate and body temperature. Logs and system goals which should be reached during the execution are stored in the knowledge repository component. These are used by the strategy manager to generate an appropriate strategy, the strategy enactor can then apply to the managed system. For the thesis, we use the Arduino implementation of the BSN provided by Marc Carwehl in his master thesis[4]. The components are connected via wires. Each sensor node as well as the central hub and the adapting unit are programmed on independent Arduinos. The Arduino BSN provides two sensors, one for the body temperature, another one for the pulse. This implementation utilizes simulated data and does not operate with real patients.

[4]`https://github.com/carwehlm/masterthesis/tree/master/code/BSN`

3 System

In this chapter we elaborate how our system is designed. Our system consists of two components. The monitor, which is partly generated but also makes use of custom C++ classes we implemented to provide state machine support and the monitor generator. First, we talk about the concept of our runtime monitors in Section 3.4, where we go over the creation of our state machine in C++. Then, we explain how the generation of monitors works in Section 3.6. As we provide a monitor generation tool, we will refer to our system as *Monerator* throughout this thesis. The implementation of our approach can be accessed at our github repository[5] and can be freely used for future research.

3.1 Concept

The concept of our system can be seen in Figure 2. The property configuration describes user input. If the user has a specific property in mind, they have to look through the specification patterns and decide which one seems fitting for their use case and then provide this information by creating a property file. We explain the structure of our generated monitors in detail in Section 3.4. In Section 3.6 we talk about the structure of our generation tool and the design decisions we made.

3.2 Assumptions and Limitations

Before we explain the details of our system we need to talk about some assumptions we make for our monitor generation. Our monitors are designed to be generated out of the observers that are provided by PSP-UPPAAL[33]. We do not intend to capture all UPPAAL features in our generator, only the ones that are used by the observers.
Furthermore we assume, that transitions are taken as soon as possible in the monitor. We make this assumption for two reasons. One, in comparison to UPPAAL, our monitor does not have to wait for another system component to finish before it can take a transition therefore it can always take the transition as long as the relevant guards and invariants are satisfied. Second, it makes the implementation simpler.
Additionally we also assume that the observer always contains a location with the `ERROR` label. This is, because we are not able to check the formulas in the queries of the UPPAAL observer. Therefore we later show how to add an `ERROR` location to an observer so that it's formula becomes `A[] not Observer.ERROR`, which means that the automaton should not hold in the `ERROR` location.
Another limitation that we need to mention is that we can only verify safety properties and timed liveness properties. During the runtime verification we can only analyse finite execution traces. We can not make statements about future events that may occur. Therefore we can verify that an event occurs during the execution or that an event occurred after a set time interval. We can not verify that an event eventually

[5]`https://github.com/timsik/Monerator`

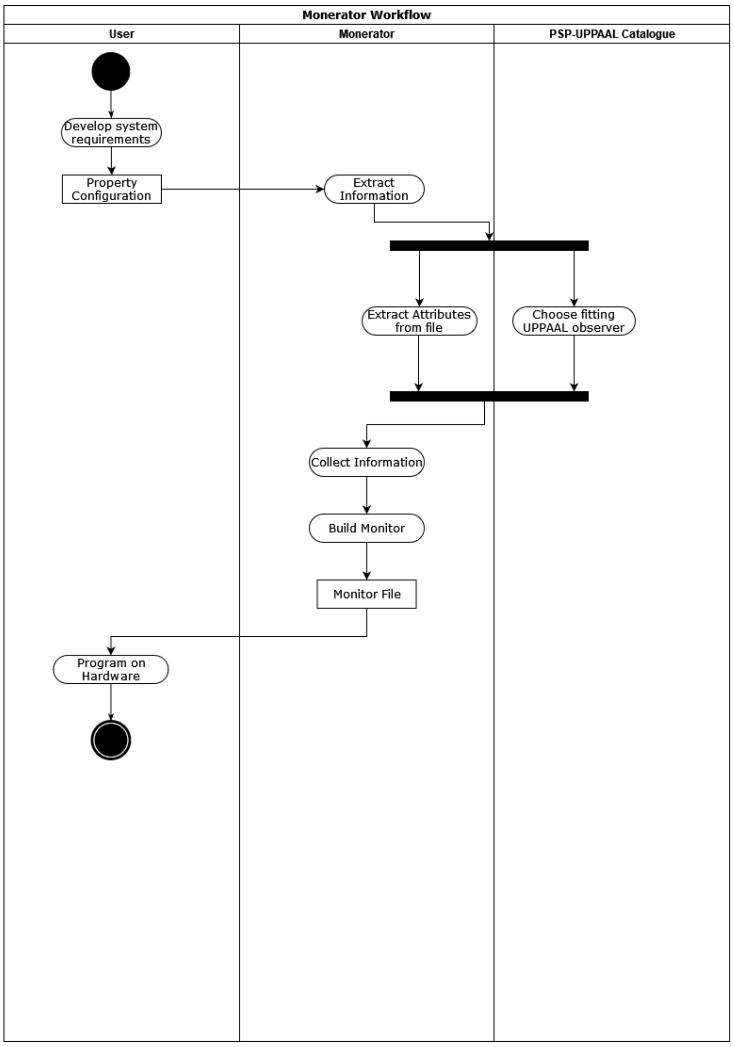


Figure 2: Overview of our presented workflow. The *User* and *PSP-UPPAAL Catalogue* lanes are background work, while the *Monerator* lane depicts the contribution of the thesis.

occurs without any time constraints, because we work with finite execution traces. Liveness properties can only be verified by analysing execution traces with possibly infinite length. Note that most of the runtime approaches have to deal with this limitation and it is not exclusive to our system.

3.3 Arduino Structure

We use Arduino as our hardware solution and therefore have to adapt our monitor architecture to the Arduino syntax. An Arduino sketch, which is the term for Arduino programs, consists of four components. These will be referenced as `header`, `globals`, `setup` and `loop` components. The `header` component covers the libraries, which are utilized in the sketch. The `globals` component covers all global variable and function

declarations, which are required for the `setup` and `loop` components. Every Arduino sketch initially contains an empty `setup()` and `loop()` function. The `setup` component is specified inside the `setup()` function. The `setup()` function is called once at the start of an Arduino and is used to initialize global variables with specific values and execute statements which should be only executed once. The `loop` component is specified inside the `loop()` function. While an Arduino is running and after executing the `setup()`, it will start to repeatedly call the `loop()` function.
This structure influences the conception of our monitor architecture as well as the design decisions we made for our monitor generator.

3.4 Monitor Architecture

Our runtime monitors are expressed as a set of states which represent the UPPAAL locations and a set of transitions. These sets make use of a set of automatically generated helper functions that represent the guards, invariants and assignments of our model.

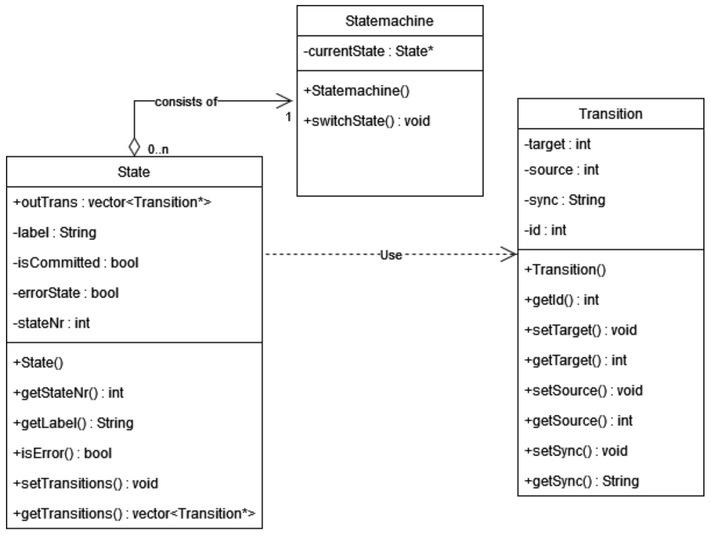


Figure 3: Class diagram of the monitor

Figure 3 shows a class diagram that depicts the relation between the states, transitions and the state machine. We go over each of the elements in the following paragraphs.

3.4.1 States

States represent the locations of the UPPAAL observer model. Each state consists of a state number, a label, a set of its outgoing transitions as well as the information if it is a committed or error state.
The state number is their unique identifier that is provided by the UPPAAL model. Each of the states is added to the `states` array by using its state number as index. That way the objects can be referenced from the monitor by calling `states[stateNr]`.

The label is used to determine the error state of the state machine. Aside from that it mostly exists to provide the same functionality of the original UPPAAL model.
The outgoing transitions as well as the information about the error and committed attributes are used to traverse the state machine during runtime.
One thing we miss in our `state` class are the invariants. For the invariants, we chose to directly generate them in form of functions. We then store those functions in an array, where the index resembles the state number of the state which contains the invariant in the UPPAAL model. We chose this approach because finding a way to encapsulate the conditional strings we extracted from the UPPAAL model in an attribute, which can be evaluated, seemed like a greater challenge. Since our monitors have the requirement to be automatically generated and executable without human intervention, the focus of our generated code relies on functionality. This is why we chose to handle invariants in the way we explained earlier.

3.4.2 Transitions

Transitions in our state machine represent the transitions in the UPPAAL model. Each transition in our state machine contains a number of attributes. These are the target, the source, the synchronisation and the id. We will go over those one by one.

The target and source attributes are equivalent the state numbers of the target and source state of the transition. Those attributes are mainly used for state machine traversal. The synchronisation attribute corresponds to a string that is either empty or resembles one of the events, the monitor receives from the system. A transition with a synchronisation has to be triggered if the corresponding event is detected. Similar to the invariants of states, we created function arrays for the guards and assignments of each transition. If we want to access these functions, we need a unique identifier for our transitions. Such an identifier is not provided in our UPPAAL observers and therefore we add an `id` attribute to every transition on creation.

3.4.3 State machine

The state machine class keeps track of the current position in the state machine, as well as managing its initial state. Additionally it provides the functionality to switch between states. This reduces the amount of redundant code that has to be generated.

3.4.4 Event Handling

For event handling we use the *SoftwareSerial*[40] library in our Arduino context to receive events of our system under test. The monitor expects to receive either `P_reached?` or `P_left?` events, where `P` is a placeholder for the variables which are used in the specification pattern of the monitor. We provide a map that holds Key-Value-Pairs of associated variables. Each time we receive an event, the associated variables for this event are changed accordingly. An example event map is shown in Table 1.

Key	Value
P_holds	0
P_held_once	0
S_holds	0
S_held_once	0

Table 1: Event map for two events P and S. Respective signals are `P_reached?/P_left?` and `S_reached?/S_left?`

Each time we receive an event it is pushed to the back of a double-ended queue (deque) which we use to keep track of the occurring events. If the monitor is ready to process the next event, it is then popped out of the front of the queue.

3.5 Monitor logic

The state machine itself functions in multiple steps. First, the state machine is build. All states and transitions will be initialized and the event handler and clock will be set up. Right afterwards the monitoring process is started. We will go over those steps in detail in the following paragraphs.

3.5.1 Build

The building stage handles the initialization of our state machine. For Arduino, most of this happens in the `setup()` function of the monitor file. In this function, we initialize the transitions and states, with all the information they need. Each state and transition is created with its specific attributes as explained in Section 3.4. During this step, we also add the states to the `states` array and the transitions to the list of outgoing transitions of their respective source states. We also populate our guard, invariant and assignment arrays with their functions. The event handling is also prepared during this stage. We start the event handler so it is able to process incoming events. Additionally we also fill the event map with Key-Value-Pairs which are expected to be set during runtime.

3.5.2 Monitoring Process

Our monitoring process is realized as a loop which keeps track of multiple things at the same time. In our context, the `loop()` function of our Arduino file is a perfect fit for the processing.
One important thing to keep track of is the time. We implement a clock which uses the `millis()` function of Arduino to keep track of the time the monitor is running, while giving us the option to reset the clock itself. The `millis()` function returns the time in milliseconds that the Arduino is running. After about 50 days, the time overflows[6], the function resets to zero and potentially breaks the monitor. Right now,

[6]`https://www.arduino.cc/reference/de/language/functions/time/millis/`

we implemented no solution for this issue. Resetting the monitor once every 50 days serves as a work around though. Algorithm 1 shows the loop running in our monitor executable. Line 1 covers our event handling as described in Section 3.4.4. Here we fill our event double edged queue and update our event map entries. Lines 2 and 3 update the clock, set the $*$_*holds* variables to the value of the respective key in the event map and update the placeholder variables. The placeholder variables are copies of the variables that are used in assignments. In UPPAAL the guards and invariants are validated after making the assignment, so we need to have a way to revert those assignments by saving the initial values in placeholders. The next line checks if we currently are in an error state. If so, the monitor returns an error message every second until the monitor gets reset manually.

Algorithm 1: Monitoring Loop

```
   inputs  : observer.transitions
   outputs : Transitions for the Monitor
   // running through the loop
1  HandleIncomingSignals();
2  SetClock();
3  SetVariablesAndPlaceholder();
4  CheckForErrorState();
5  foreach $transition \in currentState.outgoingTransitions$ do
6      ExecuteAssignment();
7      if transition can be fired then
8          prioritizeSyncTransition();
9          switchState(transition.target);
10     end
11     else
12         resetAssignments();
13     end
14 end
```

3.6 Generator

In this section we explain the generation part of our system. Figure 4 shows an example of how the resulting state machine of our generation looks compared to the observer we specified as input. Note that we use the label of the initial state to specify the invariant. This is just used for illustration. The invariants are not stored at the same place as the labels.

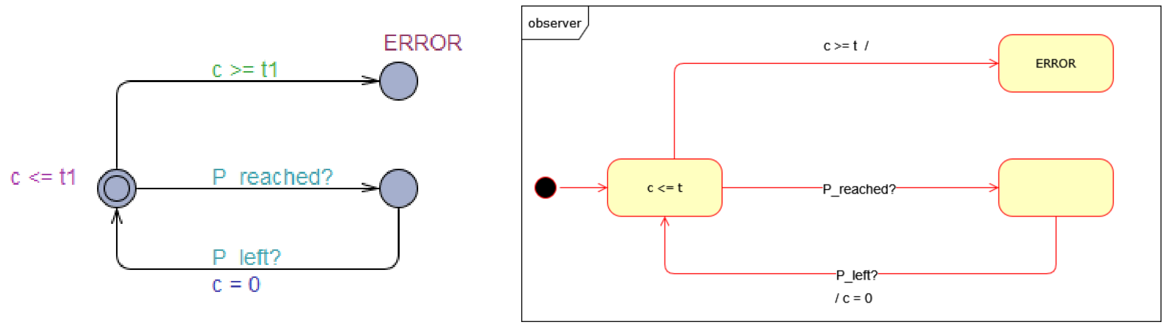


Figure 4: Timed Recurrence Observer with the Globally Scope on the left and its generated state machine on the right

Our generator works with a property file as input. Figure 5 shows an example property file for the `RecurrenceGloballyTimed` Observer

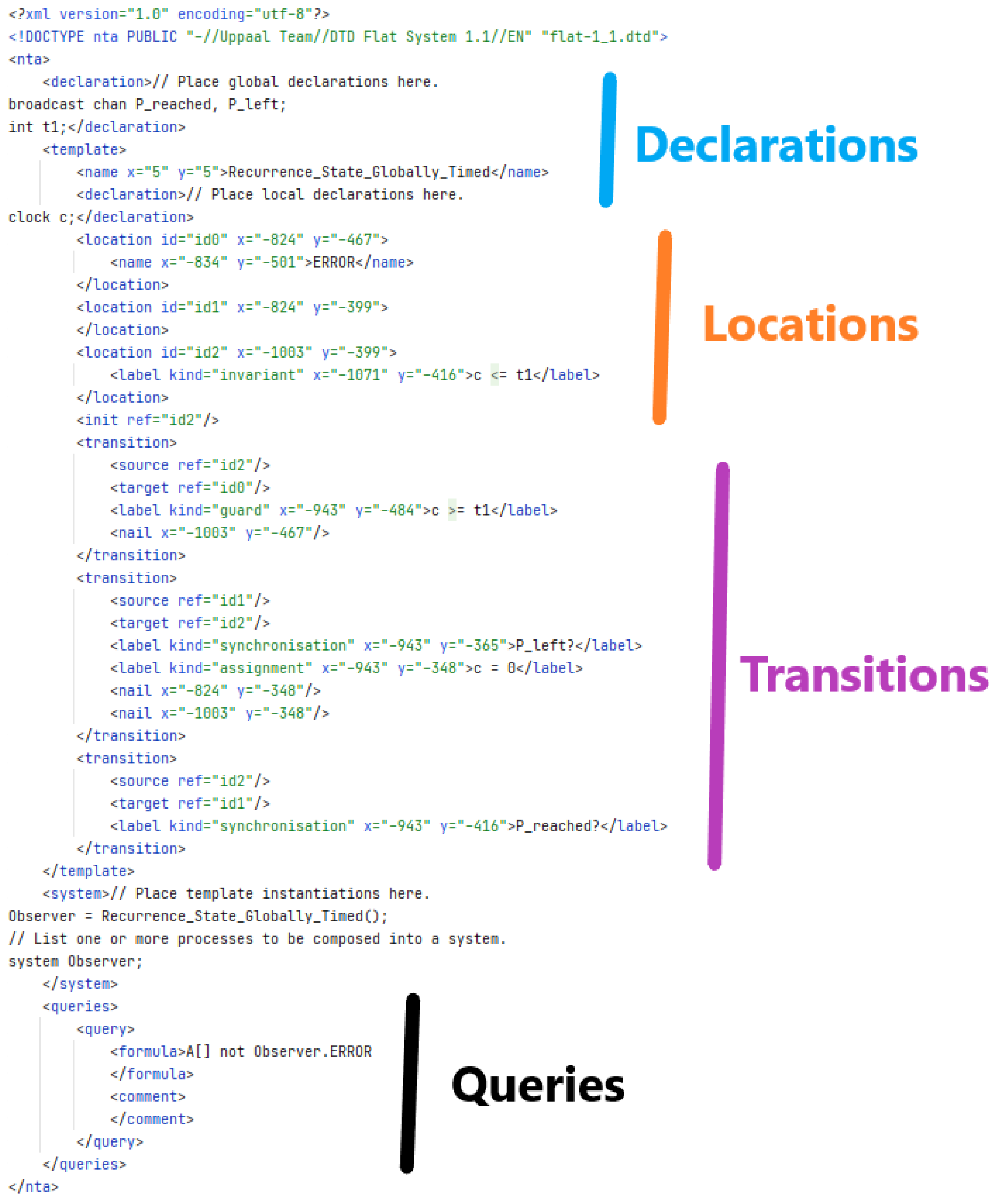
```
pattern=Recurrence
timed=true
scope=Globally
P=sensor sends message
t1=2200
```

Figure 5: Property file for the Recurrence Globally Timed Observer

This file provides information about the specification pattern, which is needed. Our generator is split into three classes we created in the context of this thesis. We chose this split to separate the data and display from our program logic. We describe each class in detail in the later sections. The `TemplateHandler` class parses the data provided by the property file to be used by the generator. The `FileHandler` manages the structure of the generated file. It provides the functionality to specify the generation path as well as the monitor name. It also handles the content of the monitor. The largest part of the generator is the `Monerator` class which manages the other classes, inherits the generation logic and also contains the `main` function where our monitor is finally generated.

3.6.1 Preservation of Information

When generating a monitor from the original UPPAAL observer we want to preserve all of the information that is necessary for the functionality of original observer. This property comes in handy when we later talk about the correctness of our monitor in Section 4. An UPPAAL automaton mostly consists of four components. Those components are the declarations, the locations, the transitions and the queries. Inside the declarations, the variables that will be used by the observer, such as the clock and the channels or constraining variables, are declared. The locations component defines the locations of the automaton, each with their respective invariant, label and location `id`. In the transitions component, we define the transitions of the automaton with their respective assignments, guards and synchronizations. The query component contains the formulas that are checked by UPPAAL during the model verification. We already talked about our limitations with those formulas in Section 3.2 and because of the fact that we cannot verify formulas other than `A[] not ERROR`, we do not include the formula in our observer. Figure 6 shows the XML content of the `RecurrenceGloballyTimed` observer, where the four components are annotated. We use a forked version of JUppaal[25] to transform this XML file into Java objects which we then utilize to generate our monitors. In the following, we provide pseudocodes of the process for the monitor generation. The declarations component is generated in two steps. All declaration variables which need a value, timing constraints for example, are generated from the property file, because they require user input. The observer specific values, the clock for example, are always generated and initialized with zero. An exception to that is the `nxtCmt` flag which is initialized with one if the initial state is also marked as committed.



```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC "-//Uppaal Team//DTD Flat System 1.1//EN" "flat-1_1.dtd">
<nta>
    <declaration>// Place global declarations here.
broadcast chan P_reached, P_left;
int t1;</declaration>
    <template>
        <name x="5" y="5">Recurrence_State_Globally_Timed</name>
        <declaration>// Place local declarations here.
clock c;</declaration>
        <location id="id0" x="-824" y="-467">
            <name x="-834" y="-501">ERROR</name>
        </location>
        <location id="id1" x="-824" y="-399">
        </location>
        <location id="id2" x="-1003" y="-399">
            <label kind="invariant" x="-1071" y="-416">c <= t1</label>
        </location>
        <init ref="id2"/>
        <transition>
            <source ref="id2"/>
            <target ref="id0"/>
            <label kind="guard" x="-943" y="-484">c >= t1</label>
            <nail x="-1003" y="-467"/>
        </transition>
        <transition>
            <source ref="id1"/>
            <target ref="id2"/>
            <label kind="synchronisation" x="-943" y="-365">P_left?</label>
            <label kind="assignment" x="-943" y="-348">c = 0</label>
            <nail x="-824" y="-348"/>
            <nail x="-1003" y="-348"/>
        </transition>
        <transition>
            <source ref="id2"/>
            <target ref="id1"/>
            <label kind="synchronisation" x="-943" y="-416">P_reached?</label>
        </transition>
    </template>
    <system>// Place template instantiations here.
Observer = Recurrence_State_Globally_Timed();
// List one or more processes to be composed into a system.
system Observer;
    </system>
    <queries>
        <query>
            <formula>A[] not Observer.ERROR
            </formula>
            <comment>
            </comment>
        </query>
    </queries>
</nta>
```

Figure 6: Annotated UPPAAL Observer XML

Algorithm 2: Generating Transitions

```
  inputs  : observer.transitions
  outputs : Transitions for the Monitor
  // generate transitions
1 foreach tra ∈ observer.transitions do
2     init new transition;
3     generate id for transition;
4     set transition source to tra.source set transition target to tra.target
        HandleGuard(tra.guard);
5     HandleAssignment(tra.assignment);
6     add tra to outgoing transitions of its source state
7 end
```

The transition generation shown in Algorithm 2 follows a similar procedure compared to the declarations. We iterate over every possible transition of the observer and add the information to our monitor by utilizing the classes we already mentioned in Section 3.4. The transitions are special in a way that we add information to the system. Each transition receives a unique id, which is not provided by the observer. We use this to call the guards and assignments of each transition. We also utilize the information of the source state to add the transition to its `outTrans` list.

Algorithm 3: Generating States

```
  inputs  : observer.locations
  outputs : States for the Monitor
  // generate states
1 foreach loc ∈ observer.locations do
2     init new state;
3     set state id and label to loc.id and loc.label;
4     if loc is initial location then
5         set init flag in state
6     if loc is committed then
7         set committed flag in state
      // assign loc invariant to state
8     HandleInvariants(loc.invariant);
9 end
```

The state generation described in Algorithm 3 needs to keep track of the attributes, if the location is committed for example, provided by each location which mostly can be depicted as boolean flags. Additionally we have to handle its invariant similar to how we handle the guards of our transitions. Additionally to that, we initialize a `statemachine` object with the initial state as parameter.

3.6.2 Template Handler

The `TemplateHandler` extracts data from the property file which specifies the pattern used and for the monitor generation as well as the value of the time constraints that can occur in timed specification patterns. The `TemplateHandler` is initialized with its attribute `pathToTemplate`, which is the path of the property file. It uses the property information to determine which UPPAAL observer is needed and provides the name of the observer file to the generator. The time constraints and events specified by the property file will also be provided to the generator.

3.6.3 File Handler

The `FileHandler` class generates the monitor file using the information provided by the generator. It serves as a class to adjust the structure of the monitors generated by the `Monerator` class. The class provides functions to generate code based on the attributes of our `Monerator` class. This can be utilized to adapt the monitor to other environments than Arduino. Unfortunately this only specifies the Arduino body (header, globals, build, setup) and not the content of the generated code. Therefore we are limited to the C++ code generated by the `Monerator` class but can alter the context in which the code is embedded.

3.6.4 Monerator

The `Monerator` handles the generation of the monitor by making use of the `TemplateHandler` and `FileHandler` classes. It also contains the `main` function for our system, where the other classes are initialized and configured. This class implements the algorithms we described in Section 3.6.1.

4 Evaluation

For the evaluation we present five research questions we try to answer in this section. The questions are located in Table 2.

RQ1	Are the generated monitors correct?
RQ2	How many properties can we express?
RQ3	How much memory is needed for the monitor?
RQ4	How long does the monitor generation take?
RQ5	How fast can we detect errors?

Table 2: Overview of our research questions

4.1 Correctness of Generated Monitors

In this section we elaborate on how we mitigate the risk of our generated monitors to be incorrect which will also answer our research question **RQ1** (Are the generated monitors correct?). Therefore we make the assumption that if we can show that there exists an isomorphism between the observer and the generated monitor then they have the same behaviour. We first explain, how we contain the necessary information to recreate a functional copy of the initial observer out of the generated monitor to show that there is no information loss in our generation. Then, we will show that an isomorphism between the observer and the generated monitors exists.

4.1.1 Problem with Information Conservation

We lose no information during the generation of our runtime monitors when using an untimed specification pattern, as we show in Section 3.6.1, but we have to alter some of the information to make up for our timing issues, when we take timed specification patterns into consideration. This makes a difference for systems that rely on these precise time constraints and therefore we can not guarantee that our monitors are correct. What we can do though is to show that the rest of the functionality of our monitor should still be the same by using an `untimed` specification pattern as example. Also note that the timing issues do only partially affect the correctness of our approach. With more research and optimization in the future and a possible swap to another more efficient hardware, we could get the grace interval close to 0ms (while never reaching exact timings, see Section 4.5). The faster we get through optimization the smaller the risk of possible deviating behaviour between our approach and the original UPPAAL observer gets.

4.1.2 Isomorphism

Isomorphism in graph theory is a property which can be utilized to show that multiple graphs share the same behaviour. Two graphs are isomorph if:

- there exists a bijection between both
- the structure is preserved

We use the definition of Hiseah et al.[16] for isomorphism of two labelled graphs $G(V, E, L_V, L_E, l)$ and $G'(V', E', L'_V, L'_E, l')$. We need to find a bijective function $f : V \rightarrow V'$ such that:[16]

$$\forall u \in V, l(u) = l'(f(u)) \tag{1}$$
$$\forall u, v \in V, (u, v) \in E \leftrightarrow (f(u), f(v)) \in E' \tag{2}$$
$$\forall (u, v) \in E, l(u, v) = l'(f(u), f(v)) \tag{3}$$

The authors of this definition use graphs which have vertices and edges with unique identifiers, because labels can sometimes be used for multiple vertices and edges. The labelling function l is then used to retrieve the label of an element using the unique id. For example, for a vertex with the label $ERROR$ and id 2 the labelling function would be $l(2) = ERROR$. Similar, for an edge with the label a, source id 3 and target id 5 the labelling function is $l(3, 5) = a$.
We want to use this definition for our UPPAAL models and state machines, therefore we define the labels of vertices L_V as tuple $(Label, Invariant, CommittedFlag, InitialFlag)$ and the labels of edges L_E as tuple $(Guards, Updates, Synchronizations)$. The labelling function l will therefore also be defined as $l(u) \in L_V$ and $l(u, v) \in L_E$ for u and v being ids of two different vertices in G. In Section 3.6.1 we explain how we preserve the information of our untimed UPPAAL observers during monitor generation. Because the states keep the identifier of their origin location and the number of vertices also stays the same, our bijective function is defined as $f(u) = u$ for u being the id of any vertex in G. Our labelling functions l and l' return the same values in both graphs for the same parameters. This is due to the fact, as we describe the generation process in Section 3.6.1, that all content of our labels in the untimed UPPAAL observer are copied to the generated monitor. We show the isomorphism between the UPPAAL observers and their generated counterpart by utilizing the properties of isomorphism (1)(2)(3).

$$(1)\ \forall u \in V, l(u) = l'(u) = l'(f(u))\ \checkmark$$

The second property states that each edge between two vertices u and v in G also exists between the vertices $f(u)$ and $f(v)$. Our generation algorithm also copies the edges of the UPPAAL observer using the identifiers of the target and source locations and new target and source states. Therefore, for each edge (u, v) in E (set of edges in G) there also exists a generated edge $(f(u), f(v))$ in E' (set of edges in G').

$$(2) \forall u, v \in V, (u, v) \in E \leftrightarrow (f(u), f(v)) \in E'\ \checkmark$$

The third property specifies that for every edge in E, the generated edge in E' as specified in property (2) inherits the same label as its UPPAAL counterpart. Showing this property applies to our monitors is trivial because we already showed the equality of labels for vertices in property (1). As for the labels of transitions, the generated transitions also copy the label of their origin, similar to the vertices as specified above and in the algorithms in Section 3.6.1.

4.2 Experimentation Setup

Our experiments are based on Arduino. Our system produces executable code that is compatible with Arduino and we also use an implementation of the BSN that was build with Arduino[6].

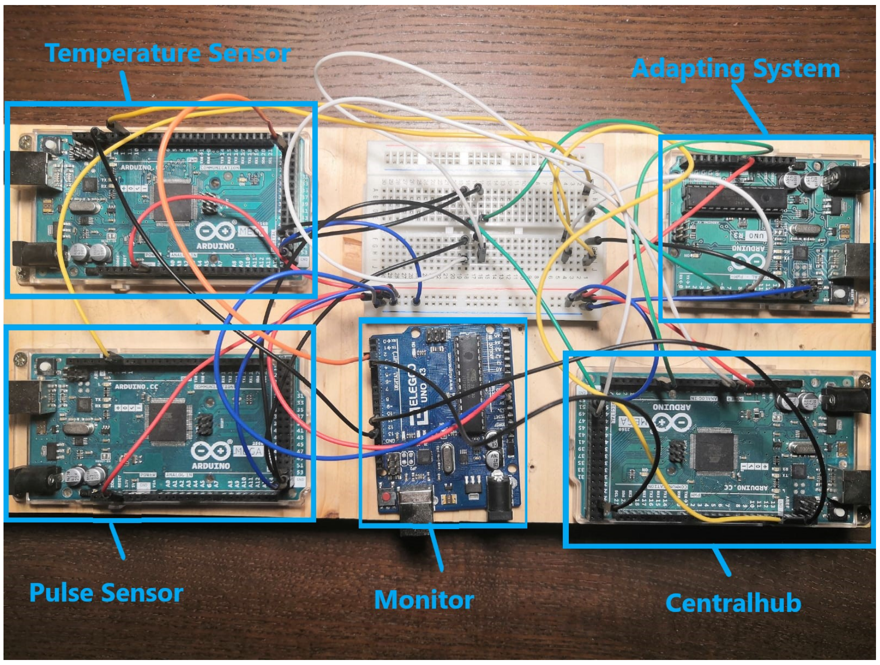


Figure 7: Setup of our experimentation system

Figure 7 shows our experimentation setup where each of the Arduinos is annotated with its programmed component. We chose Arduino as our monitor hardware mainly because of its advantages in size and convenience to use. An Arduino as monitor does not require a lot of space and also has a low power consumption. The programming language for Arduinos is `C++` which makes it easy to adapt our source code to other hardware environments for further usage outside of the Arduino context. In its current state, the BSN for Arduino partially does not provide sufficient logging for the properties we want to verify. If that were the case we would have used an additional Arduino as adapter to transform the logging output to events that can be used by the monitor. Without the logging, we instrumentalize the BSN directly by adding the correct structured events needed by the monitor into the correct places of the BSN. That way, we simulate actual logging, which would need to happen anyway for our monitor to work. Additionally, we don't have redundant implementation work by having to add logging to the BSN first, before creating an adapter which transforms the logging to

usable event logs for our monitor.
We evaluate multiple properties, which will be discussed in Section 4.3. For each property, we altered the basic centralhub or sensor implementation separately. This is done because by monitoring multiple properties on the same system would lead to the need of adding at least one adapter Arduino and also one additional Arduino per property. Our adapter would just emulate the one to one interaction between the centralhub and the monitor, therefore we would not gain a significant advantage for our evaluation despite the necessity of additional hardware.

4.3 Tested Properties

This section will provide information about our evaluation scenario and at the same time will attend to the research question **RQ2** (How many properties can we express?). We tested our approach by creating properties for our BSN system. Table 3 shows what properties we try to evaluate. We also provided the specification pattern, the property is related to. Note that these are not the only patterns our approach can transform into a runtime monitor. Our system should be able to verify any property that is based on one of Dwyer's specification patterns[10] and also complies with the limitations we proposed in Section 3.2. In Section 2.5 we already mentioned that the UPPAAL observers are based on the TCTL formulas of the specification patterns. This could be a constraint for the number of properties we can express, because we can only validate single execution paths and not all CTL/TCTL functions can be expressed in CTL/MTL. Fortunately, the way we require the observers to be constructed in Section 3.2, we only need to validate the query `A[] not ERROR` which is the UPPAAL notation for $AG\neg ERROR$. This formula is equivalent to the LTL formula $G\neg ERROR$ which makes it possible for us to verify this behaviour with finite execution traces. Therefore, we are limited to the existence of an observer for the specification pattern, which satisfies the assumptions from Section 3.2 we mention.

P01	Globally, *sensor sends message*, holds repeatedly [every 2200ms].	Recurrence Globally Timed
P02	Globally, *sensor sends message*, then in response *centralhub receives message* eventually holds [1200ms].	Response Globally Timed
P03	Globally, once *centralhub receives sensor data* it remains so for less than 5200ms.	Maximum Duration Globally
P04	Globally, *centralhub detects emergency*, then it must have been the case that *sensor detects emergency* [has occured] before *centralhub detects emergency* [holds].	Precedence Globally Timed
P05	After *sensor starts*, it is always the case that *sensor is alive* [holds]	Universitality After

Table 3: Properties verified in the thesis

P01 This property verifies if the sensor, which the monitor is connected to, sends a message to the centralhub atleast once every 2200ms. The UPPAAL observer is shown in Figure 8. The sensors are programmed to send a message to the centralhub every 2000ms. We give the sensor additional 200ms to compensate for a delay in reception and transmission of the message. For this property, we added one `P_reached?` and one `P_left?` event to the sensor when he starts to send its data to the centralhub.

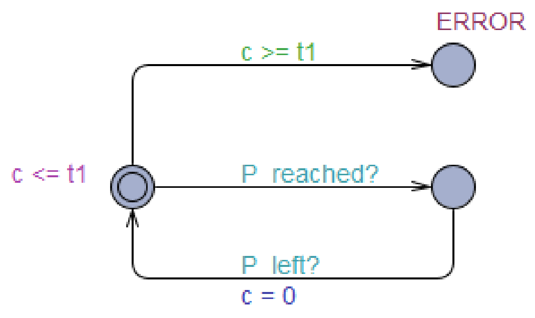


Figure 8: UPPAAL Observer for the Recurrence globally timed pattern.

P02 This property monitors if we get a response from the centralhub, when the sensor sends data. Its UPPAAL observer is depicted in Figure 9. We want to make sure, that information of the sensor actually reaches the centralhub for further processing. This was a challenge because we had to handle the messages of multiple devices. We therefore instrumentalized our monitor to emulate an adapter that handles the communication. The sensor and the centralhub use pins to notify the adapter of occurring events. Digital pins in Arduino can have the values `LOW` and `HIGH`. The adapter translates the pin values into fitting events, the monitor can evaluate.

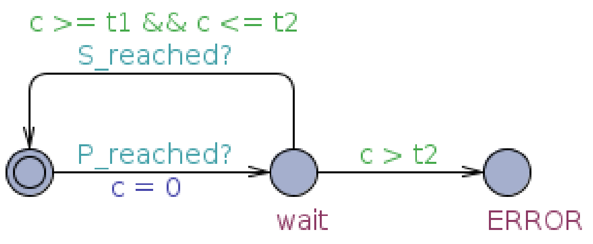


Figure 9: UPPAAL Observer for the response globally timed pattern.

P03 The property checks how long it takes the centralhub to evaluate the information send by the sensor and calculate the risk. The centralhub collects data and evaluates the risk based on the vital parameters observed by the sensors. If calculation time is delayed, the centralhub also takes more time to report its results. The maximum duration pattern observer of PSP-UPPAAL does not provide an error state. As mentioned in our assumptions in Section3.2 our generated monitors require an error location to check whether an error occured or not. Therefore we changed the initial observer by adding an ERROR location and extending the original observer with fitting guards. Both observers are shown in Figure 10.

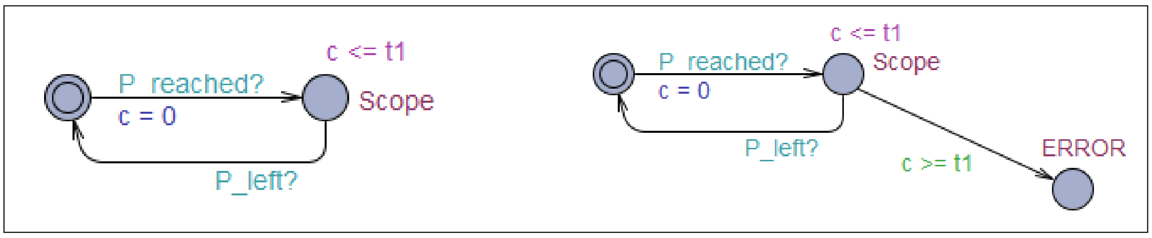


Figure 10: Original observer without error location (left) and the adjusted observer with error location (right)

P04 The property monitors, that the centralhub receives data from the sensor, if the sensor send data beforehand. This property does not work well with runtime monitoring because as shown in Figure 11 it has no *looping* behaviour. If `P_reached?` is received first, the state machine transitions to the error state. Receiving a `S_reached?` first on the other hand will satisfy the property and the state machine transitions to a state without outgoing edges. We evaluate this property by starting the monitor with or without faulty behaviour and measuring the time it takes to get from the start of the monitor to the `error` or `succeeded` state. Note that the state is not labelled. We use the term `succeeded` state to denote the state, which is not an error state, but also does not have any outgoing transitions. As communication adapter, we again instrumentalized our monitor to listen on specific pins and receive events based on the pin state.

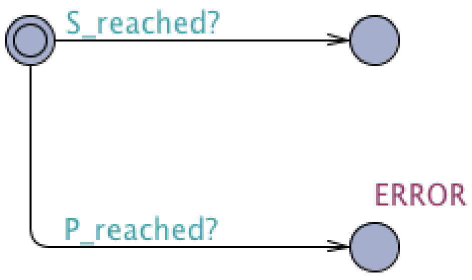


Figure 11: UPPAAL observer for the precedence globally timed pattern

P05 Figure 12 shows the UPPAAL observer of our property **P05**. This property ensures, the sensor is always online after its initial start-up. If the sensor fails, the monitor notifies the user. We utilize the `Vin` pin on the Arduino which is always `HIGH` if the Arduino has power. The *sensor starts* event is emulated by the adapter if the `Vin` pin is `HIGH` and to check if the sensor is still alive, we send the `P_left?` when the `Vin` is `LOW`.

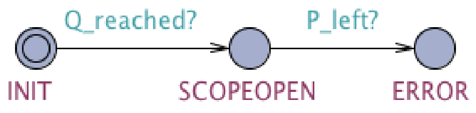


Figure 12: UPPAAL observer for the universality after timed pattern

4.4 Experimental Results

This section contains our results of the evaluation combined with problems which occurred during the evaluation.

4.4.1 Memory Usage

Here we will answer the research question **RQ3** (How much memory is needed for the monitor?). Table 4 shows the memory usage of our monitors. We provide the memory allocation of the dynamic and programmable memory of the Arduino. The percentages are based on the maximum memory of an Arduino Uno. The Arduino Uno has 32KB of programmable memory, which is used to store the code of our monitor and 2KB of static random access memory which is used to create and manipulate variables during runtime[7]. The memory usage of all the monitors is similar, which is not surprising because they share most of their code. The differences in memory can be explained with the different number of states and transitions, but also the varying guards, invariants and assignments. As shown, there is still enough space for more complex properties on the Arduino Uno. If necessary for future work, we still have the option to upgrade to the Arduino Mega which provides 256KB of programmable memory and 8KB of static random access memory. The overhead generated through the monitor is always an entire Arduino. We do not utilize the full memory space of the Arduino, but we also can not use the remaining capacity for anything else. Additional functionality would alter the precision and behaviour of our monitor.

Property	Dynamic Memory (%)	Programmable Memory (%)
P01	1223 Bytes (59%)	10392 Bytes (32%)
P02	1267 Bytes (61%)	10708 Bytes (33%)
P03	1229 Bytes (60%)	10398 Bytes (32%)
P04	1211 Bytes (59%)	10528 Bytes (32%)
P05	1221 Bytes (59%)	10520 Bytes (32%)

Table 4: Memory usage of our monitor on an Arduino Uno

4.4.2 Generation Time and Detection Time

We measured the generation time of our monitors to answer the research question **RQ4** (How long does the monitor generation take?). This does not include initial

[7] `https://www.arduino.cc/en/Tutorial/Foundations/Memory`

installations or the start of the gradle daemon. Figure 13 depicts the average generation time for each of the selected properties.

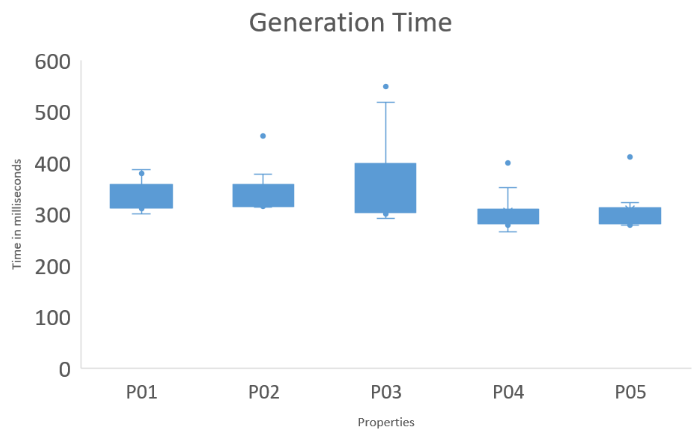


Figure 13: Measured time of the monitor generatiion

We generated each monitor ten times and deleted the monitor file before every execution. None of the monitors took longer than a second to be generated. Having short generation times adds to the accessibility of our lightweight approach. The variance in our measurement could occur due to caching. Each time we adjust a monitor configuration or change the desired property, the generation is about 100ms longer.
The next time we measured was the detection time or *time till error* of our runtime monitors in the context of research question **RQ5** (How fast can we detect errors?). Figure 14 shows the average time until the error state is reached on every monitor.

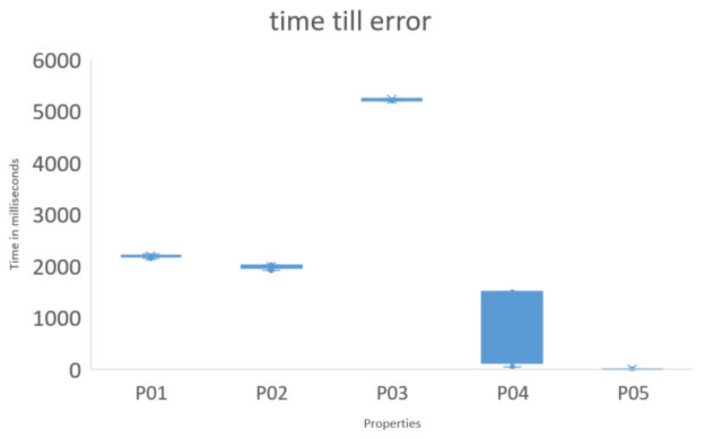


Figure 14: Measured time of the error detection

The error was introduced to the system by manipulating the hardware. This was achieved by either removing the cable or pulling the voltage of a pin to ground. Note that simply removing a cable from a pin does not necessary result in the pin voltage being `LOW`. An unconnected pin will have a floating voltage if not declared as pull-up pin or connected to ground. Therefore removing a cable does lead to "random" behaviour of the pin. The start point of the measurement was always the state which was directly connected to the `error` state. The properties we evaluate do not have more than one incoming transition in their `error` state. If multiple paths to the `error` state existed, we would have to measure multiple scenarios separately. Using other states as measurement starting point would lead to deviations in detection times because the state machine could idle in some of those states which would artificially increase the measured times. Therefore, to reach comparable values, we only took the last transition into consideration and measured the time it took until the state machine switched to the `error` state.

In case of property **P04**, where we have no looping behaviour and the initial state is directly connected to the `error` state, we use the start of the monitor as starting point for our measurement. We made this decision, as we do not print the current state of our state machine, if no transition is fired, to keep the traffic low. As expected, the detection time of the timed properties is close to their time constraints. The variance can occur due to delay in the serial output of the Arduino monitor as well as delay due to the time the monitor loop requires to evaluate the error.

Property **P04** varies the most, covering almost the whole interval from $0ms$ to $2000ms$. This can be explained when we look at the process of the system. The property fails, if the centralhub sends a message that it received data from the sensor, before the sensor even sends the data. We removed the connection between the monitor and the sensor to force a property failure. The sensor still sends data to the centralhub every 2000ms and therefore the centralhub will also notify the monitor every 2000ms. Because there is no loop and we need to start the serial monitor which does print everything with the current timestamp, we always hit the error state somewhere in the two second interval.

For property **P05** we did not measure the time between two states, but rather the time between the incoming event, that the sensor became unavailable, and the switch to the error state. The times measured were all between 1ms and 1.2ms with the exception of one outlier at 2.5ms.

4.5 The Clock and Real-time Constraints

The non-intrusive approach does not have access to the internal clock of the monitored system, which means that the monitor needs to provide its own clock to handle timing constraints. The hardware we use for our experiments does not provide parallelism in any way. These characteristics make precise time management challenging. If our implementation is not fast enough, the time it takes to generate a new clock value can be too long so that we possibly miss specific time intervals, which are necessary for the verification. Having to deal with real-time properties is a common challenge in runtime verification[38]. As we describe in Section 2.1, we are working with continuous times in the physical world and therefore guarantee to verify a guard like $time == 100ms$ in a state machine at the exact time. There is always at least a minimal deviation because

of time continuity, so we can only try to get as close as possible to the time constraint, by creating efficient code and using fast and lightweight hardware if feasible. There are multiple possible improvements for time management on Arduino, one of them being *Interrupt Service Routines (ISR)*[8]. When a hardware interrupt is triggered, the hardware interrupts the current execution and saves the system state to execute the related ISR. This trigger could be an external clock or the internal time register for example[2]. This creates multiple challenges. When we update the clock via ISRs, we can still miss timestamps, because our loop can take too long. Another idea would be to handle the state machine traversal and guard checking via ISRs. There we need to find a fitting trigger for the interrupts and a solution to keep track of the current state in our state machine.
All of this needs more research and could lead to future work.

4.6 Feasibility

In this section we take a look at the feasibility of the system. We already mentioned timing issues in earlier Sections 4.5 and 4.1.2. These pose a risk to the correctness of our state machine, as well as the feasibility. As stated in the Introduction in Section 1 runtime verification can be beneficial for safety-critical systems. Especially in systems with natural high reliability requirements, dealing with monitor deviations of up to 150ms can change the behaviour and poses an unnecessary risk. This is the case for any system that requires precise behaviour in the millisecond range. In those cases, our approach is no feasible addition to the system and should not be the only runtime verification in place. There are better alternatives, which either use more efficient hardware [35] or are build into the system to keep track of internal variables and even apply error handling routines to reduce manual intervention[32, 29]. Therefore, the idea of creating a system which automatically generates runtime monitors for safety-critical systems should not be the goal for this thesis. Furthermore we want to set the focus on the accessibility of our system and build a foundation for future research in this scientific field.

[8]`https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/`

5 Discussion

In this section we want consider some challenges which occurred during our experiments. First of all, the communication of the Arduinos via `SoftwareSerial` and via an adapter works, but can be unreliable. Sometimes we had to restart the monitor multiple times for it to recognize the changes of pin voltages of the communication partner. This can be fixed by adding delays to the sender, so that the receiver has more time to recognize the transmission. One of the problems was, that we were not allowed to make large changes to the underlying Arduino BSN, if we wanted to keep its functionality. The BSN implementation also uses communication and therefore schedules its transmissions in a way that the sensors do not send messages at the same time for example. This made changes to the transmission time difficult. For **P04** we had to add 50ms delay to achieve some reliability to the monitor. Having a logging component built into the monitored system can also increase the reliability, but has to be adjusted from system to system.
Another problem was that we had to deal with message loss. Especially in **P03**, the monitor sometimes received faulty strings `_reached?` and `_left?`, where the last one made the monitor detect an error. At this point in time, we do not provide a solution for this. A solution could be to manually map pins of the monitor to specific events and work with voltages, but we would still need to receive information from the monitored system on the adapter.

An additional challenge we had during the evaluation was the output of our monitor. The only output our monitors produce are the processed events and the target state, if a transition is fired. We could not generate output with more information value because the monitor is not able to print the information fast enough compared to the loop frequency. This led to monitor crashes. As a consequence, we reduced the amount of output printed as described above.

6 Threats to Validity

In this section we talk about possible threats to the validity of this thesis. We therefore split the threats in different parts, the internal, external and construct validity.

Internal Validity We made changes to the BSN so it can output events that occur during its runtime. These changes are only minor additions and our approach would need some kind of logging component in any case, but we can not be sure that they did not alter the behaviour of our monitored system. Therefore it could be the case that our approach does not work with the original implementation on the Arduino platform. Another threat is the possibility of message loss that can occur with our hardware, as discussed in Section 5. This threat can lead to faulty verification results because important events were either faulty or not received at all. In addition to that the synchronisation of the monitor with the monitored system posed another threat. It could happen, that the monitor misses important events such as rising edges of pins because the calculations take too long. This threat can be mitigated by implementing an efficient adapter to handle those signal changes and transforming them into the specific events for the generated monitor.
We also added a grace period to our time constrained guards to compensate for the inaccuracy of our monitor time handling. We already addressed this issue in Section 4.5 combined with possible solutions for future work. This problem poses less of a threat to the correctness of our monitor but more to the efficiency.
In addition to that, our experiments only cover five specification patterns. Though we tried to choose a wide spectrum of different patterns, there can still exist a specification pattern, for which our monitor generation produces faulty code. Defining properties for each pattern, adjusting the events sent by the BSN and running the experiments for every possible pattern would not have been feasible in the time frame of this thesis.

External Validity We were only able to test our approach on the BSN system. Though we are confident, that the generation of runtime monitors should work with any system, because it utilizes the generalisation the specification patterns provide, we can not be sure that this is actually the case without testing it on more than one system.

Construct Validity Our approach utilizes multiple systems. Each of those systems poses a threat to the validity of our system. First, we rely on the correctness of Dwyer's specification patterns. If the pattern does not mirror the desired requirement, our monitor also will not verify the correct requirement. These specification patterns are renown and widely used for requirement specifications and as groundwork for further research[20, 3] and therefore pose a small risk to our approach.
For our implementation we use the custom Arduino libraries *ArxContainer*[9] and *SoftwareSerial*[10]. The *ArxContainer* library is a lightweight implementation of the

[9] `https://www.arduino.cc/reference/en/libraries/arxcontainer/`
[10] `https://docs.arduino.cc/learn/built-in-libraries/software-serial`

C++ standard library container structures we use for an implementation of `map`, `deque`, `vector` and `array`. *SoftwareSerial* is used to provide additional serial pins for the communication between Arduinos. The libraries are still maintained regularly and we tested all functions used to make sure they work as intended. However we can not rule out that there may occur faulty software behaviour because of those libraries.

7 Related Work

In the runtime verification context there already exist a bunch of approaches for automatic runtime monitor generation. In this section we take a look at different approaches for runtime verification monitors and talk about the differences to our approach. One system which is closely related to the idea of our approach, is the `OGMA` system[29]. This system acts as a bridge between the Formal Requirements Elicitation Tool (FRET)[12] and COPILOT[7]. Similar to Dwyer's specification patterns[10], which we use to generate logical formulas from structured natural language, `FRET` is used and extended by `OGMA` to generate a formal component specification out of "specialized natural language"[12]. This specification is then utilized to configure a COPILOT monitor. The generated COPILOT monitors are no standalone executable application and are designed with the idea in mind to be embedded inside the monitored application so they can access and alter the system variables to handle property violations. Those violation handlers have to be manually written by the user. The authors provide a depiction of their workflow which can be seen in Figure 15.

Though a lot of similarities exist between our approaches there are also significant differences between them. `OGMA` utilizes stream-based monitoring. Stream-based monitoring utilizes arithmetic operations on streams and evaluates streams to determine whether a property failed or not. The advantage of stream-based monitoring is the expressiveness of the data the monitor can evaluate. While our approach is based on boolean events and therefore needs precise events to validate a property, streams provide the ability to evaluate other data such as integers and aggregate this information. This enables them to verify different properties, without additional effort. Another difference is the way we handle updates on our monitors. The COPILOT monitors need to access the external variables of the system, which is why the authors advice the user to embed the monitors into the application. Even if built outside the system, the monitor still relies on having on-demand access to the variables in specific time intervals, because COPILOT utilizes sampling-based monitoring[31][32].

Sampling-based monitoring is done by updating the information, the monitor receives from the monitored system on predefined time intervals. This type of monitoring can solve the timing issues we presented earlier, because the monitor can act on those intervals without the need to listen to events between those times. While this approach can be prone to error, because the sampling could lead to omitted information, the authors propose a solution to this by making the monitor and the monitored "system share the same global clock and providing static periodic schedule"[31]. This property makes the monitors invasive by nature, which differs vastly from our approach. Our event-based approach has the advantage of not intervening with the monitored system by just catching communication traffic. That way, we do not interfere the monitored systems performance or behaviour in any way. This also leads to the problem that we need to process events when they arise and therefore we are unable to do sampling-based monitoring, which would solve our timing issues.

While `OGMA` does not address the issues we are trying to solve in our thesis, because of the invasive nature of their monitors it looks like a powerful tool for embedded runtime

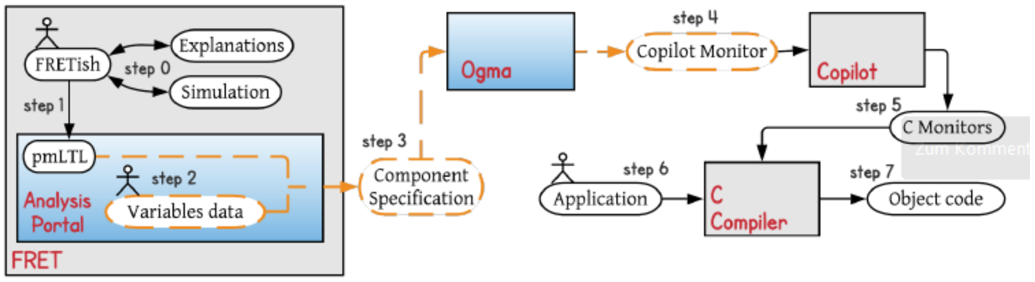


Figure 15: Workflow for automatic monitor generation out of structured natural language proposed by Perez et al. [29]

monitor verification, because it also allows for direct error handling by providing the option to specify handler functions for violated properties.

Another approach that actually utilizes event-based monitoring for unsynchronized systems is the specification language RTLola[34]. RTLola is an extension of the specification language Lola[8] which adds real time monitoring and works with asynchronous systems. The authors wrote multiple papers describing the workflow with RTLola[35]. The approach also uses natural language that is converted into logical formulas for their property specification. The authors provide compiler for *FPGA* and *RUST* monitors and a pre-compiled interpreter[11]. RTLola utilizes input streams for its monitoring to produce corresponding output streams based on the underlying specification. The interval in which new values are processed for the output streams can be configured either by providing a frequency or setting the computation to event-based. In case of the event-based monitoring new values for the output stream are generated only if new values arrive in the input stream. While working with unsynchronized systems, the authors still assume that the events from different components are received in the correct order, respective to their time stamps.
Another real-time based approach from Leucker et al.[24] also addresses stream-based monitoring. This approach also takes asynchronous events but does not rely on an arrival order of events from different components. The authors use $TeSSLa_a$[19] as their specification language of choice.
None of the mentioned real-time based approach which work with asynchronous systems provide automatic monitor generation. Each of the monitor specifications has to be specified manually using the specification language provided.
Another approach which does not utilize stream-based runtime verification is `MESA`[39]. `MESA` is a non-intrusive runtime verification framework which utilizes the communication messages of different actors in the system to verify properties. These properties can be specified using temporal logic and finite state machines[38]. Similar to our approach they verify the specified property by traversing the state machine based on the monitored messages. Another similarity to our system is that `MESA`, according to its authors, "[...] is not suitable for verifying hard real-time systems where there are time

constraints on the system response."[38]. There also exist differences between their system and our approach. Compared to us, they provide a much higher reliability by utilizing a multi-step publish/subscribe communication combined with hardware that has much higher computation power. Our advantage compared to their approach on the other hand is, that we do not rely on an manual state machine specification which was designed for one property only, because we can utilize the specification patterns and the observer models provided by PSP-UPPAAL.

8 Conclusion and Future Work

In this thesis we presented an approach to automatically generate executable code for runtime montiors on lightweight hardware with limited memory. We showed how to build C++ state machines based on UPPAAL models. This enabled us to verify properties, which were specified for the system model, during runtime. We also provided examples of the generation process and used a BSN implementation on Arduinos for our monitored system, which we also used for our evaluation. Our experiments showed that our system generates correct runtime verification monitors for untimed specification patterns with minimal configuration, given an UPPAAL observer from PSP-UPPAAL exists. We presented the short generation time of our monitors and showed that the memory capacity of an Arduino Uno is sufficient to cover them. The simplicity of the approach makes runtime verification accessible for a variety of systems which have no runtime verification implemented yet but already made the effort to specify the requirements for their system using Dwyer's specification patterns[10].

We also reflected on the systems limitations and drawbacks. While producing seemingly correct monitors for untimed properties, our approach struggles with precise time constraints. Right now, we are not able to evaluate specific guards or invariants without any deviation, because we work with unsynchronized systems and our monitors utilize hardware that does not provide parallelization. We also are vulnerable to message loss between the monitor and the system. We proposed solution ideas to overcome these issues, which present a good opportunity for future work.
Another future addition to our system can be the evaluation of formulas of the UPPAAL generator. The addition would remove our reliance on error states and therefore widen the range of different properties, we can monitor without altering the original UPPAAL observer.
In addition to that, when working with self-adaptive systems, the idea of adaptive system requirements comes into mind. The possibility to change the runtime monitors during runtime based on the adjustments made by the adapting component can lead to more dynamic and precise verifications. Applying this property to our monitor generator and therefore creating monitors that adapt their constraints and specifications during runtime, is also an interesting idea for future work.
We already mentioned the advantages of stream-based monitors in the related work Section 7.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. “Model-Checking in Dense Real-Time”. In: *Information and Computation* 104.1 (May 1993), pp. 2–34. ISSN: 08905401. DOI: `10.1006/inco.1993.1024`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0890540183710242` (visited on 10/22/2022).

[2] *Arduino Tutorial - Timer*. EXP Tech. URL: `https://www.exp-tech.de/blog/arduino-tutorial-timer` (visited on 09/25/2022).

[3] Marco Autili et al. “Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar”. In: *IEEE Transactions on Software Engineering* 41.7 (July 2015). Conference Name: IEEE Transactions on Software Engineering, pp. 620–638. ISSN: 1939-3520. DOI: `10.1109/TSE.2015.2398877`.

[4] Howard Barringer et al. “Rule-Based Runtime Verification”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernhard Steffen and Giorgio Levi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 44–57. ISBN: 978-3-540-24622-0. DOI: `10.1007/978-3-540-24622-0_5`.

[5] Ezio Bartocci et al. “Introduction to Runtime Verification”. In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Cham: Springer International Publishing, 2018, pp. 1–33. ISBN: 978-3-319-75632-5. DOI: `10.1007/978-3-319-75632-5_1`. URL: `https://doi.org/10.1007/978-3-319-75632-5_1`.

[6] *carwehlm/masterthesis*. GitHub. URL: `https://github.com/carwehlm/masterthesis` (visited on 09/19/2022).

[7] *Copilot: Stream DSL for hard real-time runtime verification*. original-date: 2015-06-09T23:48:41Z. Aug. 31, 2022. URL: `https://github.com/Copilot-Language/copilot` (visited on 09/01/2022).

[8] Ben D’Angelo et al. “Lola: Runtime Monitoring of Synchronous Systems”. In: *12^th International Symposium on Temporal Representation and Reasoning (TIME’05)*. event-place: Burlington, Vermont. IEEE Computer Society Press, June 2005, pp. 166–174.

[9] Nilanjan Dey et al. “Medical cyber-physical systems: A survey”. In: *Journal of Medical Systems* 42.4 (Mar. 10, 2018), p. 74. ISSN: 1573-689X. DOI: `10.1007/s10916-018-0921-x`.

[10] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. “Patterns in Property Specifications for Finite-State Verification”. In: *Proceedings of the 1999 International Conference on Software Engineering, ICSE’ 99, Los Angeles, CA, USA, May 16-22, 1999*. Ed. by Barry W. Boehm, David Garlan, and Jeff Kramer. ACM, 1999, pp. 411–420. DOI: `10.1145/302405.302672`.

[11] Peter Faymonville et al. “StreamLAB: Stream-based Monitoring of Cyber-Physical Systems”. In: *Computer Aided Verification.* Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 421–431. ISBN: 978-3-030-25539-8 978-3-030-25540-4. DOI: 10.1007/978-3-030-25540-4_24. URL: http://link.springer.com/10.1007/978-3-030-25540-4_24 (visited on 09/19/2022).

[12] *FRET: Formal Requirements Elicitation Tool.* original-date: 2019-11-26T04:56:57Z. Aug. 25, 2022. URL: https://github.com/NASA-SW-VnV/fret (visited on 09/01/2022).

[13] Eric Bernd Gil et al. *Body Sensor Network: A Self-Adaptive System Exemplar in the Healthcare Domain.* 2021. arXiv: 2103.14948 [cs.SE].

[14] Valentin Goranko and Antje Rumberg. “Temporal Logic”. In: *The Stanford Encyclopedia of Philosophy.* Ed. by Edward N. Zalta. Summer 2022. Metaphysics Research Lab, Stanford University, 2022. URL: https://plato.stanford.edu/archives/sum2022/entries/logic-temporal/ (visited on 10/26/2022).

[15] Lars Grunske. “Specification patterns for probabilistic quality properties”. In: *2008 ACM/IEEE 30th International Conference on Software Engineering.* 2008, pp. 31–40. DOI: 10.1145/1368088.1368094.

[16] Shu-Ming Hsieh, Chiun-Chieh Hsu, and Li-Fu Hsu. “Efficient Method to Perform Isomorphism Testing of Labeled Graphs”. In: *Computational Science and Its Applications - ICCSA 2006.* Ed. by Marina L. Gavrilova et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 422–431. ISBN: 978-3-540-34080-5. DOI: 10.1007/11751649_46.

[17] Fei Hu. *Cyber-physical systems.* Taylor & Francis Group LLC, 2014.

[18] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Google-Books-ID: eUggAwAAQBAJ. Cambridge University Press, Aug. 26, 2004. 326 pp. ISBN: 978-1-139-45305-9.

[19] Hannes Kallwies et al. “TeSSLa – An Ecosystem for Runtime Verification”. In: *Runtime Verification.* Ed. by Thao Dang and Volker Stolz. Vol. 13498. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 314–324. ISBN: 978-3-031-17195-6 978-3-031-17196-3. DOI: 10.1007/978-3-031-17196-3_20. URL: https://link.springer.com/10.1007/978-3-031-17196-3_20 (visited on 10/06/2022).

[20] Aaron Kane. “Runtime Monitoring for Safety-Critical Embedded Systems”. In: 2015. DOI: 10.1184/r1/6721376.v1.

[21] J.C. Knight. “Safety critical systems: challenges and directions”. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. May 2002, pp. 547–550.

[22] S. Konrad and B.H.C. Cheng. “Real-time specification patterns”. In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. ISSN: 1558-1225. May 2005, pp. 372–381. DOI: 10.1109/ICSE.2005.1553580.

[23] Daniela Lepri, Erika Ábrahám, and Peter Csaba Ölveczky. “Timed CTL Model Checking in Real-Time Maude”. In: *Rewriting Logic and Its Applications.* Ed. by Franciso Durán. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 182–200. ISBN: 978-3-642-34005-5. DOI: 10.1007/978-3-642-34005-5_10.

[24] Martin Leucker et al. “Runtime verification of real-time event streams under non-synchronized arrival”. In: *Software Quality Journal* 28.2 (June 2020), pp. 745–787. ISSN: 0963-9314, 1573-1367. DOI: 10.1007/s11219-019-09493-y. URL: https://link.springer.com/10.1007/s11219-019-09493-y (visited on 10/02/2022).

[25] Kasper Luckow. *JUppaal.* original-date: 2016-02-02T22:51:25Z. May 19, 2020. URL: https://github.com/ksluckow/juppaal (visited on 09/14/2022).

[26] Saraju Mohanty. “Advances in Transportation Cyber-Physical System (T-CPS)”. In: *IEEE Consumer Electronics Magazine* 9 (July 1, 2020), pp. 4–6. DOI: 10.1109/MCE.2020.2986517.

[27] Anik Momtaz. “Runtime Verification for Distributed Cyber-Physical Systems”. In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS).* 2021 40th International Symposium on Reliable Distributed Systems (SRDS). ISSN: 2575-8462. Sept. 2021, pp. 349–350. DOI: 10.1109/SRDS53918.2021.00044.

[28] James Nutaro. “Toward a Theory of Superdense Time in Simulation Models”. In: *ACM Trans. Model. Comput. Simul.* 30.3 (May 2020). ISSN: 1049-3301. DOI: 10.1145/3379489. URL: https://doi.org/10.1145/3379489.

[29] Ivan Perez et al. “Automated Translation of Natural Language Requirements to Runtime Monitors”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 387–395. ISBN: 978-3-030-99524-9.

[30] Lee Pike, Sebastian Niller, and Nis Wegmann. “Runtime Verification for Ultra-Critical Systems”. In: *Runtime Verification.* Ed. by Sarfraz Khurshid and Koushik Sen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 310–324. ISBN: 978-3-642-29860-8. DOI: 10.1007/978-3-642-29860-8_23.

[31] Lee Pike et al. “Copilot: A Hard Real-Time Runtime Monitor”. In: vol. 6418. Nov. 1, 2010, pp. 345–359. ISBN: 978-3-642-16611-2. DOI: 10.1007/978-3-642-16612-9_26.

[32] Lee Pike et al. “Copilot: monitoring embedded systems”. In: *Innovations in Systems and Software Engineering* 9.4 (Dec. 2013), pp. 235–255. ISSN: 1614-5046, 1614-5054. DOI: `10.1007/s11334-013-0223-x`. URL: `http://link.springer.com/10.1007/s11334-013-0223-x` (visited on 09/19/2022).

[33] *Property Specification Patterns for UPPAAL.* original-date: 2021-01-12T15:55:13Z. Dec. 16, 2021. URL: `https://github.com/hub-se/PSP-UPPAAL` (visited on 09/06/2022).

[34] *RTLola.* URL: `https://www.react.uni-saarland.de/tools/rtlola/` (visited on 09/05/2022).

[35] Maximilian Schwenger. “Monitoring Cyber-Physical Systems: From Design to Integration”. In: *Runtime Verification.* Ed. by Jyotirmoy Deshmukh and Dejan Ničković. Cham: Springer International Publishing, 2020, pp. 87–106. ISBN: 978-3-030-60508-7. DOI: `10.1007/978-3-030-60508-7_5`.

[36] Koushik Sen and Grigore Roşu. “Generating Optimal Monitors for Extended Regular Expressions”. In: *Electronic Notes in Theoretical Computer Science* 89.2 (Oct. 2003), pp. 226–245. ISSN: 15710661. DOI: `10.1016/S1571-0661(04)81051-X`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S157106610481051X` (visited on 10/06/2022).

[37] Sanjit A Seshia. “Introduction to Temporal Logic”. In: (), p. 28.

[38] Nastaran Shafiei, Klaus Havelund, and Peter Mehlitz. “Actor-Based Runtime Verification with MESA”. In: *Runtime Verification.* Ed. by Jyotirmoy Deshmukh and Dejan Ničković. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 221–240. ISBN: 978-3-030-60508-7. DOI: `10.1007/978-3-030-60508-7_12`.

[39] Nastaran Shafiei, Oksana Tkachuk, and Peter Mehlitz. “MESA: Message-Based System Analysis Using Runtime Verification”. In: (), p. 16.

[40] *SoftwareSerial Library | Arduino Documentation.* URL: `https://docs.arduino.cc/learn/built-in-libraries/software-serial` (visited on 09/11/2022).

[41] Thomas Vogel and Marc Carwehl. *Property Specification Patterns for UPPAAL.* original-date: 2021-01-12T15:55:13Z. Dec. 16, 2021. URL: `https://github.com/hub-se/PSP-UPPAAL` (visited on 08/18/2022).

[42] *UPPAAL.* URL: `https://uppaal.org/` (visited on 09/20/2022).

[43] Thomas Vogel et al. *A Property Specification Pattern Catalog for Real-Time System Verification with UPPAAL.* Rochester, NY, Dec. 21, 2021. DOI: `10.2139/ssrn.3990519`. URL: `https://papers.ssrn.com/abstract=3990519` (visited on 10/17/2022).

[44] Jean-Paul A. Yaacoub et al. “Cyber-physical systems security: Limitations, issues and future trends”. In: *Microprocessors and Microsystems* 77 (Sept. 2020), p. 103201. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2020.103201. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7340599/ (visited on 10/26/2022).

[45] Man Zhang et al. “Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model”. In: *Modelling Foundations and Applications*. Ed. by Andrzej Wąsowski and Henrik Lönn. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 247–264. ISBN: 978-3-319-42061-5. DOI: 10.1007/978-3-319-42061-5_16.

[46] Xi Zheng et al. “BraceAssertion: Runtime Verification of Cyber-Physical Systems”. In: *2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems*. 2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems. Oct. 2015, pp. 298–306. DOI: 10.1109/MASS.2015.15.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 26. Oktober 2022 Tim Jellinek