UNIVERSITÄT ZUM BEISPIEL
INSTITUT FÜR BEISPIELE

Monitoring der AUTOSAR Timing Extensions mittels TeSSLa

Monitoring of the AUTOSAR Timing Extensions with TeSSLa

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Hendrik Streichhahn

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Dr. Martin Sachenbacher und
Daniel Thoma

Lübeck, den 1.1. 1970

Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Hendrik Streichhahn)
Lübeck, den 1.1. 1970

Kurzfassung Abstract Deutsch

Abstract Kurzfassung Englisch.

Contents

Liste der Todos

1. Introduction

Timing behavior is one of the most important properties of computer systems. Especially in safety-critical applications, a wrong timed reaction of the system can have disastrous consequences, for example in the Electronic Stability Control of a vehicle. The *AUTOSAR* (**AUT**omotive **O**pen **S**ystem **AR**chitecture) standards are used by almost all car manufacturers in their software development processes to standardize components and therefore increase the interoperability and exchangeability.
To describe the timing behavior of soft- and hardware components of cars, the *AUTOSAR Timing Extensions* were developed. The goal of this thesis is to implement a monitoring tool for the timing constraints defined in this standard.
Some of the constraints defined in the *AUTOSAR* standard are written in an informal way and can be misunderstood, which will be describe as part of this thesis. This is problematic for monitoring, because the implementation of a monitor should be based on unambiguous definitions. To solve this problem, the timing constraints defined in *TADL2* (**T**iming **A**ugmented **D**escription **L**anguage Version **2**) are used as basis for the monitoring tool. The TADL2 timing constraints are comparable and partly compatible to the AUTOSAR Timing Extensions, as most of the constraints defined in the AUTOSAR standard can be described as equivalent combination of TADL2 timing constraints.
The monitoring tool is written in *TeSSLa* (**T**emporal **S**tream-based **S**pecification **L**anguage), which is made for stream runtime verification and is capable of non-intrusive observation and can be run as Java program or on specialized embedded hardware, like FPGAs.

In the first part of this thesis, an overview over the AUTOSAR Timing Extensions and an example about the informal and ambiguous definitions will be given. Next, the TADL2 timing constraints will be listed and the relations between the these constraints and the AUTOSAR Timing Extensions will be described. In the next chapter, TeSSLa, its fundamental functionality and other prerequisites, which are needed for understanding the theoretical part of this thesis, will be explained. The term of *finite monitorability* is introduced, which insures, that a property on infinite streams can always be monitored with finite resources. Then, each of the TADL2 timing constraint is checked, if it finite monitorable or not. After that, the TeSSLa implementations of these constraints is described and evaluated in a theoretical and practical way.

In the end an overview of the accomplished is given and ideas for further work will be discussed.

2. Timing Constraints

2.1. AUTOSAR Timing Extensions

AUTOSAR is a development partnership in the automotive industry. As stated before, the main goal is to define a standardized interface and increasing interoperability, exchangebility and re-usability of parts and therefore simplifying development and production. Three different layers are defined in the specification. *Basic Software* is an abstraction layer from components, like network or diagnostic protocols, or operating systems. *AUTOSAR-Software* defines the methods, how applications have to be build. For Basic Software and AUTOSAR Software, there are definitions for standardized Interfaces to enable the communication via the *AUTOSAR Runtime Environment*. It works as middleware, in which the *virtual function bus* is defined [AUT17]. The AUTOSAR Timing Extension are describing timing constraints for actions and reactions of components, that are communicating via the Virtual Function Bus. They are defined via *events*, which consists of a time and a data value, the type of the time and data value is arbitrary, the only restriction is, that the time values are strictly increasing. To describe the logical relationship between groups of events, *event chains* are defined, which consist of a *stimulus* and *response* event. The *response* event is understood as the answer to the *stimulus* event.
The AUTOSAR Release 4.4.0 ([AUT18]) is used for this thesis, there are 12 timing constraints defined in this version of the AUTOSAR Timing Extensions

1. The subset of 5 **EventTriggeringConstraints** are describing, at which points in time specific events may occur.

    1 The **PeriodicEventTriggering** defines repetitions of event with the same time distance and offers the possibility to set an allowed deviation from this pattern. Also the minimal distance between two subsequent events can be defined.

    2 The **SporadicEventTriggering** specifies sporadic event occurrences by defining the minimal and maximal distance between subsequent events. Optionally, periodic repetitions and allowed deviations from the period can be described.

3 With the **ConcreteEventTriggering**, offsets between a set of subsequent events in a time interval can be described. These intervals may not overlap, and periodic repetitions of them can be defined optionally.

4 The **BurstPatternEventTriggering** describes not overlapping event clusters with a minimal and maximal number of events and optionally periodic repetitions of these clusters.

5 The **ArbitraryEventTriggering** defines the distance between subsequent event by defining *ConfidenceIntervals*, which describe the probability, in which time interval the following event will occur.

2. The **LatencyTimingConstraint** specifies the minimal, nominal and maximal time distance between the stimulus and response events of an event chain.
3. The **AgeConstraint** is a simpler form of the *LatencyTimingConstraint* by defining minimal and maximal age a event may have at the point of time, when it is processed.
4. The **SynchronizationTimingConstraint** is used for describing events of different kinds, that occur synchronized in a time interval of a specific length.
5. The **SynchronizationPointConstraint** defines two sets of executables and events. Every element of the first set must have finished or occurred, before the first element of the second set may start or occur.
6. The **OffsetTimingConstraint** specifies the minimal and maximal time distance between corresponding *source* and *target* events.
7. The **ExecutionOrderConstraint** offers the possibility to define a list of executables, which must start and finish in the order given in the list.
8. The **ExecutionTimeConstraint** defines the minimal and maximal runtime of an executable, including or excluding the runtime of external functions and interruptions.

In this simplified form, some constraints are redundant. The semantic differences will be shown in section 2.2.3.

Problematic with the AUTOSAR Timing Extensions is, that the definitions are not very formal and have room left for interpretation. As example, the *BurstPatternEventTriggering* will be analyzed in the following. This constraint describes events clusters, with events that occur with short time distances, with larger time distances between the clusters. The following attributes are needed:

- ***maxNumberOfOccurrences*** (positive integer)
  Maximal number of events per burst

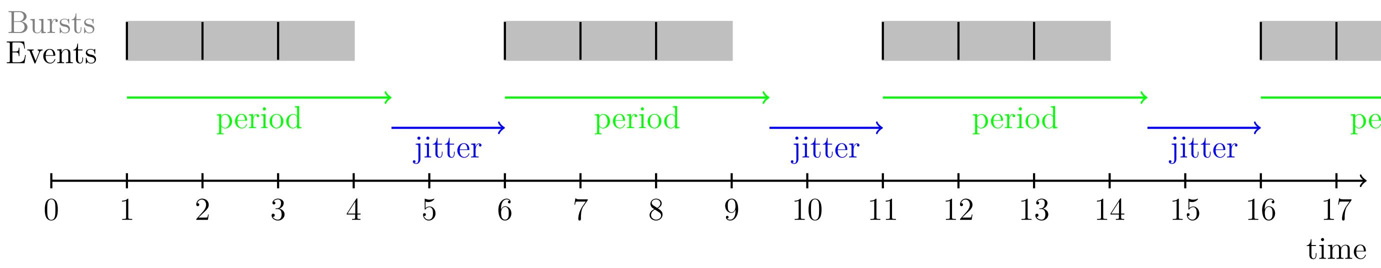

Figure 2.1.: BurstPatternEventTriggering Period-Jitter **accumulating**

- ***minNumberOfOccurrences*** (positive integer)
  Minimal number of events per burst (optional)
- ***minimumInterArrivalTime*** (time value)
  Minimal distance between subsequent events
- ***patternLength*** (time value)
  Length of each burst
- ***patternPeriod*** (time value)
  Time distance between the starting points of subsequent burst(optional)
- ***patternJitter*** (time value)
  Maximal allowed deviation from the periodic pattern (optional)

As example, we set:

- $maxNumberOfOccurrences = 3$
- $minNumberOfOccurrences = 1$
- $minimumInterArrivalTime = 1$
- $patternLength = 3$
- $patternPeriod = 3.5$
- $patternJitter = 1.5$

The combination of *patternPeriod* and *patternJitter* can be interpreted in an accumulating as seen in 2.1 or non-accumulating way as seen in 2.2 way. In the accumulating interpretation, the reference for the periodic occurrences is only the start point of the previous burst. In the non-accumulating way, there is an global reference point for the periodic repetitions.

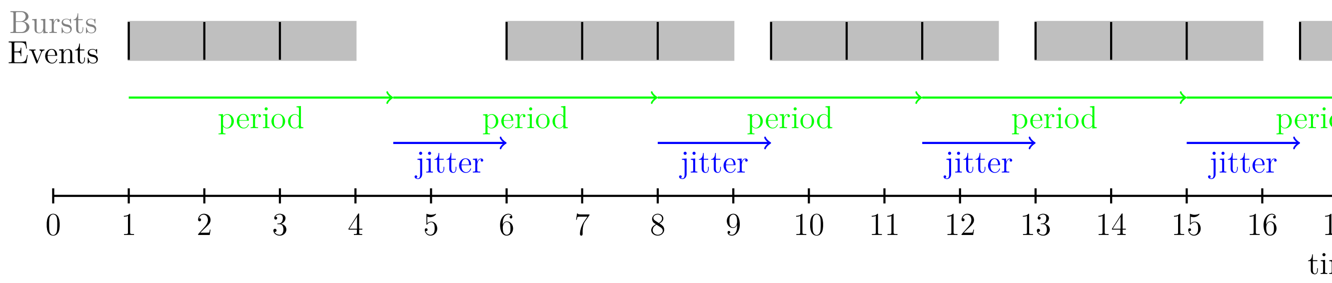

Figure 2.2.: BurstPatternEventTriggering Period-Jitter **non-accumulating**

With the definition of *patternLength* ("time distance between the beginnings of subsequent repetitions of the given burst pattern") you would think, that the accumulating variant is meant. Against that, the period attribute in *PeriodicEventTriggering*-Constraint is defined as "distance between subsequent occurrences of the event" in the text, hence it is also understandable the accumulating way, but there is the formal definition

$$\exists t_{reference} \forall t_n : t_{reference} + (n+1) * period \leq t_n \leq t_{reference} + (n-1) * period + jitter,$$

where t_n is the time of the n-th Event and $t_{reference}$ is a reference point, from which the periodic pattern starts, so the *PeriodicEventTriggering*-Constraint is meant to be understood in the non-accumulating way. It remains unclear, in which way the *BurstPatternEventTriggering* is meant to be understood.

Another problem of the AUTOSAR Timing Extensions is, that they were made for design purposes, monitoring them can be difficult, as they may need continuously growing time and memory resources, which makes online monitoring unsuitable in nearly all scenarios (more on monitorability in 3). As example, we will use the burst pattern again, this time using the attributes

- $maxNumberOfOccurrences = INT\_MAX$
- $minNumberOfOccurrences = 1$
- $minimumInterArrivalTime = 0$
- $patternLength = 3$
- $patternPeriod$ unused
- $patternJitter$ unused

Figure 2.3 shows the application of the *BurstPatternEventTriggering* constraint with the given parameters on a stream with events at the timestamps 3, 3.5, 4, 4.5. The development of possible the burst cluster with ongoing time is visualized. The gray

bars show, where the burst cluster can lay, the black lines show, where they definitely are. In timestamp 3 with only one event so far, only one burst has to be considered and it can lay between timestamp 0 and 6, the only limitation is, that it must include timestamp 3 with the event in that point. In Timestamp 3.5, there are two events (at 3 and 3.5) so far and there are two possibilities for burst placements. The first possibility with only one burst with both events in it, and the second possibility, where the events are in different bursts. The third graphic shows the trace in timestamp 4 with three different events so far (3, 3.5, 4) and three different possibilities for burst placements to consider. One possible burst contains all three events, the second possibility has one burst with the event at timestamp 3 and one burst with the events at 3.5 and 4 and the third possibility has one Burst with the events at 3 and 3.5 and one burst with the event at 4. The possible bursts in graphic 4 are analog to the third graphic, one possibility with one burst containing all 4 events and 3 possibilities with the first burst containing the first event, the first and second event or the first, the second and the third event and the second burst containing the remaining events.
In this Example, we see, that it is possible to create an unlimited number of possibilities for burst placements within one burst length, when the *minimumInterArrivalTime*-attribute is 0, which results in an infeasible resource consumption, as unlimited memory and time is needed to check the constraint in following events. Therefore, online monitoring this constraint is unsuitable in most cases.

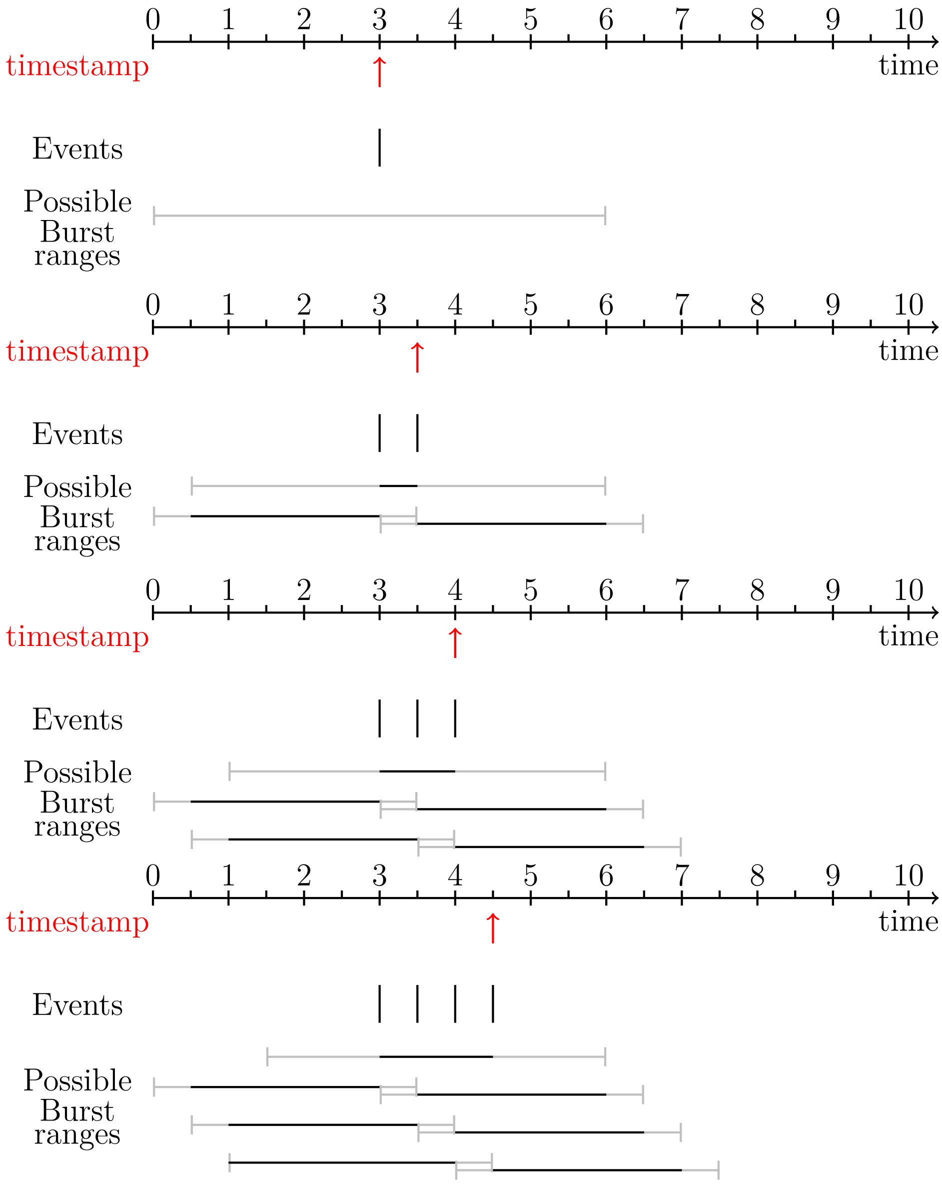

Figure 2.3.: BurstPatternEventTriggering Possible bursts, ↑ shows the current time

2.2. Timing Augmented Description Language

As timing extension to EAST-ADL(**E**lectronics **A**rchitecture and **S**oftware **T**echnology-**A**rchitecture **D**escription **L**anguage), the TIMMO (**Tim**ing **Mo**del) project, and its successor TIMMO2USE, were initiated. A part of this project was the **T**iming **A**ugmented **D**escription **L**anguage V2 (TADL2), were created. TADL2 has similar goals as AUTOSAR, but the definitions are written in a more formalized fashion. The definitions of the AUTOSAR Timing Extensions are only textually described often, the TADL2-Definitions are defined in a more formal way, as they offer a formal definition of each constraint in a timing constraint logic [BFL+12]. EAST-ADL is much less used in the automotive industry, but the EAST-ADL Timing Constraints are partly compatible to the AUTOSAR Timing Extensions, as they are sub- or supersets of each other. Many of the AUTOSAR Timing Extensions can be defined via a combination of TADL2 Constraints, as explained in section 2.2.3.
The timing constraints are defined on events or event chains, similar to the AUTOSAR Timing Extensions. In TADL2, all events of an event chain have a color attribute, which shows the logical connection of these events. This attribute is defined as abstract and possibly infinite datatype. The only restriction is, that an equality test on these color values must exist. TADL2 offers 18 timing constraints, which will briefly explained in the following:

- The **StrongDelayConstraint** defines the minimal and maximal time distance of the events from two event sets (*source* and *target*).
- The **DelayConstraint** is a less strict variant of the **StrongDelayConstraint**, because it allows additional events in *target*.
- The **RepeatConstraint**, **RepetitionConstraint**, **PeriodicConstraint**, **SporadicConstraint** and **ArbitraryConstraint** are describing the time distance between subsequent events, whereby they are having small semantic differences. An exact distinction between these constraints will be given in section 2.2.2.
- The **SynchronizationConstraint** and **StrongSynchronizationConstraint** define groups of event sets, whose events occur in common time intervals. The SynchronizationConstraint allows more than one event of each group per interval, the StrongSynchronizationConstraint does not.
- The **ExecutionTimeConstraint** is used to set a minimum and a maximum for the runtime of a task, not considering interruptions in the execution.
- The **OrderConstraint** defines that the n^{th} event of one event set must occur before or at the n^{th} event of a second event set.
- The **ComparisonConstraint** is used to describe ordering relations of timestamps.

- The **PatternConstraint** defines the time distance between periodic points in time to several events.
- The **BurstConstraint** regulates the maximum number of events in time intervals of a specific length.
- The **ReactionConstraint** describes the minimal and maximal time a response event must occur after the associated stimulus event. Additional response events are allowed, additional stimulus events not.
- The **AgeConstraint** is similar to the ReactionConstraint, but it is defined the other way around. Therefore, it describes the minimal and maximal time a stimulus event must occur before the associated response event. Additional stimulus events are allowed, additional response events not.
- The **OutputSynchronizationConstraint** is used to describe groups of event chains, which all have the same response events. The response events of the event chain must occur in common time intervals, like in the SynchronizationConstraint. In the **InputSynchronizationConstraint**, the roles of the stimulus and response events are swapped.

2.2.1. Parenthesis - Simple and Flexible Timing Constraint Logic

The formal definition of the TADL2 timing constraint are written in *Timing Constraint Logic* (short: *TiCL*), which was developed as part of the TIMMO-2-USE project. TiCL was formally introduced in [LN12], for better understanding the key aspects of this article will be explained in the following.
The main goal of TiCL is to be formal and expandable and offering the possibility of defining finite and infinite behaviors of events. In TiCL, only points in time, when events occur, are considered, therefore an events only consists of a real number as timestamp, without the possibility of adding a data value. There are 7 syntactic categories in TiCL

$$\begin{aligned}
\mathbb{R}&(\text{arithmetic constants})\\
Avar&(\text{arithmetic variables})\\
AExp&(\text{arithmetic expressions})\\[1ex]
Svar&(\text{set variables})\\
SExp&(\text{set expressions})\\[1ex]
TVar&(\text{time variables})\\
CExp&(\text{constraint expressions})
\end{aligned}$$

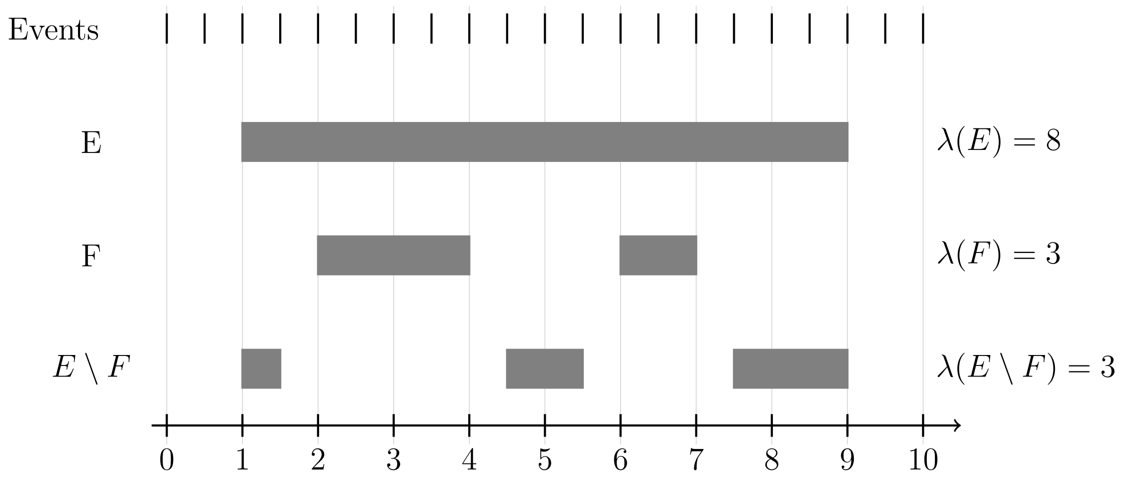

Figure 2.4.: Graphical example of $\lambda(E), \lambda(F)$ and $\lambda(E \setminus F)$

Arithmetic expressions can be defined as arithmetic constants, arithmetic variables, application of $+, -, *, /$ on arithmetic expressions, application of the cardinality operator on a set ($|E|$, $E \in SExp$) or as measure $\lambda(E)$ ($E \in SExp$). $\lambda(E)$ is defined as Lebesgue measure, which is figuratively speaking, the length of all continuous intervals of E. In figure 2.4 an example of the measure operator λ is visualized. The set E contains all Events between the timestamps 1 and 9, the set F contains the events at the timestamps between 2 and 4 and 6 and 7, therefore $E \setminus F$ contains the events at the timestamps $\{1, 1.5, 4.5, 5, 5.5, 7.5, 8, 8.5, 9\}$. E consists of one continuous interval from timestamp 1 to 9 with the length of 8, F consists of two continuous intervals from 2 to 4 with the length of 2 and from 6 to 7 with the length of 1, therefore $\lambda(F) = 3$. $E \setminus F$ consists of three continuous intervals, the first from 1 to 1.5 (length $= 0.5$), the second from 4.5 to 5.5 (length $= 1$) and the last from 7.5 to 9 (length $= 1.5$), so the total length of the continuous intervals of $E \setminus F$ is 3.

Set expressions can be defined as set variables, or as set of time variables that fulfill a given constraint expression.
Constraint expressions can be defined as application of the $\leq$-operator on time or arithmetic expressions, the $\in$ operator on time variables and set expressions, the logical conjunction on constraint expressions, the negation of constraint expressions and the $\forall$-Quantifier on arithmetic, set and time variables over an constraint expression.
As extension to this definition, well known syntactic abbreviations like $true \equiv 0 \leq 1$ or the $\exists$-quantifier will be used, but there are also some TiCL-specific syntactic abbreviations, like interval constructors, which will be defined and explained in the following.

Interval Constructors

Let $x, y \in Tvar$ and $E, F \in SExp$.
The constructor $[x \leq]([x <])$ is defined as $\{y : x \leq y\}(\{y : x < y\})$, therefore the interval contains all points in time laying behind of x, possibly containing x.
$[\leq x]([< x])$ is defined as complement of $[x <]([x \leq])$ and contains all timestamps laying before x.
$[x..y]$ is defined as $[x \leq] \cap [< y]$, so all points of time after x and before y, including x but not y, are part of this interval.
$[E \leq]$ is defined as $\{y : \exists x \in E : x \leq y\}$, this interval contains all point of time at and after the first timestamp in E. $[E <]$ is equal to $\{y : \forall x \in E : x < y\}$, therefore it defines the interval containing all timestamps after the latest point of time in E.
$[\leq E]$ ($[< E]$) is defined as $[E <]^C$ ($[E \leq]^C$), analogous to the operators on time variables.
$[E]$ is equal to $[E \leq] \cap [\leq E]$. It defines the time interval between the first and last element of E, including these points in time.
$E_{x<}(E_{<x})$ is defined as $E \cap [x <](E \cap [< x])$. This operators filters the timestamps in E so that only the points in time before (after) remain.
$[x..E]$ equals $[x \leq] \cap [< (E_{x<})]$. The interval begins at x and ends right before the first element of E after x.
$[E..F]$ is defined as $\{x : \exists y \in E : x \in [y..F]\}$ and describes the intervals, where the previous operator is applied on every element of E.

2.2.2. TADL2-Timing Constraints

For better understanding of the following chapters, the TADL Constraints will be presented next. As abbreviation and unification, all timing expressions are defined as set $\mathbb{T}$, which are understood as real numbers but expanded with ∞ and $-\infty$ in this chapter, but other value ranges for time expressions are possible and will be used in other parts of this thesis.
We define an event as a time value, possibly combined with an data value. The range of the data values are arbitrary, infinite data types are possible, also as empty data types, when only the point in time is relevant for the event. All TADL constraints are defined with attributes, which can be events, timing or arithmetic expressions or sets of them. Also, *EventChains* can be used as attributes. An *EventChain* consists of two sets of events (*stimulus* and *response*), which are causally related. All events in an *EventChain* must have a color value in their data field. This color possibly has an infinite type and an equality check on this must be defined. It is used to check, which events of an *EventChain* are directly related.

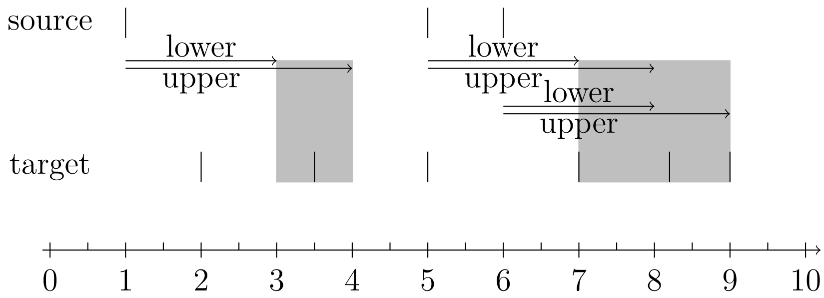

Figure 2.5.: Example DelayConstraint - $lower = 2$, $upper = 3$

DelayConstraint

The *DelayConstraint* has 4 attributes

source	event set
target	event set
lower	$\mathbb{T}$ (time expression)
upper	$\mathbb{T}$

and is defined as

$\forall x \in source : \exists y \in target : lower \leq y - x \leq upper.$

For all events x in *source*, there must be an y event in *target*, so that y lays between *lower* and *upper* after x. Note, that *lower* and *upper* can have negative values and that additional events in *target*, without an associated *source* event are allowed.
Figure 2.5 shows a visualized example of the *DelayConstraint* with the attributes $lower = 2$, $upper = 3$, $source = \{1, 5, 6\}$ and $target = \{2, 3.5, 5, 7, 8.2, 9\}$. The first element of source at timestamp 1 results in a required event in target between the timestamp 3 and 4 that is fulfilled by the event at 3.5. The second event of source requires an target event between 7 and 8, fulfilled by the event at 7. The last event of source is satisfied by the target event at 8.2 and 9.

StrongDelayConstraint

The *StrongDelayConstraint* has 4 attributes

source	event set
target	event set
lower	$\mathbb{T}$

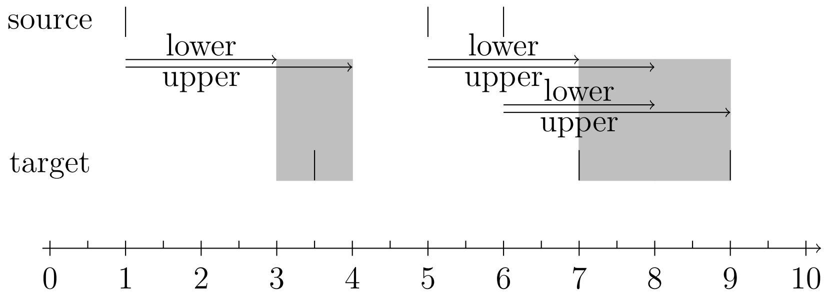

Figure 2.6.: Example StrongDelayConstraint - *lower* = 2, *upper* = 3

upper	$\mathbb{T}$

and is defined as

$|source| = |target| \wedge$
$\forall i : \forall x : x = source(i) \Rightarrow \exists y : y = target(i) \wedge lower \leq y - x \leq upper.$

The *StrongDelayConstraint* is a stricter version of the *DelayConstraint*, as it requires a bijective assignment between the source and target events, therefore additional events in target without matching source event are not allowed. Figure 2.6 shows an example of the *StrongDelayConstraint*. The example is the same as in the previous constraint, but without the additional target events at 2, 5 and 8.2.

RepeatConstraint

The *RepeatConstraint* also has 4 attributes

event	event set
lower	$\mathbb{T}$
upper	$\mathbb{T}$
span	*integer*

and is defined as

$\forall X \leq event : |X| = span + 1 \Rightarrow lower \leq \lambda([X]) \leq upper.$

As reminder, the $A \leq B$-operator over two sets of events A, B describes, that A is a sub-sequence of B, the $\lambda(A)$-function calculates the total length of all continuous intervals in A and the $[A]$ returns the time interval between the oldest and newest event in A.

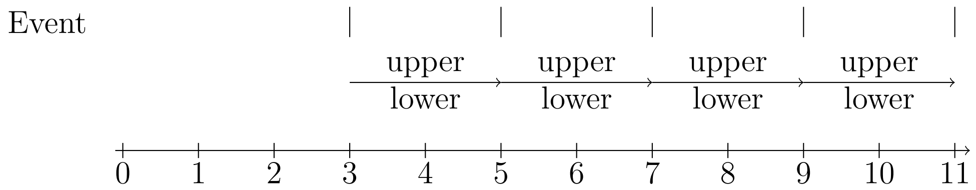

Figure 2.7.: Example RepeatConstraint - *lower* $= 2$, *upper* $= 2$, *span* $= 1$

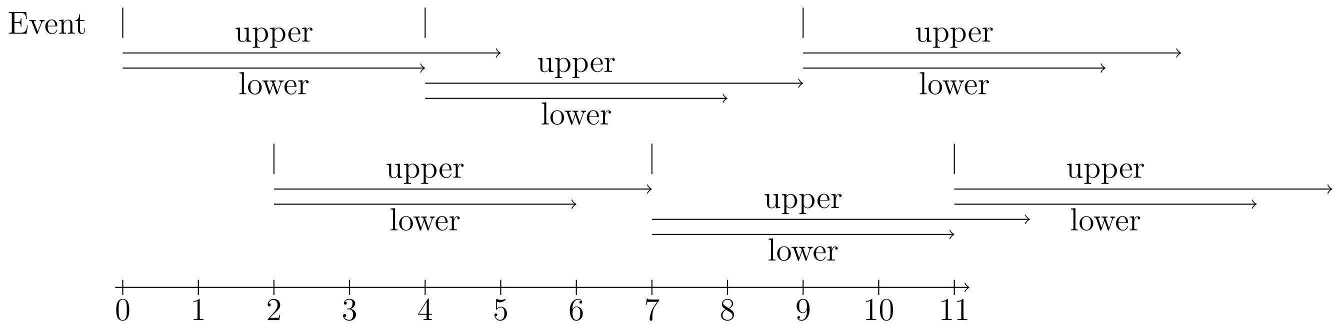

Figure 2.8.: Example RepeatConstraint - *lower* $= 4$, *upper* $= 5$, *span* $= 2$

The definition specifies that the length of each time interval containing $span + 1$ consecutively events must be between *upper* and *lower*.
The idea behind this constraint is to define repeated occurrences of events, with the possibility of overlapping, specified by the *span* attribute. After any event x, there are $span - 1$ events and than the next event must be between *lower* and *upper* after x.
Figure 2.7 shows an example of the RepeatConstraint with the attributes $event = \{3, 5, 8, ...\}$, $lower = upper = 2$ and $span = 1$. Because *lower* is equal *upper* and *span* is 1, the events are following a strictly periodic pattern after the first event. Figure 2.8 shows a more complex example with events at $\{0, 2, 4, 7, 9, 11, ...\}$, $lower = 4$, $upper = 5$ and $span = 2$. The *span*-attribute is 2, so the time distance between all subsequent events with an even index are considered, just like the subsequent events with an uneven index.

RepetitionConstraint

The *RepetitionConstraint* has 5 attributes

event	event set
lower	$\mathbb{T}$
upper	$\mathbb{T}$

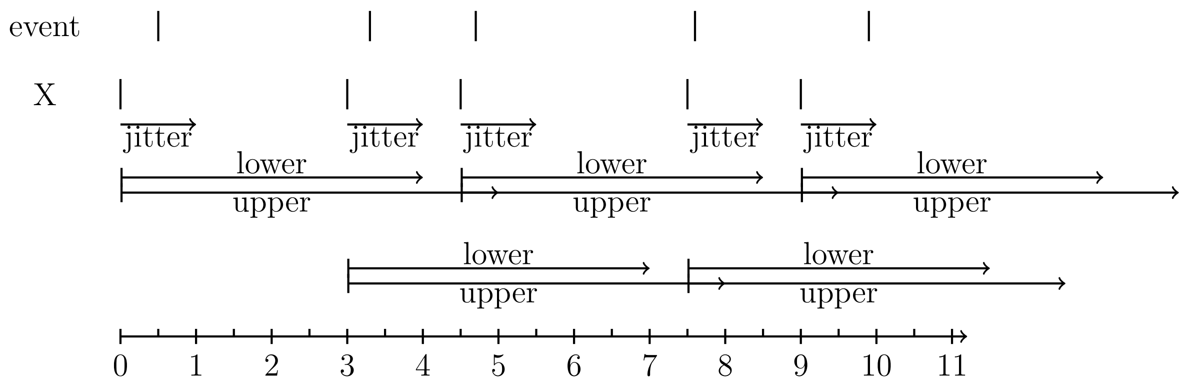

Figure 2.9.: Example RepetitionConstraint - *lower* = 4, *upper* = 5, *span* = 2, *jitter* = 1

span	*integer*
jitter	$\mathbb{T}$

and is defined via the *RepeatConstraint* and the *StrongDelayConstraint* as

$$\exists X : RepeatConstraint(X, lower, upper, span) \wedge \\ StrongDelayConstraint(X, event, 0, jitter)$$

where X is a set of arbitrary time stamps, that follow the structure of the *RepeatConstraint*(various(*span*) loose periodic repetitions). The actual points in time of *event* lay between the timestamps of X and *jitter* after that. For each point of time there is one, and only one, corresponding timestamp in X. Figure 2.9 shows an example of the *RepetitionConstraint* with the attributes *event* = $\{0.5, 3.3, 4.7, 7.6, 9.9, ...\}$, *lower* = 4, *upper* = 5, *span* = 2 and *jitter* = 1. The shown timestamps of X are only one possibility and may change due to later elements of *event*.

SynchronizationConstraint

The *SynchronizationConstraint* has 2 attributes

event	set of event sets, $\lvert event \rvert \geq 2$
tolerance	$\mathbb{T}$

and is defined via the *DelayConstraint* as

$$\exists X : \forall i : DelayConstraint(X, event_i, 0, tolerance) \wedge \\ DelayConstraint(event_i, X, -tolerance, 0)$$

X is a set of arbitrary point in time and there must be at least one timestamp in

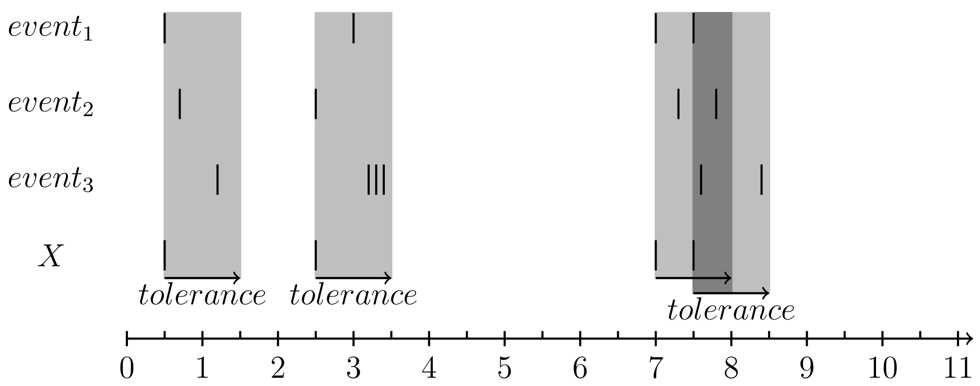

Figure 2.10.: Example SynchronizationConstraint - *tolerance* = 1

each set of *event*, that is between an element of X and *tolerance* after that. Also, for each element in any set of *event*, there must be a matching element of X.
In figure 2.10 is an example of the *SynchronizationConstraint* with the attributes $event = \{\{0.5, 3, 7, 7.5\}, \{0.7, 2.5, 7.3, 7.8\}, \{1.2, 3.2, 3.3, 3.4, 7.6, 8.4\}\}$ and $tolerance = 1$. The first points in time of each element of event form the first cluster, the corresponding element of X can be between 0.2 and 0.5. For simplification, only the latest possible value for the element of X are shown, which is the first event of the synchronization cluster. In the second cluster of events it can bee seen that multiple timestamps from one element of *event* can be associated with a single element of X. The third and fourth cluster show, that overlapping is also possible.

StrongSynchronizationConstraint

The *StrongSynchronizationConstraint* has the same two attributes as the *SynchronizationConstraint*

event	set of event sets, $\|event\| \geq 2$
tolerance	$\mathbb{T}$

and is defined as

$\exists X : \forall i : StrongDelayConstraint(X, event_i, 0, tolerance)$

The *StrongSynchronizationConstraint* is a stricter variant of the *Synchronization-Constraint*, as it requires a bijective assignment between the elements of X to one element of each set of *event*. For every $x \in X$, only one corresponding timestamp per set in *event* is allowed, like seen in figure 2.11, which shows the same example as the one for the *SynchronizationConstraint*, but the excess time stamps at 3.2 and 3.3 have been removed.

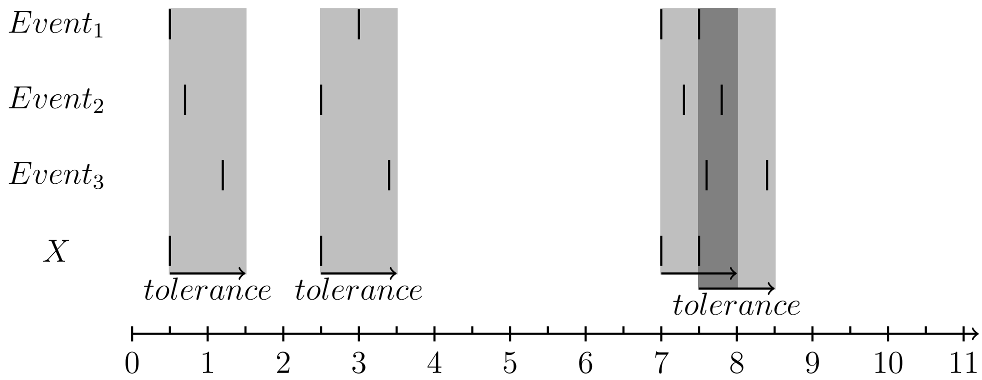

Figure 2.11.: Example StrongSynchronizationConstraint - $tolerance = 1$

ExecutionTimeConstraint

The *ExecutionTimeConstraints* takes 6 attributes

start	set of events
stop	set of events
preempt	set of events
resume	set of events
lower	$\mathbb{T}$
upper	$\mathbb{T}$

and is defined as

$\forall x \in start : lower \leq \lambda([x..stop] \setminus [preempt..resume]) \leq upper$

The interval constructor $\forall x \in start : [x..stop]$ defines the time interval between each point in time of *start* until the next element of *stop*, excluding the *stop* timestamp. $[preempt..resume]$, which is removed from the considered interval length, defines the intervals between each element of preempt until the next timestamp of resume. The Idea behind this constraint is to test the run time of a task, without counting interruptions.
Figure 2.12 shows an example of the *ExecutionTimeConstraints* with $start = \{1\}$, $end = \{7\}$, $preempt = \{2, 5\}$ and $resume = \{3, 6.5\}$. Therefore, $[start..end]$ spans the interval from time 1 to 7 with the length of 6 and $[preempt..resume]$ spans two intervals, 2 to 3 and 5 to 6.5 with the length 1 and 1.5. As result, $\lambda([x..stop] \setminus [preempt..resume])$ for $x = 1$ is 3.5 and the constraint is fulfilled, if, and only if, lower is equal or *lower* than 3.5 and *upper* is greater than that.

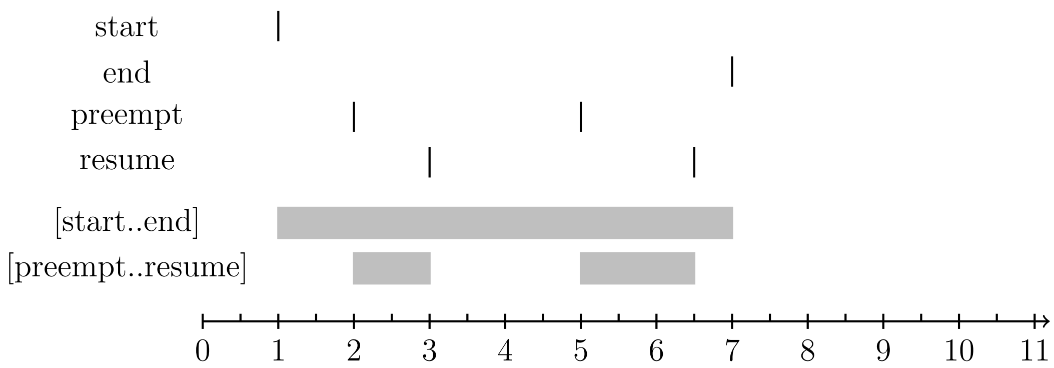

Figure 2.12.: Example ExecutionTimeConstraint

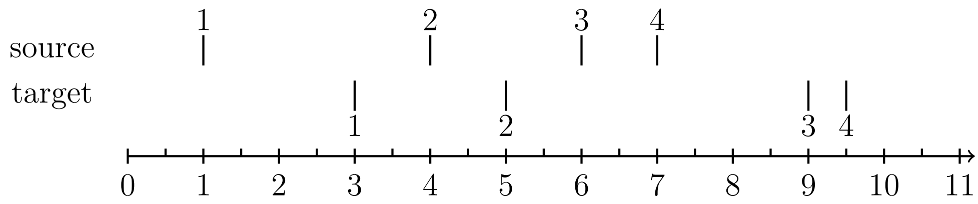

Figure 2.13.: Example OrderConstraint

OrderConstraint

The *OrderConstraint* takes two attributes

source	set of events
target	set of events

and is defined as

$$|source| = |target| \wedge \forall i : \exists x : x = source(i) \Rightarrow \exists y : y = target(i) \wedge < x \leq y$$

This constraints ensures the order of events, so that the i-th event of *target* is after the i-th event of *source*. Also, the number of events in *source* and *target* must be equal.

Figure 2.13 visualizes an example of the *OrderConstraint* with $source = \{1, 4, 6, 7\}$ and $target = \{3, 5, 9, 9.5\}$. The constraint is fulfilled, because the number of elements is equal and each i-th timestamp in *target* is later that the i-th timestamp of *source*.

ComparisonConstraint

The *ComparisonConstraint* is significant different to all previous and following constraints, as it does not describe the behavior of events and only compares two time expressions. It takes 3 attributes

leftOperand	$\mathbb{T}$
rightOperand	$\mathbb{T}$
operator	comparisonOperator($\in \{LessThanOrEqual, LessThan, GreaterThanOrEqual, GreaterThan, Equal\}$)

The definition is pretty straight forward as it only applies the given operator to the operands:

$$ComparisonConstraint(leftOperand, rightOperand, LessThanOrEqual)$$
$$\Leftrightarrow leftOperand \leq rightOperand$$

$$ComparisonConstraint(leftOperand, rightOperand, LessThan)$$
$$\Leftrightarrow leftOperand < rightOperand$$

$$ComparisonConstraint(leftOperand, rightOperand, GreaterThanOrEqual)$$
$$\Leftrightarrow leftOperand \geq rightOperand$$

$$ComparisonConstraint(leftOperand, rightOperand, GreaterThan)$$
$$\Leftrightarrow leftOperand > rightOperand$$

$$ComparisonConstraint(leftOperand, rightOperand, Equal)$$
$$\Leftrightarrow leftOperand = rightOperand$$

Due to the simplicity of this constraint, no explicit example is given.

SporadicConstraint

The *SporadicConstraint* takes 5 attributes

event	set of events
lower	$\mathbb{T}$
upper	$\mathbb{T}$
jitter	$\mathbb{T}$
minimum	$\mathbb{T}$

and is defined as combination of the *RepetitionConstraint* and the *RepeatConstraint* as

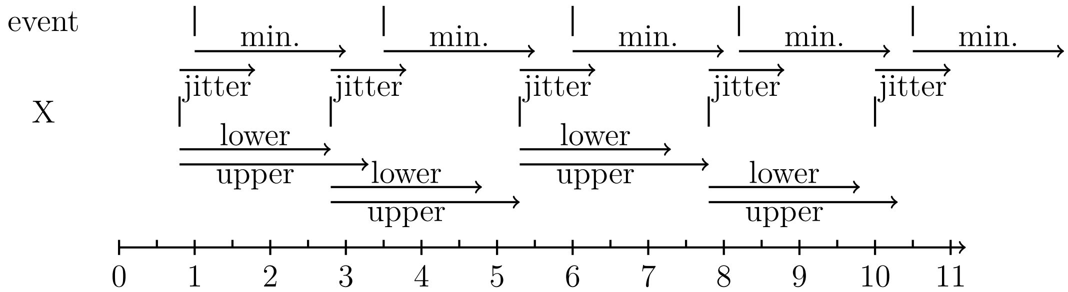

Figure 2.14.: Example SporadicConstraint - $lower = 2$, $upper = 2.5$, $jitter = 1$, $minimum = 2$

$RepetitionConstraint(event, lower, upper, 1, jitter)$
$\wedge\ RepeatConstraint(event, minimum, \infty, 1)$

The second part of the definition, using the *RepeatConstraint*, ensures that all events in *event* lay at least *minimum* apart. The application of the *RepetitionConstraint* generates a set of events X, that lay between *lower* and *upper* apart from each other. For each point in time in X, there must be exactly one timestamp in *event*, that is not before the corresponding element of X and not later than *jitter* after that.
Figure 2.14 shows a possible application of the *SporadicConstraint* with the attributes $lower = 2$, $upper = 2.5$, $jitter = 1$, $minimum = 2$ and $event = \{1, 3.5, 6, 8.2, 10.5, ...\}$. Like in the *RepetitionConstraint*, the exact position of the timestamps in X is variable and may need to be changed due to later entries in *event*.

PeriodicConstraint

The *PeriodicConstraint* takes 4 attribute

event	set of events
period	$\mathbb{T}$
jitter	$\mathbb{T}$
minimum	$\mathbb{T}$

and defines a specialized form of the *SporadicConstraint*

$SporadicConstraint(event, period, period, jitter, minimum)$

The variable timestamps in the set X are now following a strictly periodic pattern, where subsequent elements of this set lay exactly *period* apart. Each element of *event* lays between one element of X and *jitter* after that. Again, there must be

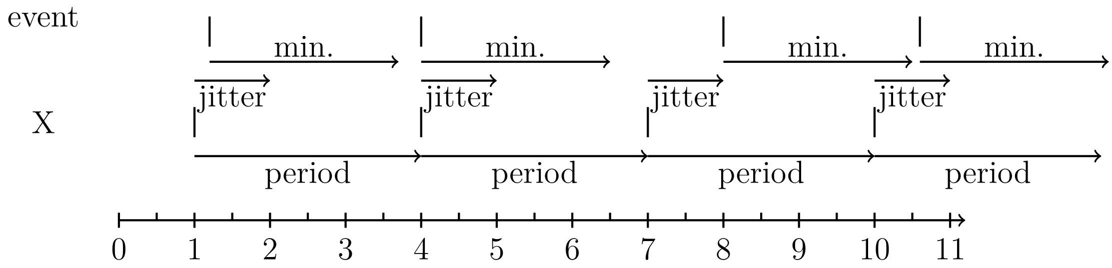

Figure 2.15.: Example PeriodicConstraint - $period = 3$, $jitter = 1$, $minimum = 2.5$

bijective mapping between the elements of *event* and X.
In figure 2.15, the *PeriodicConstraint* with the attributes $period = 3$, $jitter = 1$, $minimum = 2.5$ and $event = \{1.2, 4.0, 8, 10.6, ...\}$ is visualized. The timestamps of X lay exactly *period* apart and the *events* behind that in the previously described way. Also, the minimum time distance between all points of time in *event* is *minimum*.

PatternConstraint

The *PatternConstraint* takes 5 attributes

event	set of events
period	$\mathbb{T}$
offset	set of $\mathbb{T}$
jitter	$\mathbb{T}$
minimum	$\mathbb{T}$

and is defined as

$$\begin{aligned}
&\exists X : PeriodicConstraint(X, period, 0, 0)\\
&\quad \wedge \forall i : DelayContraint(X, event, offset_i, offset_i + jitter)\\
&\quad \wedge RepeatConstraint(event, minimum, \infty, 1)
\end{aligned}$$

This constraint can be understood as a modification of the *PeriodicConstraint*, as it describes periodic behavior, but not from single events, but from groups of $|offset|$ subsequent events, that follow specific time distances (specified by $offset$) after the strictly periodic timestamps of X.
There is a major weak spot in the definition of this constraint, because the set X can be set to the empty set. In this case, the part of the definition, which uses the *PeriodicConstraint* and the *DelayContraint*, are always satisfied, irrespective of the event in *event*. Therefore, the *PatternConstraint* only ensures the minimal distance

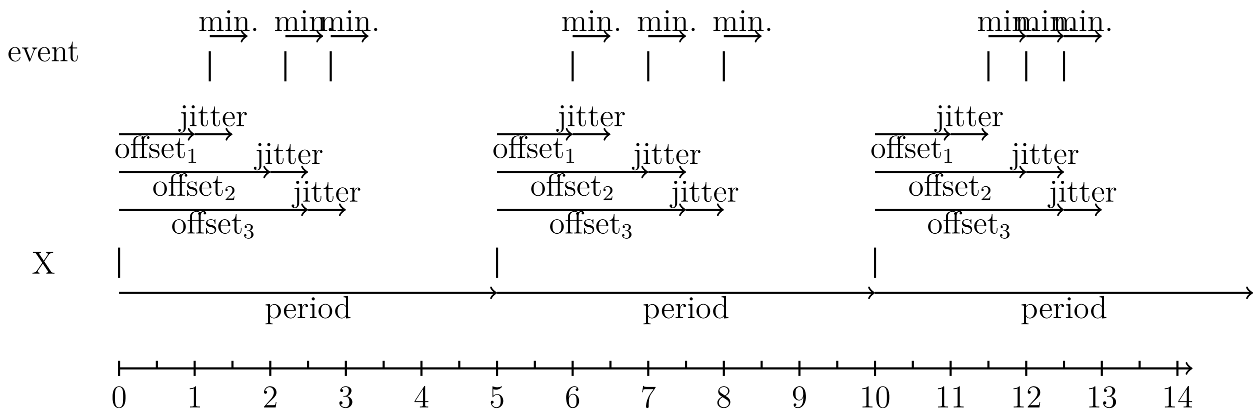

Figure 2.16.: Example PatternConstraint - $period = 5$, $offset = \{1, 2, 2.5\}$, $jitter = 0.5$, $minimum = 0.5$

between two events, what should not the purpose of this constraint. The obvious countermeasure to this problem would be to restrict X in a way that ensures that it is not empty and the first element of X must lay before the first *event* occurrence. The textual description of the constraint, which says literally the "PatternConstraint requires the constrained event occurrences to appear at a predetermined series of offsets from a sequence of reference points" contradicts this countermeasure, because the *DelayConstraint* allows additional events in the *target* events with no matching *source* event. Therefore, any event occurrences additionally to the events following the offset scheme, would be allowed, which conflicts with the citation. Because of this problem, the *PatternConstraint* is redefined as

$$\begin{aligned}
&\exists X : PeriodicConstraint(X, period, 0, 0)\\
&\quad \wedge \forall i : \boldsymbol{Strong}DelayContraint(X, event, offset_i, offset_i + jitter)\\
&\quad \wedge RepeatConstraint(event, minimum, \infty, 1)
\end{aligned}$$

for the scope of this thesis. The use of the *StrongDelayConstraint*, instead of the *DelayConstraint*, ensures that each event occurrence is following the time distances defined by the offsets. This notion of the *PatternConstraint* is also carried by the described relations between the TADL2 timing constraints and the AUTOSAR Timing Extensions, which were done as part of the development of TADL2. These descriptions equate the *PatternConstraint* and AUTOSARs *ConcretePatternEventTriggering*, which is clearly defined in the way of this redefinition.

Figure 2.16 shows an application of the *PeriodicConstraint* with attributes $period = 5$, $offset = \{1, 2, 2.5\}$, $jitter = 0.5$, $minimum = 0.5$ and $event = \{1.2, 2.2, 2.8, 6, 7, 8, 11.5, 12, 12.5, ...\}$. Like in the previous describes constraint, the exact position of all points in time of X may change due to later timestamps of *event*.

	1	2	3	5	8	10
1	0	**1**	**2**	**4**	7	9
2		0	**1**	**3**	**6**	8
3			0	**2**	**5**	**7**
5				0	**3**	**5**
8					0	**2**
10						0

Table 2.1.: Time distances as seen in figure 2.17

ArbitraryConstraint

The *ArbitraryConstraint* takes 3 attributes

event	set of events
minimum	set of $\mathbb{T}$
maximum	set of $\mathbb{T}$

where $|minimum| = |maximum|$. It is defined as

$\forall i : RepeatConstraint(event, minimum_i, maximum_i, i)$

The Idea behind the *ArbitraryConstraint* is to describe the time distance between each event and several following events. The first entry of *minimum* and *maximum* define the distance between every event and it direct successor. The second entries, where the *span* attribute of the *RepeatConstraint* is 2, set the distance between one event and its next but one successor and so on.
Figure 2.17 shows an example of the *ArbitraryConstraint* with the attributes $minimum = \{1, 2, 3\}$, $maximum = \{5, 6, 7\}$ and $event = \{1, 2, 3, 5, 8, 10, ...\}$. The time distances between subsequent events with 0, 1 and 2 skipped events are shown in table 2.1, the relevant distances are written in **bold** font. Apparently, the time distances are matching the ranges, given by the *minimum*- and *maximum* attribute.

BurstConstraint

The *BurstConstraint* takes 4 attributes

event	set of events
length	$\mathbb{T}$
maxOccurrences	*integer*

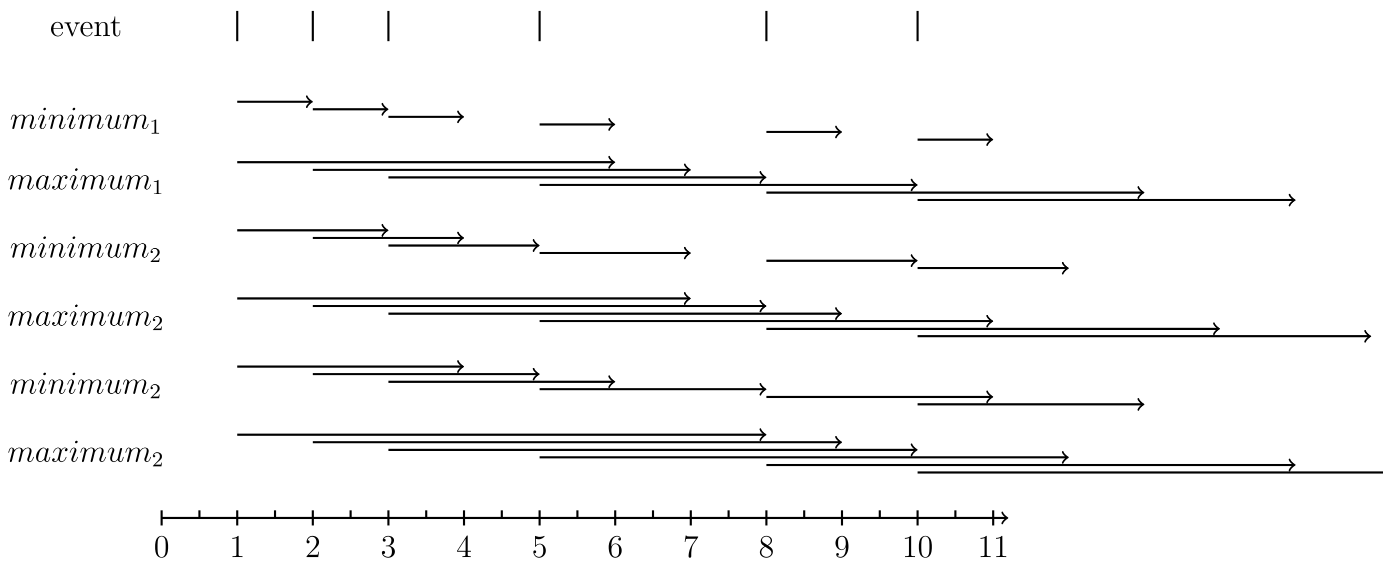

Figure 2.17.: Example ArbitraryConstraint - $period = 5$, $offset = \{1, 2, 2.5\}$, $jitter = 0.5$, $minimum = 0.5$

$$minimum \quad \mathbb{T}$$

and is defined as

$$\begin{aligned}&RepeatConstraint(event, length, \infty, maxOccurrences)\\ &\quad \wedge RepeatConstraint(event, minimum, \infty, 1)\end{aligned}$$

The idea of this constraint is to describe the maximum number of events that may occur in a time interval of the given *length*. Additionally all subsequent event must be at least *minimum* apart. Therefore, the intuition is different to the AUTOSAR *BurstPatternEventTriggering*, where clusters of events are described. A complete comparison of these constraints will be done in section 2.2.3.
In figure 2.18 the *BurstConstraint* with the attributes $length = 5$, $maxOccurrences = 3$, $minimum = 0.8$ and $event = \{1, 2, 3, 7, 8, 9\}$ is visualized. In every interval of the length 5, there are three or less events, also all subsequent events lay at least 0.8 apart. Therefore, the constraint is fulfilled.

ReactionConstraint

The *ReactionConstraint* takes 3 attributes

$$\begin{aligned} scope &\quad EventChain\\ minimum &\quad \mathbb{T}\end{aligned}$$

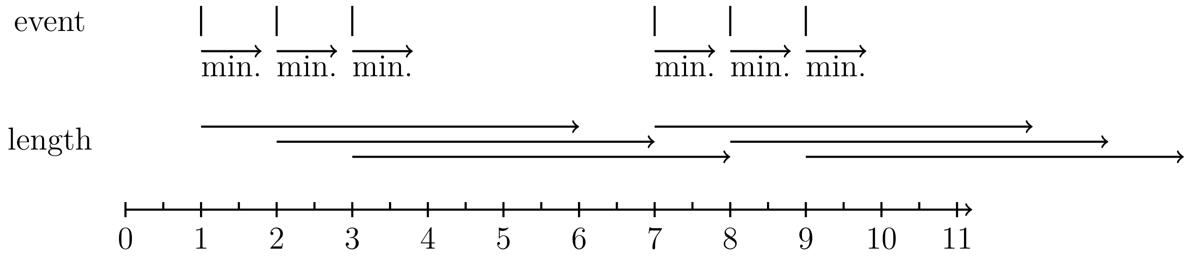

Figure 2.18.: Example BurstConstraint - *length* = 5, *maxOccurences* = 3 *minimum* = 0.8

$$maximum \quad \mathbb{T}$$

and is defined as

$$\begin{aligned}
&\forall x \in scope.stimulus : \exists y \in scope.response : \\
&\quad x.color = y.color \\
&\quad \wedge (\forall y' \in scope.response : y'.color = y.color \Rightarrow y \leq y') \\
&\quad \wedge minimum \leq y - x \leq maximum
\end{aligned}$$

The definition says, that after every event x of *scope.stimulus*, there is an event y in *scope.response* with the same color. The time distance between these events must be at least *minimum* and at most *maximum*. Additional events with the same color as y in *scope.response* are allowed, if they lay behind y. The definition implies, that additional events with other colors are allowed in *scope.response*, but not in *scope.stimulus* and every color is only allowed once in *scope.stimulus*.
A visualized example with the attributes $minimum = 1$, $maximum = 3$, $scope.stimulus = \{(1, red), (5, green), (5.5, purple), (8, orange)\}$ and $scope.response = \{(0.8, blue), (2.1, red), (4.5, blue), (6.6, purple), (6.7, purple), (9.5, purple), (7.5, green), (10, orange)\}$ can be seen in figure 2.19. The red *stimulus*-event is followed by the red *response*-event at 2.1, the green *stimulus* event at 5 by the *response* event at 7.5 and so on. The blue *response* events at 1 and 4.5 are additional events without an associated stimulus event. The purple events at 6.7 and 9.5 are the second and third event of this color in *scope.response* and therefore, their time distance to the *stimulus* event with the same color is irrelevant.

AgeConstraint

The *AgeConstraint* takes 3 attributes

$$\begin{aligned}
scope &\quad EventChain \\
minimum &\quad \mathbb{T}
\end{aligned}$$

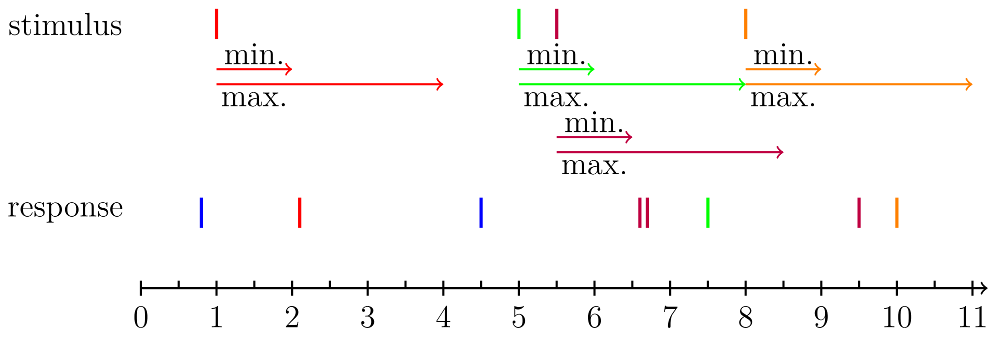

Figure 2.19.: Example ReactionConstraint - $minimum = 1$, $maximum = 3$

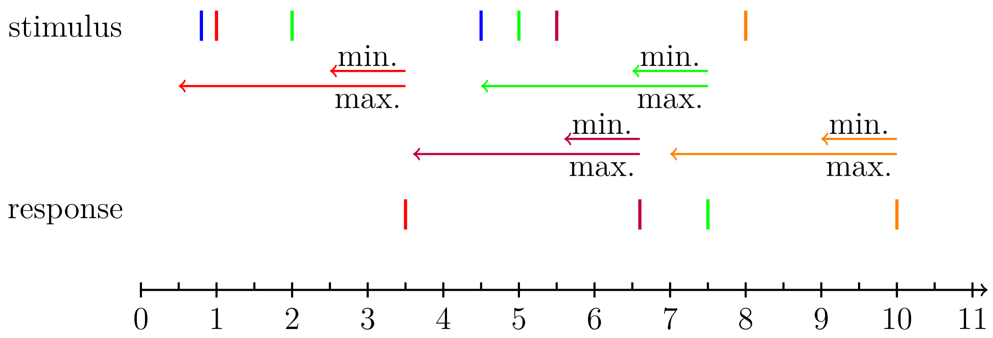

Figure 2.20.: Example AgeConstraint - $minimum = 1$, $maximum = 3$

$maximum \quad \mathbb{T}$

and is defined as

$$\begin{aligned}
&\forall y \in scope.response : \exists x \in scope.stimulus : \\
&\quad x.color = y.color \\
&\quad \wedge (\forall x' \in scope.stimulus : x'.color = x.color \Rightarrow x' \leq x) \\
&\quad \wedge\, minimum \leq y - x \leq maximum
\end{aligned}$$

The *AgeConstraint* is a turned around counterpart to the *ReactionConstraint*. For every event of *scope.response*, there must be an event with the same color in *scope.stimulus*, that is between *minimum* and *maximum* older than the *response* event. Additional events are only allowed in *scope.stimulus*, and only before the event that matches with a *response* event.
Figure 2.20 shows an application of the *AgeConstraint* with the attributes $minimum = 1$, $maximum = 3$, $scope.stimulus = \{(0.8, blue), (1, red), (2, green), (4.5, green), (5, green), (5.5, purple), (8, orange)\}$ and $scope.response = \{(3.5, red), (7.5, green), (6.6, purple), (10, orange)\}$. The blue timestamps are additional events without matching events in *scope.response*.

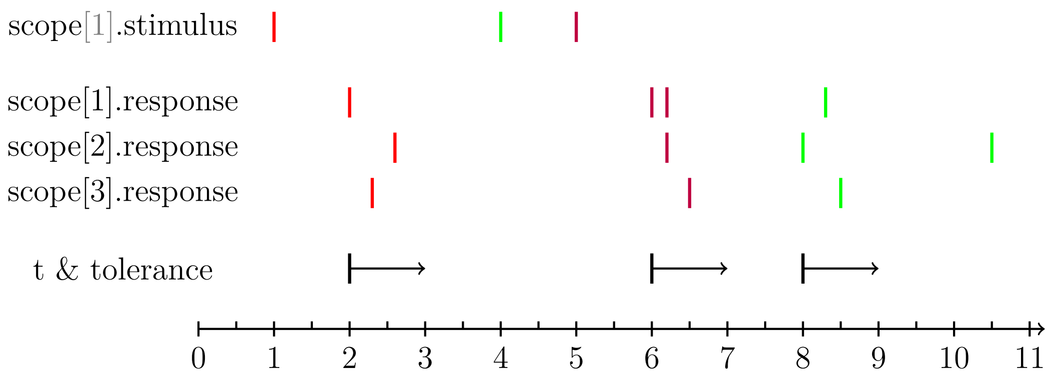

Figure 2.21.: Example OutputSynchronizationConstraint - $tolerance = 1$

OutputSynchronizationConstraint

The *OutputSynchronizationConstraint* takes 2 attributes

scope	Set of *EventChain*
tolerance	$\mathbb{T}$

where all elements of *scope* have the same *stimulus*. It is defined as

$$\begin{aligned}&\forall x \in scope_1.stimulus : \exists t : \forall i : \exists y \in scope_i.response :\\ &\quad x.color = y.color\\ &\quad \wedge (\forall y' \in scope_i.response : y'.color = y.color \Rightarrow y \leq y')\\ &\quad \wedge 0 \leq y - t \leq tolerance\end{aligned}$$

The definition says, that after each event x in $scope_1.stimulus$, there must be a interval with the length of *tolerance*, in which every $scope_i.response$ must have an event y with the same color as x. Additional response events with this color are only allowed after y. Figure 2.21 shows an example of the *OutputSynchronization-Constraint* with the attributes $tolerance = 1$,
$scope[1].stimulus = scope[2].stimulus = scope[3].stimulus = \{(1, red), (4, green), (5, purple)\}$,
$scope[1].response = \{(2, red), (6, purple), (6.2, purple), (8.2, green)\}$,
$scope[2].response = \{(2.6, red), (6.2, purple), (8, green), (10.5, green)\}$,
$scope[3].response = \{(2.3, red), (6.5, purple), (8.5, green)\}$.

InputSynchronizationConstraint

The *InputSynchronizationConstraint* takes 2 attributes

scope	Set of *EventChain*

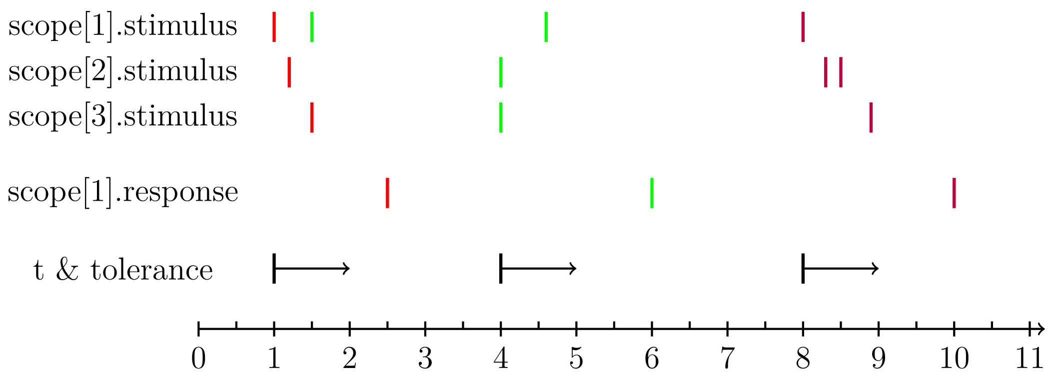

Figure 2.22.: Example InputSynchronizationConstraint - $tolerance = 1$

$tolerance \quad \mathbb{T}$

where all elements of *scope* have the same *response*. It is defined as

$$
\begin{aligned}
&\forall y \in scope_1.response : \exists t : \forall i : \exists x \in scope_i.stimulus : \\
&\quad x.color = y.color \\
&\quad \wedge (\forall x' \in scope_i.stimulus : x'.color = x.color \Rightarrow x \leq x') \\
&\quad \wedge\, 0 \leq x - t \leq tolerance
\end{aligned}
$$

The *InputSynchronizationConstraint* is a counterpart of the *OutputSynchronization-Constraint*, as the *stimulus* events must be synchronized, not the *response* events. Figure 2.22 contains an example of the *InputSynchronizationConstraint* with the attributes $tolerance = 1$
$scope[1].stimulus = \{(1, red), (1.5, green), (4.6, green), (8, purple)\}$
$scope[2].stimulus = \{(1.2, red), (4, green), (8.3, purple), (8.5, purple)\}$
$scope[3].stimulus = \{(1.5, red), (4, green), (8.9, purple)\}$
$scope[1].response = scope[2].response = scope[3].response = \{(2.5, red), (6, green), (10, purple)\}$

2.2.3. Comparison TADL2 - AUTOSAR Timing Extension

As said before, the *TADL2 Timing Constraints* and the *AUTOSAR Timing Extension* are compatible in parts and many of the *AUTOSAR Timing Extension* can be expressed as equivalent combinations of the *TADL2 Timing Constraints*. In [BFL+12], the relation between these constraints is shown, but this comparison is based on an outdated version of the AUTOSAR Timing Extensions and some of the constraints have been updated, therefore each of the *AUTOSAR Timing Extensions* will be listed in this chapter and it will be explained, if and how they can be expressed using *TADL2 Timing Constraints*.

The types used in the AUTOSAR Timing Extension are similar to the ones in TADL2. TADL2 *Events* are called *TimingDescriptionEvent* in AUTOSAR, the same goes for *EventChains*, which are called *TimingDescriptionEventChains*. A larger difference can be seen in the definition of time. While TADL2 defines time as real numbers, the time definition used in the AUTOSAR Timing Extension can also be multidimensional, for example when the real time and the angle of the crankshaft is regarded. For simplification, all timestamps are considered as real numbers in the following, but an extension to multidimensional time stamps is possible, as AUTOSAR requires a strict order between all time stamps. *Executable entities* as defined in the AUTOSAR Timing Extension describe things, that can be executed, for example a function. For the timing constraints, only striking point in times of these entities are relevant, for example the start or end points. It should be noted, that the set of TADL2 timing constraints are not equal to the AUTOSAR Timing Extension and that there are constraint, that cannot be expressed using the corresponding counterpart.

PeriodicEventTriggering

The *PeriodicEventTriggering* defined in AUTOSAR with the attributes $(event, period, jitter, minimumInterArrivalTime)$ is equivalent to the *TADL2 PeriodicConstraint* with the same attributes.

SporadicEventTriggering

AUTOSARs *SporadicEventTriggering* with the attributes $(event, jitter, maximumInterArrivalTime, minimumInterArrivalTime, period)$ is equivalent to the *TADL2 SporadicConstraint*, but the names of the attributes are different:
$lower = period$
$upper = maximumInterArrivalTime$
$jitter = jitter$
$minimum = minimumInterArrivalTime$

ConcretePatternEventTriggering

The idea behind the *ConcretePatternEventTriggering* from AUTOSAR is the same as behind TADL2s *PatternConstraint*, but subtleties are different. Both define a periodic behavior and offsets, that describe time distances between the periods and

the actual events. The main difference is the *jitter* attribute. In AUTOSARs *ConcretePatternEventTriggering*, the *patternJitter* attribute defines the allowed deviation of the start points of the periodic repetitions, as in TADL2 the *jitter* value describes the deviation between the offsets and the actual event.
The *ConcretePatternEventTriggering* from AUTOSAR additionally defines an *patternLength* attribute, which describes the length of the intervals, in which the clusters of events will occur. It is constrained by

$$0 \leq max(offset) \leq patternLength$$
$$\wedge\, patternLength + patternJitter < patternPeriod$$

The *patternLength* attribute can not be described with TADL2 timing constraints, as it would require to determine the distance of filtered events, which is not possible with the TADL2 constraints.
TADL2 defines the *minimum* attribute for the *PatternConstraint* that describes the minimal time distance between subsequent events. In AUTOSAR, this must be described by using the *ArbitraryEventTriggering*, where $minimumDistance_1$ is $minimum$ and $maximumDistance_1$ is ∞.

BurstPatternEventTriggering

The *BurstPatternEventTriggering* as defined in AUTOSAR and TADL2s *BurstConstraint* share the same target, as they define a maximum number of events that may occur in a specific time interval, but the *BurstPatternEventTriggering* is way more complex. Additionally to the attributes of TADL2s *BurstConstraint*, that define the *length* of the time interval, the *maxOccurrences* of the event in this interval and the minimal time between subsequent events, the *BurstPatternEventTriggering* allows to define the minimal number of events in the interval and periodic repetitions of the burst interval.
Every set of attributes fulfilling the TADL2 *BurstConstraint* fulfill the AUTOSAR *BurstPatternEventTriggering*, when the attributes are renamed to the AUTOSAR equivalents ($length \rightarrow patternLength$, $maxOccurences \rightarrow maxNumberOfOccurences$, $minimum \rightarrow minimumInterArrivalTime$). This does not work the other way around, even if the attributes, that exist in the *BurstPatternEventTriggering* and not in the *BurstConstraint* are unused. The reason for this is, that the observed interval must start at an event in the TADL2 *BurstConstraint*, in the *BurstPatternEventTriggering* those can start in any point of time.

ArbitraryEventTriggering

AUTOSARs *ArbitraryEventTriggering* is similar to the *ArbitraryConstraint* as defined in TADL2, but *ArbitraryEventTriggering* allows to set a list of *ConfidenceIntervals*, to describe the probability, how far the events may lay apart. These probabilities can not be expressed in TADL2.

LatencyTimingConstraint

The *LatencyTimingConstraint* of AUTOSAR takes 5 attributes, a latency type $latencyConstraintType \in \{age, reaction\}$, three time values *maximum*, *minimum* and *nominal* and an event chain *scope*, consisting of the stimulus and response events. The *nominal*-value is not relevant for a formal definition of the Constraint, therefore there is no counterpart to this in the TADL2 Constraints. If the $latencyConstraintType$ of the *LatencyTimingConstraint* is *age*, it can be expressed with $AgeConstraint$ defined in TADL2. The *LatencyTimingConstraint* with the $latencyConstraintType$ *reaction* is equivalent to the $reactionConstraint$.

AgeConstraint

The goal of the *AgeConstraint* in AUTOSAR is to define a minimal and maximal age of an event at the point in time, when it is processed. There is no counterpart to this in the TADL2 constraints, because the point in time, when the event is processed, is unknown. If this point in time is known, AUTOSARs *AgeConstraint* can be expressed using TADL2s *AgeConstraint*, but in that case, it could also be expressed using AUTOSARs *LatencyTimingConstraint*.

SynchronizationTimingConstraint

The *SynchronizationTimingConstraint* is similar to the *SynchronizationConstraint*, the *StrongSynchronizationConstraint*, the *OutputSynchronizationConstraint*, the *InputSynchronizationConstraint* or combinations of them, depending on the attributes. Table 2.2 shows, with which attributes the *SynchronizationTimingConstraint* is equivalent to which TADL2 Constraint(s).

eventOccurrence-Kind	scope/ scopeEvent	synchronization-ConstraintType	tolerance	TADL2 Constraints
multiple Occurrences	scopeEvent	*not set*	tolerance	SynchronizationConstraint (scopeEvent, tolerance)
single Occurrences	scopeEvent	*not set*	tolerance	StrongSynchronizationConstraint (scopeEvent, tolerance)
multiple Occurrences	scope	response Synchronization	tolerance	OutputSynchronizationConstraint (scope, tolerance) $\wedge$ SynchronizationConstraint (scope.response, tolerance)
single Occurrences	scope	response Synchronization	tolerance	OutputSynchronizationConstraint (scope, tolerance) $\wedge$ StrongSynchronizationConstraint (scope.response, tolerance)
multiple Occurrences	scope	stimulus Synchronization	tolerance	InputSynchronizationConstraint (scope, tolerance) $\wedge$ SynchronizationConstraint (scope.stimulus, tolerance)
single Occurrences	scope	stimulus Synchronization	tolerance	InputSynchronizationConstraint (scope, tolerance) $\wedge$ SynchronizationConstraint (scope.stimulus, tolerance)

Table 2.2.: SynchronizationTimingConstraint $\Leftrightarrow$ TADL2 Constraints

SynchronizationPointConstraint

The *SynchronizationPointConstraint* describes, that a list of executables and a set of events or executable entities, defined in *sourceEec* and *sourceEvent*, must finish and occur, before the executables and events in *targetEec* and *targetEvent* will start or occur. There is no counterpart to this in the TADL2 constraints.

OffsetTimingConstraint

The *OffsetTimingConstraint*, defined in AUTOSAR Timing Extensions, is semantically the same as the TADL2 *DelayConstraint*, just some attributes are named differently. The *maximum* attribute of the *OffsetTimingConstraint* is named *upper* and the *minimum* attribute *lower* in the *DelayConstraint*.

ExecutionOrderConstraint

The goal of *ExecutionOrderConstraint* of the AUTOSAR Timing Extensions is used to describe the order of events or the execution order of executable entities, defined as *orderedElement* attribute. There is no constraint in TADL2 that describes exactly this, but if the *ExecutionOrderConstraint* is used to describe only the order of events, it can be described as

$$OrderConstraint(orderedElement_1, orderedElement_2)$$
$$\wedge \ldots \wedge$$
$$OrderConstraint(orderedElement_{n-1}, orderedElement_n)$$

If the *ExecutionOrderConstraint* is used for executable entities, each executable entity must be turned into one or more events to be described via TADL2 Constraints, depending on the other attributes. For example, if the attribute *executionOrderConstraintType* is set to *ordinaryEOC*, the start and finish points of the entities define the observed events.

ExecutionTimeConstraint

The idea behind the *ExecutionTimeConstraint* is similar in AUTOSAR and TADL2. Both describe the minimal and maximal allowed run time of an executable entity, not counting interruptions. AUTOSARs *ExecutionTimeConstraint* is defined directly on an executable entity and the TADL2 constraint on events describing the *start*, *stop*, *preemption* and *resume* timestamps. Therefore the executable entity must be turned into these events to express the AUTOSAR *ExecutionTimeConstraint* via TADL2 constraints. The start and stop points of the executable must be turned into these events, the start and stop points of the interruptions must be turned into the events in the *preempt* and *resume* event sets. If external calls should be excluded from the run time, they must also be transferred into the *preempt* and *resume* event sets.

3. Monitoring Timing Constraints on possibly infinite Streams

The goal of this thesis is to implement online monitors for the TADL2 Timing Constraints on possibly infinite streams. An online monitor checks the current execution of a system, parallel to its execution. Because every computing system has finite memory resources and the online monitor should be able to process more events than occurs in the stream in a specific amount time, not every property can be monitored in an online monitoring setting. In this chapter, the term of *Finite Monitorability* will be introduced, which ensures that monitoring a property on infinite streams is possible with finite memory resources and finite time resources per event. As introduction into the setting, some related work will be described, inter alia *TeSSLa*, the programming language which is used for the implementation.

3.1. Related Work

Runtime Verification

As monitoring plays a major role in runtime verification, a short overview of this will be given. The definitions of [LS09] are used, in which *Runtime Verification* is a technique that can detect deviations between the run of a system and its formal specification by checking correctness properties. A *run*, which might also be called *trace*, is sequence of the system states, which might be infinite and an *execution* is an finite prefix of this run. A *monitor* reads the trace and decides, whether it fulfills the correctness properties or violates them.
A distinction is made between *offline* and *online* monitoring. Offline monitoring is using a stored trace, that has been recorded before. Therefore, the complete trace (or the complete part of the trace, that should be analyzed) is known in the analysis. Online monitoring checks the properties, while the system is running, which means that the analysis must be done incrementally on a growing prefix of the trace. Because of memory and time limitations, not all previous states can be read again in online monitoring, more detailed contemplations on the limitations of online monitors will be given in in this chapter.

TeSSLa

TeSSLa (**Te**mporal **S**tream-based **S**pecification **La**nguage) [LSS+18] is a specification language build for stream Stream Runtime Verification. In TeSSLa, all streams in one specification must have a common global clock, but events or changes in a signal may occur in streams irregularly, independent of events in other streams. The verified streams are either considered as signal, which remain unchanged for certain amount of time (called *piece wise constant signals*), or they are *event streams*, in which each event consists of a timestamp and a data value. Both variants can be transferred into each other, like described in [LSS+18]. A formal definition of the TeSSLa language core can be found in [CHL+18], a short overview of the formal definition of event streams will be given next.
An event stream is defined over a time domain $\mathbb{T}$ and a data domain $\mathbb{D}$ and is an possibly infinite sequence $s = a_0a_1... \in \mathcal{S}_D = (\mathbb{T}\cdot\mathbb{D})^\omega \cup (\mathbb{T}\cdot\mathbb{D})^+ \cup (\mathbb{T}\cdot\mathbb{D})^* \cdot (\mathbb{T}_\infty \cup \mathbb{T}\cdot\{\perp\})$ where $a_{2i} < a_{2(i+1)}$ for all i with $0 < 2(i+1) < |s|$ ($0 < 2(i+1) < \infty$ if the sequence is infinite). While the data domain $\mathbb{D}$ can be bounded (e.g. boolean or integer) or unbounded (e.g. maps or lists), the time domain $\mathbb{T}$ is a *totally ordered semi-ring* $(\mathbb{T}, 0, 1, +, *, \leq)$, that is not negative.
In TeSSLa, computations are done, when new events are arriving. Based on the specification, output streams are generated with events on the same timestamps as the used input streams, but filtering is possible, where not all input events produce output events. With the *delay*-operator, it is possible to create new timestamps. In a memory perspective, streams are not understood as event streams, but as *piece wise constant signals*. Only the timestamp and the data value of the youngest event can be directly accessed. This event is available until the next event of this stream occurs. With the use of the *last*-operator, which can be used recursively, the data value of the previous event can be accessed. Another important operator is the *lift*-operator, which applies a function on data values (for example the + operator) on the data value of every event of one or more streams and creates a new stream with events at the same timestamps and the results of the function as data values.

LOLA-Efficient Monitorable

[DSS+05] introduces *LOLA*, a specification language for the observation of synchronous event streams, comparable to TeSSLa.The paper also defines the term of *Efficiently Monitorable Specifications*, which describes that the worst case memory requirement of a LOLA Specification is independent of the length of the observed trace.

Deterministic Finite State Transducer[Ber79]

A *Deterministic Finite State Transducer*(DFST) is a 5-Tuple $(\Sigma, \Gamma, Q, q_0, \delta)$, where

- Σ is an input alphabet
- Γ is an output alphabet
- Q is a finite set of states, with initial state q_0
- $\delta : Q \times \Sigma \to Q \times \Gamma$ is a state transition function

DFSTs are similar to deterministic finite automata, with two major differences. First, the transition function outputs a symbol of Γ at every transition and second, the DFST is not accepting, it only *transduces* an input word.

Timed Deterministic Finite State Transducer

Timed Deterministic Finite State Transducer(TDFST) are an extension of DFSTs. They are defined as 6-Tuple $(\Sigma, \Gamma, Q, q_0, C, \delta)$, where

- Σ is an input alphabet
- Γ is an output alphabet
- Q is a finite set of states, with initial state q_0
- C is a set of clocks
- $\delta : Q \times \Sigma \times \Theta(C) \to Q \times 2^C \times \Gamma$ is a state transition function

Additional to the input symbols and the current state, the state transition function of TDFSTs takes a set of clock constraints into account when defining the next state of the transducer.

3.2. Monitorability

In this section, the term *Finite Monitorability* is introduced. It represents a stricter alternative to *Efficiently Monitorable Specifications* mentioned above, by also restricting the allowed run time per event. *Finite Monitorability* ensures, that the worst case memory consumption of a monitor is independent of the input events, consequently can every finite monitorable property be monitored with a efficient monitorable LOLA specification.

Preliminary - Timestamps

As we consider possibly infinite streams, the time value of events can also grow into infinity. This is problematic, because it leads to infinite memory and runtime requirements, which cannot be met, especially not in the context of online monitoring. Therefore, the time domain $\mathbb{T}$ is restricted by the following constraints:

- $\mathbb{T}$ must be discrete.
- The first used timestamp has the value $t_0 = 0$
- All used timestamps must be smaller than t_{max}.
  t_{max} must be big enough, so it is not reached in practical use [1].
- The distance between two subsequent time values is small enough to observe the wanted property.

Finite Monitorability

The concept behind the definition of *finite monitorability* is, that a monitor for event streams is defined by three parts, first the state transition function, a state defining the memory of the monitor and an output function. At each timestamp containing input events, the new state is created by applying the state transition function to the previous state and the input events of the current timestamp. The output function is applied to the new state and the previous output and evaluates, whether the specification is met until this timestamp.
All following definitions of streams and functions follow the syntax and semantic from [CHL+18]. The left half of figure 3.1 visualizes the definitions, which will be done now.

- **Input Streams**
  Let $S_1, S_2, ..., S_n$ be the input streams with
  $\forall i : S_i = (\mathbb{T} \cdot \mathbb{D}_i)^\omega \cup (\mathbb{T} \cdot \mathbb{D}_i)^+ \cup (\mathbb{T} \cdot \mathbb{D}_i)^* \cdot (\mathbb{T}_\infty \cup \mathbb{T} \cdot \{\perp\})$
  All types D_i have a finite size.
- **State Stream**
  Let S_{state} with $S_{state} = (\mathbb{T} \cdot \mathbb{D}_{state})^+ \cup (\mathbb{T} \cdot \mathbb{D}_{state})^*$ be a state stream, where $\mathbb{D}_{state}$ has a constant worst case memory requirement.
  Further let $f : S_1 \times S_2 \times ... \times S_n \times S_{state} \to S_{state} \times \mathbb{T}$ be a state transition function, which defines the state stream in an incremental fashion:

[1] for example, a 64-bit unsigned integer variable is enough, to cover nanoseconds for 584.55 years

$\forall t \in \mathbb{T} \exists i \in \{1, 2, ..., n\} : S_i(t) \in \mathbb{D}_i$
$\rightarrow S_{state}(t) = f(S_1(t), S_2(t), ..., S_n(t), last(S_{state}, merge(S_1, S_2, ..., S_n))(t))$
The runtime of f is in $\mathcal{O}(1)$.

- **Output Stream**
Let $S_{output} = (\mathbb{T} \cdot \{true_{until}, false\})^+ \cup (\mathbb{T} \cdot \{true_{until}, false\})^*$
be the output stream, which is defined via a function
$g : \mathbb{D}_{state} \times \{true_{until}, false\} \times \mathbb{T} \rightarrow \{true_{until}, false\} \times \mathbb{T}$
The runtime of g is in $\mathcal{O}(1)$.

- **Evaluation** A property of a set of streams is called *Finite Monitorable*, if a state transition function f, a type $\mathbb{D}_{state}$ and a output function g exist, which fulfill the characteristics called above, and which outputs $true_{until}$, as long as the property is fulfilled and $false$, in any other case. It should be noted that these definitions are *timestamp conservative*, because no new timestamps are created.

- **Equivalences**
The combination of a finite state and state transition function is equivalent to a Deterministic Finite State Transducer(DFST), where
  - $Q = \mathbb{D}_{state}$ is the finite set of possible states with initial state q_0
  - $\Sigma = ((\mathbb{D}_1 \times \mathbb{T}), ..., (\mathbb{D}_n \times \mathbb{T}))$ the input alphabet
  - $\Gamma = \mathbb{D}_{state}$ is the output alphabet and
  - $\delta : Q \times \Sigma \rightarrow Q \times \Gamma$ is the transition function.

  The definition of the output function $g : \mathbb{D}_{state} \times \{true_{until}, false\} \times \mathbb{T} \rightarrow \{true_{until}, false\} \times \mathbb{T}$ is the same as described above.
  It should be noted that the function g could also be included into the DFST, which represents the state and the state transition function of the monitor without changing the expressiveness. This is not done to keep analogies to the following definition.

Finite Monitorability with Delay

Most of the TADL2 constraints can not be monitored in a *timestamp conservative* way. For example, the *RepeatConstraint* with the attributes *lower* = *upper* = 4 and *span* = 1 expects subsequent events to have a time distance of 4. If one event is missing, the output of a timestamp conservative monitor would remain $true_{until}$, until the next input event arrives. Therefore, the monitor cannot not check the constraint correctly. Because of this problem, the definition of *Finite Monitorability* is expanded by the ability of introducing new timestamps. To ensure the finiteness

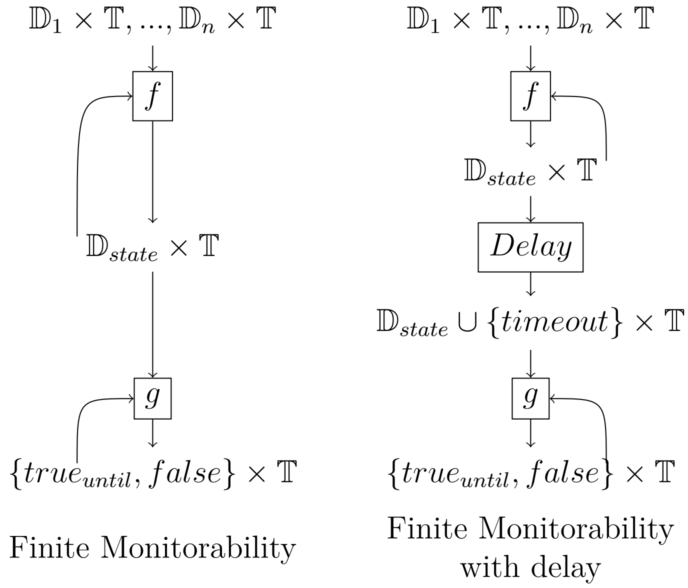

Figure 3.1.: Overview Finite Monitorability - with or without *delay*

of the monitor, only one new timestamp can be introduced. The following definitions are visualized in the right half of figure 3.1.

- **Input Streams**
  The definition of the input streams are unchanged.
- **State Stream** The function f remains unchanged, but the state stream S_{state} is expanded by an *timeout* value, which is inserted after a specific period of time, in which no input event has arrived. Like before, the runtime of f is in $\mathcal{O}(1)$.
- **Delay**
  A *Delay Generator* is inserted into the definition. It has two tasks, first it copies each input it gets from the state transition function f to its output. At the timestamp where an input is copied, a timer, which length depends on the state of the monitor, is started. If the next input comes before the timer runs out, the timer is resetted and started again. If the timer runs out, the Delay Generator outputs the *timeout* signal, which is repeated at every following input. After the timer has run out once, it is not started again. The calculation of the needed delay is in $\mathcal{O}(1)$ in terms of time.
- **Output Stream**
  The input of the output function g is expanded by the *timeout* value:
  $g : (\mathbb{D}_{state} \cup \{timeout\}) \times \mathbb{T} \rightarrow \{true_{until}, false\} \times \mathbb{T}$

Obviously, g always outputs false, if the functions receives the *timeout* value. The definition of the output stream S_{output} remains unchanged.

- **Evaluation**
  A property of a set of streams is called *Finite Monitorable with Delay*, if a function f, a type $\mathbb{D}_{state}$, a delay generator and a function g exist, which fulfill the characteristics called above, and which outputs $true_{until}$, as long as the property is fulfilled and $false$, in any other case.

- **Equivalences** Similar to *Finite Monitorability* (without Delay), equivalences to finite state machines can be worked out. Like before, the combination of a finite state and state transition function is equivalent to a Deterministic Finite State Transducer(DFST), where

  - $Q = \mathbb{D}_{state}$ is the finite set of possible states with initial state q_0
  - $\Sigma = ((\mathbb{D}_1 \times \mathbb{T}), ..., (\mathbb{D}_n \times \mathbb{T}))$ the input alphabet
  - $\Gamma = \mathbb{D}_{state}$ is the output alphabet and
  - $\delta : Q \times \Sigma \rightarrow Q \times \Gamma$ is the transition function.

  The *Delay Generator* is equivalent to an extended version of *Timed Deterministic Finite State Transducer*, which allows ε-transitions, which are guarded by a clock constraint, but do not require an input symbol to perform a state transition. This special form of TDFSTs will be defined next.

  Let $tmr : \mathbb{D}_{state} \rightarrow \mathbb{T}$ be a function that determines the required delay for every possible state of the monitor. Let further

  - $Q = \{q_{start}, q_{timeout}\} \cup \{q_{wait,i} | \forall i \in \mathbb{D}_{state}\}$ be a finite set of states with initial state q_{start}
  - $\Sigma = \mathbb{D}_{state}$ be an input alphabet
  - $\Gamma = \mathbb{D}_{state} \cup \{timeout\}$ be an output alphabet
  - $C = \{c\}$ be a set of exactly one clock and
  - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Theta(C) \rightarrow Q \times 2^C \times \Gamma$ a state transition function. δ is defined as:

$$\begin{aligned}
\forall i \in \mathbb{D}_{state} : \delta(q_{start}, i, \emptyset) &= (q_{wait,i}, \{c\}, i) \\
\forall i, i' \in \mathbb{D}_{state} : \delta(q_{wait,i'}, i, \{c < tmr(i')\}) &= (q_{wait,i}, \{c\}, i) \\
\forall i \in \mathbb{D}_{state} : \delta(q_{wait,i}, \varepsilon, \{c < tmr(i)\}) &= (q_{timeout}, \emptyset, timeout) \\
\forall i \in \mathbb{D}_{state} : \delta(q_{timeout}, i, \emptyset) &= (q_{timeout}, \emptyset, timeout)
\end{aligned}$$

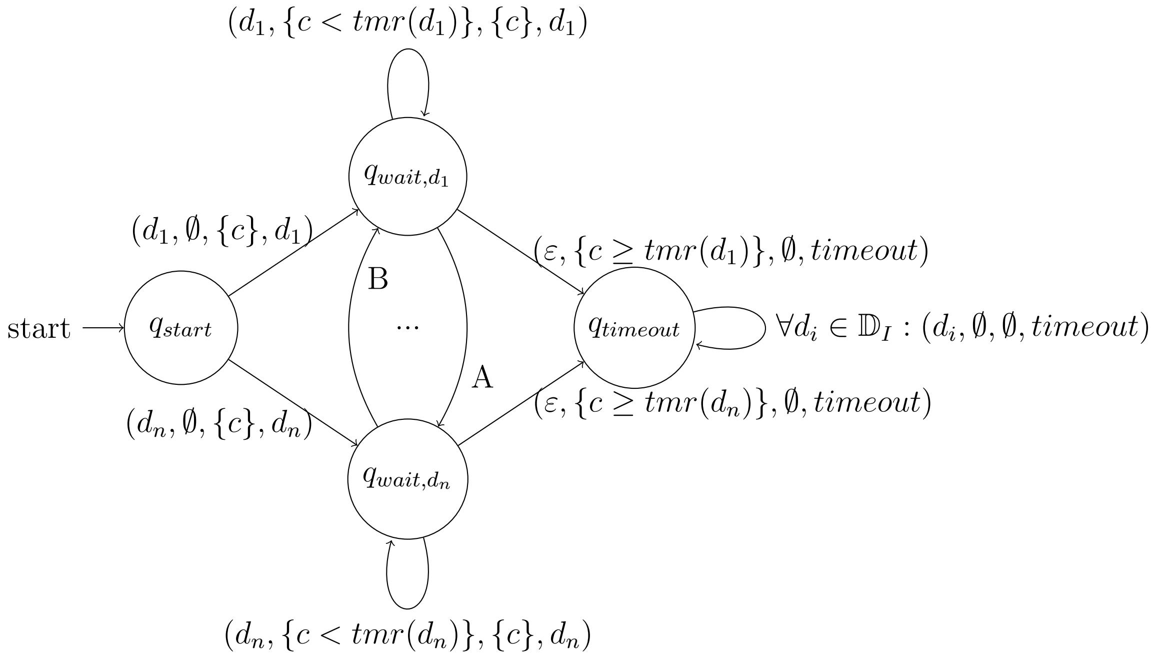

Figure 3.2.: Visualization of the Delay Generator. Description A means $(d_n, \{c < tmr(d_1)\}, \{c\}, d_n)$ and description B means $(d_1, \{c < tmr(d_n)\}, \{c\}, d_1)$.

This definition is visualized in figure 3.2. On the left side is the initial state. The first input leads to a transition to the wait state of the corresponding input state. The clock c is resetted in this transition. In the middle column of the figure are the wait states, one for each possible state of the monitor. $|\mathbb{D}_I| + 1$ transitions leave each wait state, one is the ε-transition introduced above, which is constrained in a way, that the value of clock c must be equal or greater than the corresponding delay time. This ε-transition leads to $q_timeout$ and outputs the *timeout* symbol. Every other transition leaving the waiting states are done at input symbols, while the value of clock c is less than the corresponding delay time. In these transitions, the input symbol d_i is used as output and clock c is resetted. In the output state, each input symbol leads to a repetition of the *timeout* symbol.

Like before, the output function is defined as $g : (\mathbb{D}_{state} \cup \{timeout\}) \times \mathbb{T} \to \{true_{until}, false\} \times \mathbb{T}$.

Non-Finite Monitorability

Not all TADL2 constraints are finite monitorable, because they may require memory or time resources, which size is not independent from the events of the observed trace. This makes correct online monitoring of these constraints impossible for arbitrary traces, because a machine with infinite resources does not exist. In a practical view, many of these problems are solved by using a system with finite memory, with the hope that its finite memory would be enough, to cover the inputs of the "real world". In these cases, a distinction is useful, as the memory requirements of some properties grow continuously with every input event, and other constraints only require infinite resources in worst case scenarios. The ones with continuous requirement growth will be called *always Non-Finite Monitorable* and the others *worst case Non-Finite Monitorable* in the following.
Obviously, the constraints with continuous resource requirement growth cannot be monitored infinitely, but the constraints, that only need infinite resources, can be monitored in many cases.

4. Analysis of the Monitorability of the TADL2 Timing Constraints

In this chapter, each of the TADL2 constraints will classified into the classes *Finite Monitorable*, *Finite Monitorable with Delay* and *Non-Finite Monitorability*, like defined in chapter3. For the last class, it will be demonstrated, if the constraint is non-finite monitorable in any cases or just in worst case scenarios.

DelayConstraint

The *DelayConstraint* is defined as

$\forall x \in source : \exists y \in target : lower \leq y - x \leq upper.$

and describes that in the time interval between *lower* and *upper* after any *source* event, there is an *target* event. Therefore, the state that need to be stored to monitor the *DelayConstraint* is the set of *source* events, that did not have a matching *target* event. Updates to this state and outputs of the monitor are done at *source* and *target* events and at delay timestamps *upper* after *source* events, if there hasn't been a matching *target* event.
The maximal required storage size of the state depends on the number of *source* events, which can possibly occur in any time interval of the length *upper*. An example of this worst case situation can be seen in figure 4.1. The attributes in this example are $lower = upper = 5$, $source = \{1, 1.1, ..., 5.9\}$ and $target = \{6, 6.1, ..., 11\}$. At timestamp 6, all 49 *source* events must be stored, as they are all required to generate the correct output in this and the following timestamps. At this timestamp, the oldest *source* event can be removed from the storage, as the matching *target* event occurs in this timestamp. With every following *target*, the oldest event can be removed from the storage, until every *source* had its matching *target* event at timestamp 11.
Because the time domain is understood as real numbers in TADL2, a possibly infinite number of events can be placed in any interval of the length *upper*, therefore the required storage space can grow infinitely. Because the *source* events are removed from the state, when a matching *target* event occurs, the required storage space does not grow continuously and infinite resources are only required in worst case scenarios. Therefore, the *DelayConstraint* is *worst case Non-Finite monitorable*.

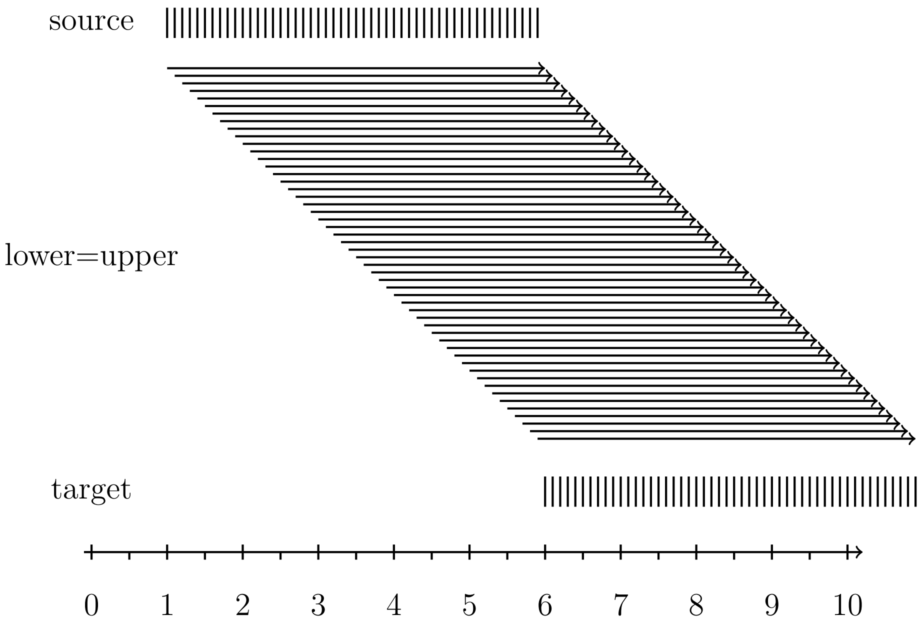

Figure 4.1.: *DelayConstraint* or *StrongDelayConstraint* with $lower = upper = 5$

StrongDelayConstraint

The difference between the *DelayConstraint* and the *StrongDelayConstraint* is, that for every *source* event, there must be exactly one matching *target* event in the *StrongDelayConstraint*. Therefore, the state of the monitor is nearly the same, as every *source* event, that did not have a matching *target* event yet, must be stored. Therefore, the only difference is, when these *source* events can removed from state and the *StrongDelayConstraint* is *worst case Non-Finite monitorable*, like the *DelayConstraint*.

RepeatConstraint

The *RepeatConstraint* defines the time distance between each event and its $span^{th}$ successor. Therefore, the state, that must be stored, consists of the timestamps of the $span + 1$ latest events. The state is updated at every event, the oldest stored event is removed and the timestamp of the current event is placed in the storage. The output function checks, if the time distance between the oldest stored event and the current timestamp is between *lower* and *upper*. To monitor this constraint, a single delay is required, because a missing event, or an event that occurs too late, would not be determined in the right timestamp.
As the memory requirements are fix ($span + 1$ timestamps must be stored) and the

state transition and output function can be programmed in a way that they are in $\mathcal{O}(1)$, the *RepeatConstraint* is finite monitorable with delay.

RepetitionConstraint

The *RepetitionConstraint* is defined as

$RepetitionConstraint(s, lower, upper, span, jitter)$
$\equiv \exists X \subset \mathbb{T} : RepeatConstraint(X, lower, upper, span)$
$\wedge\ StrongDelayConstraint(X, s, 0, jitter)$

The elements of set X follow the RepeatConstraint and the events, which should be monitored, are following in an interval of the length *jitter* after the elements of X. For each element of X, there is exactly one event and vice versa.
The monitoring algorithm for this constraint, which will be explained in detail in 5, stores the upper and lower bounds for the next *span* elements of X. These borders are stored in a list and calculated by

$lowerBound := List\_append(last(List\_tail(LowerBound), s), lowerBoundNow + lower)$ for the lower bound and
$upperBound := List\_append(last(List\_tail(UpperBound), s), upperBoundNow + upper)$.

The oldest item in these lists (the head of these lists) are removed and the newly calculated bounds for the *span* next element of X is inserted. $lowerBoundNow$ and $upperBoundNow$ are the describing the limitations of the element of X right before the current event. They are calculated using the list mentioned above and the timestamp of the current event by the following definition:

$lowerBoundNow := max(List\_head(last(LowerBoundX, s)), time(s) - jitter)$
$upperBoundNow := min(List\_head(last(UpperBoundX, s)), time(s))$

If the timestamp of the current event is between $lowerBoundNow$ and $upperBoundNow$, the output of the monitor is *true*, in any other case, or when the delay ran out, it is $false$.
The size of these lists has a fixed upper limit (*span*) and the state transition and output functions are in $\mathcal{O}(1)$, therefore the *RepetitionConstraint* is finite monitorable with delay.

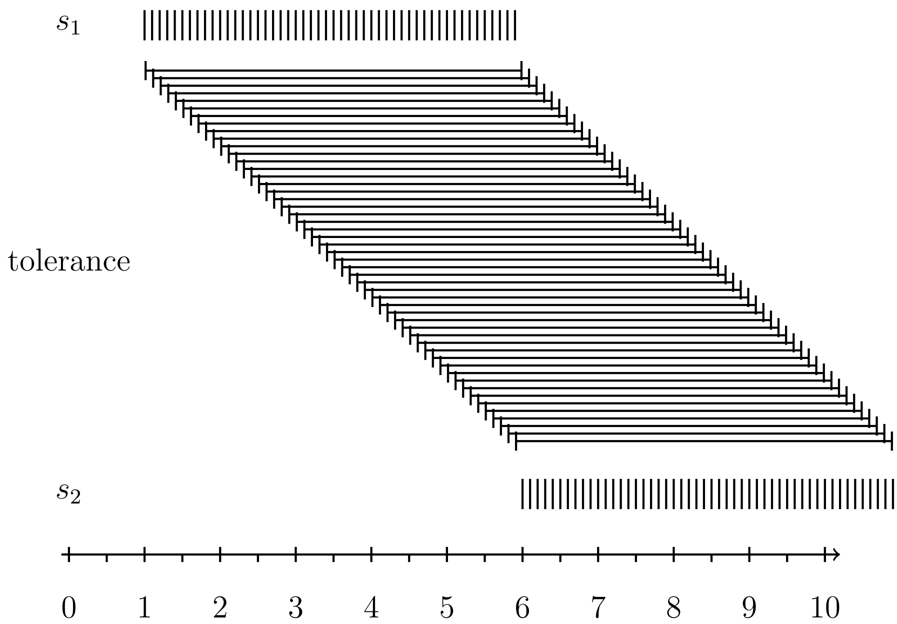

Figure 4.2.: *SynchronizationConstraint* or *StrongSynchronizationConstraint* with *tolerance* = 5

SynchronizationConstraint

The *SynchronizationConstraint* describe groups of event sets, which events occur in common clusters. Each of these sets must have at least one event in each of these intervals. Any events, that lay outside of these intervals are prohibited.
Figure 4.2, which is similar to the example for the *DelayConstraint*, shows an example of this constraint, which is an worst case scenario in terms of monitoring. The *tolerance* interval is 5 timestamps long, the event set s_1 contains the events $\{1, 1.1, ..., 5.9\}$ and s_2 is containing $\{6, 6.1, ..., 11\}$. Each of the events of s_1 must be stored until the end of the *tolerance* interval, otherwise it would be impossible to check the constraint correctly. Like described in chapter 4, an infinite number of events can be placed in this interval, therefore infinite memory resources are needed. Because the required storage space is not growing continuously, as the stored events can be removed at the end of the *tolerance* interval, the *SynchronizationConstraint* is worst case Non-Finite monitorable.

It should be noted, that the illustration of the constraint in figure 4.2 may be misleading, because the *tolerance* intervals are only shown after the events of s_1, not after the events of s_2. Every implementation of a monitor for this constraint must also store the events of s_2 for the length of *tolerance*, as they could be important for events following after them.

StrongSynchronizationConstraint

The difference between the *StrongSynchronizationConstraint* and the *SynchronizationConstraint* is, that in the StrongSynchronizationConstraint, only one event per event set is allowed in each synchronization cluster. Therefore, this constraint can be classified as worst case Non-Finite monitorable with the same argumentation as the previous constraint.

ExecutionTimeConstraint

The *ExecutionTimeConstraint* ensures that the time distance between *stop* and *start* events, not counting interruptions (which are specified by *preempt* and *resume* events).
Under the assumption that the input events are in logical order (every execution is started by an *start* event and finished by an *stop* event, every *preempt* event is directly followed by an *resume* event and no *preempt* or *resume* events occur outside of the intervals spanned by *start* and *stop* events), three time values must be stored to monitor this constraint. First, the timestamp of the latest *start* event. Second, the timestamp of the latest *preempt* event and third, the sum of each timestamp of *resume*, minus the respectively latest timestamp of *preempt*. The sum is reseted at every *start* event. These values are updated on events in *start*, *stop* and *preempt*.
For the output function, the run time can be calculated by
$runtime = time(now) - time(start) - (sum(time(resume) - time(preempt)))$.
At any event, this value must smaller or equal to *upper* and at *stop* events, additionally the runtime must be greater or equal to *upper*.
To monitor this constraint correctly, a delay is required, when an *stop* event is late or missing. The required storage space is fixed, also the runtime of the state transition and output function is in $\mathcal{O}(1)$, therefore the *ExecutionTimeConstraint* is finite monitorable with delay.

OrderConstraint

The *OrderConstraint* describes, that an i^{th} *target* event must exist, if an i^{th} *source* event exists and that the i^{th} *target* event occurs after the i^{th} *source* event. Because it is possible that an arbitrary large number of *source* events occur before the first *target* occurs, a possibly infinite large number must be stored, which requires infinite memory resources. As this is only a worst case scenario and the size of the stored number can be decreased, when a *target* event occurs, the *OrderConstraint* is worst case non-finite monitorable.

ComparisonConstraint

The *ComparisonConstraint* defines an ordering relation between two single timestamps. Therefore, no additional storage is needed and as the relations ($\leq, <, \geq, >, =$) can be decided in constant time for discrete timestamps, this Constraint is finite monitorable.

SporadicConstraint

The *SporadicConstraint* is defined via the *Repetition*- and *RepeatConstraint* without introducing any new timestamps in the definition of the *SporadicConstraint*. These Constraints are finite monitorable with delay, therefore the *SporadicConstraint* is also finite monitorable with delay.

PeriodicConstraint

The *PeriodicConstraint* is special application of the *SporadicConstraint*, therefore it is also finite monitorable with delay.

PatternConstraint

The *PatternConstraint* was redefined to

$$\begin{aligned}
&\exists X : PeriodicConstraint(X, period, 0, 0)\\
&\quad \wedge \forall i : StrongDelayContraint(X, event, offset_i, offset_i + jitter)\\
&\quad \wedge\ RepeatConstraint(event, minimum, \infty, 1)
\end{aligned}$$

in section 2.2.2. The events ($event$), which are given as attribute, occur after strictly periodic timestamps (X). The distances between the elements of X and the following events is defined by $offset$.
This constraint can be monitored by storing upper and lower limits of the current latest element of X and the number of event occurrences, reseted by every $|offset|$ event ($count(event)$ modulo $|offset|$). The limits of the elements of X can be narrows down by every event occurrence, because the valid distance between the event and the element of X is known by *offset* and $jitter$. At every $|offset|^{th}$ event occurrence, the limitations of the current X must be increased by *period*. The validity of the constraint can be tested by checking, that the current event has the right distance to the limitations of the current element of X. To be able to recognize late or missing events, a delay is required.
Because the memory requirements (two timestamps and a finite integer) are finite

and the mentioned state transition and evaluation functions can be implemented in constant time, the *PatternConstraint* is finite monitorable with delay.

If the redefinition of the *PatternConstraint* is not done, the constraint can be reduced to

$RepeatConstraint(event, minimum, \infty, 1)$

like stated before in section 2.2.2. In this variant, the constraint is finite monitorable (without delay), because only the minimal distance between two events must be checked.

ArbitraryConstraint

The *ArbitraryConstraint* is defined as combination of the *RepeatConstraint*:

$ArbitraryConstraint(event, minimum_1, ..., minimum_n, maximum_1, ..., maximum_n)$
$\Leftrightarrow \forall i \in 1, ..., n : RepeatConstraint(event, minimum_i, maximum_i, i)$.

The *RepeatConstraint* is finite monitorable with delay, therefore the *ArbitraryConstraint* is also finite monitorable with delay.

BurstConstraint

The *BurstConstraint* is defined as combination of the *RepeatConstraint*:

$RepeatConstraint(event, length, \infty, maxOccurrences)$
$\quad \wedge RepeatConstraint(event, minimum, \infty, 1)$

The *RepeatConstraint* is finite monitorable with delay, therefore the *BurstConstraint* is also finite monitorable with delay.

EventChain

EventChains, which are required for the following constraints, are defined as sets of *stimulus* and *response* events. The events have an color attribute, which describes the causal connection individual events of *stimulus* and *response*. It is required, that any *stimulus* event with a specific color must occur before the first *response* event with the same color. The datatype of this attribute is not specified, except that it may be infinite and an equality test exist.
Monitoring this property is difficult, because it is required to store every color which

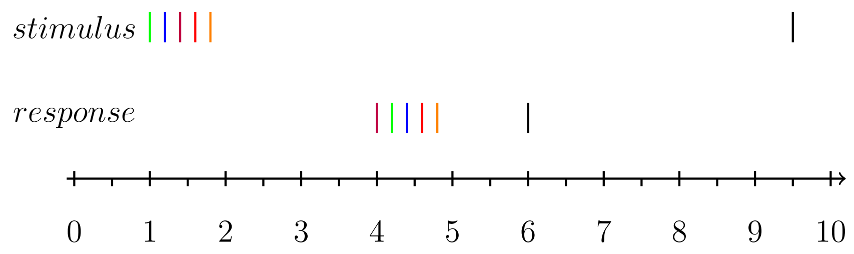

Figure 4.3.: Color attribute

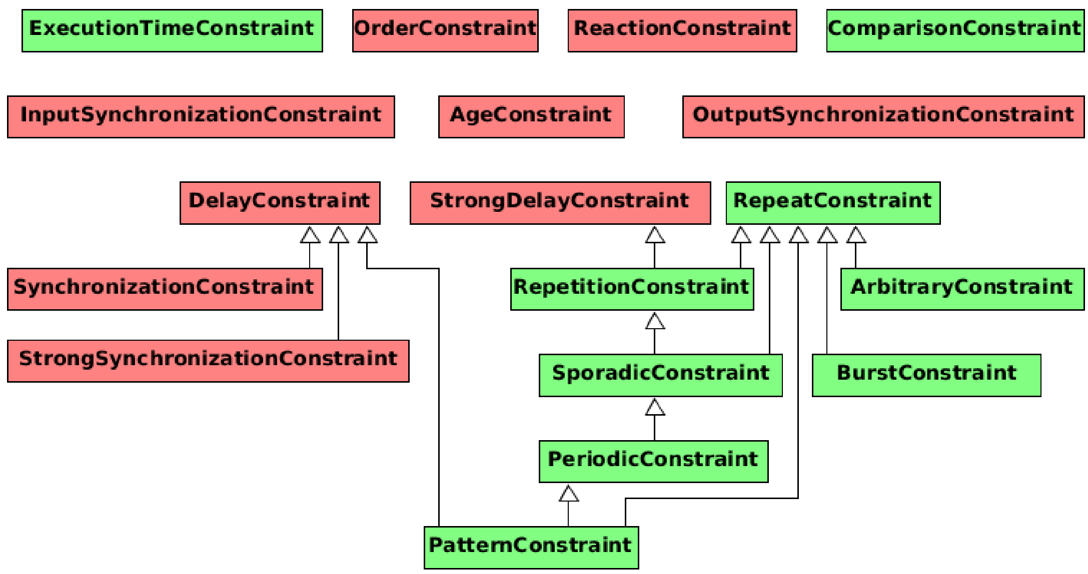

Figure 4.4.: Overview over constraints - Finite Monitorable - Non-Finite Monitorable

has occurred in *response*. The reason for this can be seen in figure 4.3. In the interval between the timestamps 1 and 2, there are 5 events of different colors in *stimulus*. Their counterparts in *response* occur in the interval between 4 and 5. In timestamp 6, there is an event in *response* with a color, that hasn't been used in *stimulus* before, therefore this color may not be used in *stimulus* anymore. To check this for further events in *stimulus*, it is required to know any color that previously occurred in *response*.
The memory consumption to monitor this is growing continuously with any event that introduces a new color in *response*, therefore any constraint, that requires the color attribute (***ReactionConstraint, AgeConstraint, OutputSynchronizationConstraint, InputSynchronizationConstraint***) is always non-finite monitorable.

Figure 4.4 gives an overview, which TADL2 timing constraints are finite monitorable and which are not. All of the 9 finite monitorable constraints require the possibility to create new timestamps (delays), except the *ComparisonConstraint*, which only compares timestamps. The other half of the constraint is not finite monitorable, but they also split up into two categories, *worst case* and *always non finite monitorable*. All constraints that are using the *color* attribute of the events (*Reaction-*, *Age-*, *InputSynchronization-* and *OutputSynchronization*) are *always non finite monitorable*. The other 5 red colored constraints are *worst case non finite monitorable*. The arrows show, which constraint is defined via other constraints, for example the *RepetitionConstraint* is defined via the *StrongDelay-* and *RepeatConstraint*. It should be noted, that constraints, which are defined via non finite monitorable constraints, can still be finite monitorable, because of further restrictions, which limit the required storage space or runtime.

5. Implementierungen

In this chapter, the implementation of the monitor of each constraint will be explained. Three major aspects will be considered for every constraint

1. A short **documentation** of the implementation
2. An analysis of the **computational complexity** in terms of time consumption per event and overall memory usage
3. A **performance analysis** of the implementation by analyzing a large, randomly generated trace

All implementations have in common that they consist of 2 or 3 sections, similar to the state transition, delay (if needed) and output as defined in chapter 3. These sections are the basis for the analysis of the computational complexity, because the generated state defines the required memory capacity and the function connecting these sections define the required time per event.
The performance analysis is done by monitoring a trace with 10.000 events, which is randomly generated. The implementations are run in the TeSSLa interpreter, the events of the traces are fed in subsequently by a java program, which puts the events into the standard input stream of the interpreter and measures the time until an answer appears on the standard output stream. This program is exactly described in section A.2 and will be called *Event Feeder* hereafter.
To increase reproducibility and minimize disruptive influences on the timing behavior, the interpreter is run on a *Raspberry Pi 2 Model B*, using Raspberry Pi OS lite and openJDK version 11.0.8. Because the TeSSLa source code is not pre-compiled and must be interpreted at run time, the first event is send to the monitor 60 seconds after the start of the program.Each constraint monitor is run three times, the minimal, maximal and average needed time will be listed and described.

DelayConstraint

The implementation of the *DelayConstraint* monitor stores a linked list of *source* events, which did not have a matching *target* event yet as state. This list is expanded by every *source* event, which is appended at the end of the list. If a *target* event occurs, all matching *source* events (possibly none) are removed from the list. Like stated in section 4, this list can grow infinitely in worst cases, when the time

	Minimum	Maximum	Average
run 1	5.9ms	1091.66ms	9.71ms
run 2	5.91ms	1080.43ms	9.71ms
run 3	5.86ms	793.78ms	9.68ms
Average	5.98ms	988.62ms	9.7ms

Table 5.1.: Runtime of the *DelayConstraint* Monitor

domain defined in an uncountable way. In these worst cases, an infinite number of *source* events may, before any event can be removed from the list, because a matching *target* event occurs.
TeSSLa is using integer values as time domain, therefore it is countable and the list cannot grow infinitely. The largest possible size of this list is equal to the parameter *upper*, therefore, and because this list is the only growable memory usage, the algorithm is in $\mathcal{O}(upper)$ in terms of memory.
The state transition as described above is in $\mathcal{O}(upper)$ in terms of time. Appending an *source* event to the list is done in constant time. Removing all events that matched with a *target* event may require to check every event in this list in worst cases, because possibly, all of them must be removed.
The output function checks, if the updated list of unmatched *source* events is either empty, or the event in the head of the updated list is not older than *upper*. Therefore, it is in $\mathcal{O}(1)$.
The required delay period is calculated by adding *upper* to the timestamp of the head of the list of unmatched *source* event, subtracted by the timestamp of the current event ($\mathcal{O}(1)$).

The run time per event of the *DelayConstraint* monitor with the generated trace can be seen in table 5.1. This trace has 3114 *source* and 6886 *target* events and was generated in a way that fulfills the *DelayConstraint* with the attributes $lower = 4$ and $upper = 10$. The average minimal runtime per event was 5.98ms, the average maximum 988.62ms and the overall average runtime was $9.7ms$ (103.1 events per second). The large difference between the average and maximal runtime can be traced back to the used testing setup. First, Raspberry Pi OS lite is not a real time operating system, therefore interruptions in the program flow of the TeSSLa interpreter or the event feeder can occur in an unpredictable way. Additionally, Garbage Collectors are used in the Scala and Java programming languages that are used for TeSSLa and the event feeder, which results in additional interruptions. All of these interruptions are not detected by the event feeder and distort the results.

	Minimum	Maximum	Average
run 1	0.16ms	595.1ms	9.42ms
run 2	0.1ms	924.69ms	9.37ms
run 3	0.17ms	874.97ms	9.38ms
average	0.14ms	798.25ms	9.39ms

Table 5.2.: Runtime of the *StrongDelayConstraint* Monitor

StrongDelayConstraint

The *StrongDelayConstraint* is implemented very similarly to the *DelayConstraint*. The only difference is, that only the head of the list of unmatched *source* events is removed, when a matching *target* event occurs. Therefore, the state transition is in $\mathcal{O}(1)$ in terms of time per event, while the memory consumption is still in $\mathcal{O}(upper)$. Additionally, the output function checks, if *target* event occurrences have exactly one matching *source* event (which always is in the head of the list). Therefore, it is still in $\mathcal{O}(1)$. The calculation of the delay period remains unchanged.

RepeatConstraint

The implementation of the *RepeatConstraint* stores the timestamps of the *span* previous events as state, using TeSSLa's *last* operator recursively (a macro called *nLastTime* was programmed for this). Therefore, *span* timestamps are stored and the *last* operator is called *span* times, which means the state transition function is in $\mathcal{O}(span)$ in terms of time and the implementation is in $\mathcal{O}(span)$ in terms of memory. The timestamp of the $span^{th}$ oldest event is stored directly as integer, therefore it can be accessed in constant time.
The required delay is calculated by adding *upper* to the $span^{th}$ oldest event (or the first event, if there hasn't been *span* events before) minus the current timestamp, therefore it is in $\mathcal{O}(1)$ in terms of time, because the relevant timestamps can be directly accessed, like stated before.
The output function checks, if the $span^{th}$ oldest event is not older than *upper* and not younger than *lower*. If there hasn't been *span* events before, it is checked, if the first event is not older than *upper*. Because the timestamps of the $span^{th}$ oldest and the first event and *lower* and *upper* can be directly accessed, the output function is in $\mathcal{O}(1)$ in terms of time.
The runtime per event can be seen in table 5. As trace, 10.000 events, which fulfill the *RepeatConstraint* with the attributes $lower = 100$, $upper = 200$ and $span = 2$ were created. The average minimal run time was 3.62ms and the average of the

	Minimum	Maximum	Average
run 1	3.61ms	919.84ms	5.98ms
run 2	3.6ms	985.69ms	6ms
run 3	3.64ms	903.94ms	6.08ms
average	3.62ms	936.49ms	6.02ms

Table 5.3.: Runtime of the *RepeatConstraint* Monitor

average run times was 6.02ms. Therefore, ca. 161 events were processed per second in average.

RepetitionConstraint

The *RepetitionConstraint* is defined as

$RepetitionConstraint(s, lower, upper, span, jitter)$
$\equiv \exists X \subset \mathbb{T} : RepeatConstraint(X, lower, upper, span)$
$\wedge\ StrongDelayConstraint(X, s, 0, jitter)$

The implementations of the *Repeat-* and the *StrongDelayConstraint* cannot be used for the implementation of this constraint, because the timestamps of X are unknown and need to be narrowed down.
Relevant for the monitoring are the bounds of the elements of X, which precede the actual events in the event stream s. The bounds are stored as two lists with the length of *span*. One list is containing the lower bounds for the next *span* X, the other list is containing the upper bounds. At every input event, the new boundaries for the $span^{th}$ next X are calculated, the lower bound by $max(List\_head(last(LowerBoundX, e)), time(e) - jitter)$ and the upper bound by $min(List\_head(last(UpperBoundX, e)), time(e))$. These new boundaries are appended to the end of the lists, while the oldest entries in the head of the lists are removed. These two lists with the size of *span* are the only storage, which size is dependent on the input, therefore the algorithm is in $\mathcal{O}(span)$ in terms of memory. The run time of the state transition function is in $\mathcal{O}(1)$, because the described operations are done in constant time.
The output function checks, if the current timestamp is between the lower bound for the current timestamp of X and *jitter* behind the upper bound for that value. If this is the case, the output is *true*, in any other case, it is false. Because the upper and lower bound for the current X value can be directly accessed (they are the head of the lists), the output function is in $\mathcal{O}(1)$.
The required delay period is calculated by adding *jitter* to the timestamp of the head of the list of the upper limits, subtracted by the current timestamp($\mathcal{O}(1)$).

	Minimum	Maximum	Average
run 1	0.07ms	858.08ms	8.25ms
run 2	0.07ms	856.01ms	8.30ms
run 3	0.06ms	898.51ms	8.37ms
average	0.07ms	870.87ms	8.31ms

Table 5.4.: Runtime of the *RepetitionConstraint* Monitor

SynchronizationConstraint

The *SynchronizationConstraint* is defined via an application of the *DelayConstraint*, but the application uses a set of unknown timestamps($\exists X : ...$), therefore the *DelayConstraint* cannot be used for the implementation of this Constraint.
In the implementation of the *SynchronizationConstraint*, a set of information for every event, that occurred not longer than *tolerance* ago, is stored in a linked list. This information contains the stream, in which the event occurred, the timestamp of the event occurrence and a boolean variable, that expresses if a fulfilled synchronization cluster for this event has already been found.
This list is updated by every event occurrence in three steps. First, each event occurrences in this timestamp is appended to this list. Second, the list is separated into two parts, one with events older and one with events younger than $tolerance$. The part of old events is still stored in this timestamp, but removed after it. The younger events form the state that is stored for the next event occurrence. Third, it is checked, if at least one event of every stream is part of the list of younger events. In this case, a fulfilled synchronization cluster has been found and the boolean variable, that states if a synchronization cluster is found for this event, is set to *true* for all events in this list.
Similar to the *DelayConstraint*, this list can grow infinitely, when the time domain is uncountable, which is not the case in TeSSLa. Because the TeSSLa uses integers as time domain, at most $|event|$[1] $* tolerance$ events can occur in the $tolerance$ interval. Therefore, the algorithm is in $\mathcal{O}(|event| * tolerance)$ in terms of memory. The first step of the state transition is in $\mathcal{O}(|event|)$, because at most $|event|$ events have to be appended to the list. In worst cases, every event in the list(which is in ascending order) is older than tolerance, therefore the separation in the second step of the state transition is in $\mathcal{O}(|event| * tolerance)$ in terms of time. In the third step, the complete stored list of young events must be examined, to check if the cluster is fulfilled and, if needed, every event in the list must be set to fulfilled. Therefore the third step is in $\mathcal{O}(|event| * tolerance)$ in terms of time.
The output function checks, if the boolean variable of each event in the list of events, which are older than *tolerance*, is set to true. If not, the constraint is not fulfilled.

[1] |event| is the number of streams, not the number of events.

	Minimum	Maximum	Average
run 1	48.01ms	19968.65ms	451.72ms
run 2	49.51ms	19950.48ms	453.85ms
run 3	49.42ms	20071.04ms	453.09ms
average	48.98ms	19996.72ms	452.89ms

Table 5.5.: Runtime of the *SynchronizationConstraint* Monitor

Because this list can have the size $|event| * tolerance$, the output function also is in $\mathcal{O}(|event| * tolerance)$ in terms of time.
The required delay is calculated by adding *tolerance* to the timestamp of the oldest stored unsatisfied event, subtracted by the timestamp of the current timestamp ($\mathcal{O}(|event| * tolerance)$).
The performance was tested by a trace of three streams, which fulfills the constraint with the attribute $tolerace = 3$. Even with this small *tolerance* value, the performance on the raspberry pi was significantly worse than for the previous constraints. The average minimal runtime was 48.98ms, the maximal runtime was over 20s and the overall average run time was 452.89ms. Therefore, only 2.2 events per processed in average. This slowness is due to the fact that the complete stored list of events, which are younger than *tolerace*, must be go trough several times per event. The bad performance will make this implementation unusable for most real world use cases.

StrongSynchronizationConstraint

The *StrongSynchronizationConstraint* is defined as application of the *StrongDelayConstraint*, but this application cannot be used for the implementation, like in the previous constraint.
The difference between the *Synchronization-* and the *StrongSynchronizationConstraint* is, that each event can only be part of one synchronization cluster in the *StrongSynchronizationConstraint*. Therefore, the implementation is different to the implementation of the previous constraint. Not every event is stored separately, only information about synchronization clusters, containing their start time and in which stream an event occurred in this cluster, are stored.
At event occurrences, the event is either added to one synchronization cluster, or a new cluster with the start time of the event is added to the list. In a second step, every fulfilled cluster is removed from the list.
This list is at most *tolerance* long (events in all timestamps in one stream, no events in other streams). The size of individual entries of the list is dependent on the number of streams, because they store a boolean variable for every stream. Because of these length restrictions of the list, the stated state transition is in

	Minimum	Maximum	Average
run 1	6.17ms	647.86ms	17.14ms
run 2	6.25ms	979.02ms	17.18ms
run 3	5.17ms	937.32ms	17.12ms
average	5.86ms	854.73ms	17.15ms

Table 5.6.: Runtime of the *StrongSynchronizationConstraint* Monitor

$\mathcal{O}(|event| * tolerance)$ in terms of time and in $\mathcal{O}(|event| * tolerance)$ in terms of memory.
The output function checks, if the oldest stored (therefore unfulfilled) cluster is older than *tolerance*. This cluster is in the head of the list, therefore the output function is in $\mathcal{O}(1)$ in terms of time. The required delay is calculated by adding *tolerance* to the timestamp of the oldest stored unsatisfied cluster, subtracted by the timestamp of the current timestamp ($\mathcal{O}(1)$).
Similar to the previous constraint, the performance test was done with a trace of 3 streams, which fulfill the *StrongSynchronizationConstraint* with the attribute $tolerace = 3$. The performance of the *StrongSynchronizationConstraint* (average minimum: 5.86ms, average maximum: 854.73ms and overall average: 17.15ms) is significantly better than the performance of the *SynchronizationConstraint*. 58 events were processed per second in average. The performance is so much better, because each event can only be part of one synchronization cluster, therefore only one needs to be updated. Additionally, the output function only needs to check only needs to check the age of one cluster (which can be directly accessed as head of the stored list), not the age of every stored event, like in the previous constraint.

ExecutionTimeConstraint

The implementation of the *ExecutionTimeConstraint* is using TeSSLa's *runtime* operator on the *start* and *stop* events, which calculates the absolute runtime without any interruptions. The time of interruptions is also calculated by the calculated by this operator and then summed up. The sum of these interruptions is reseted by every *start* event.
TeSSLa's *runtime* operator subtracts the timestamps of the events of the second parameter (in this case *stop* and *resume*) from the timestamps of the events of the first parameter(*start* and *preempt*), therefore it stores the timestamps of the *start* and *preempt* events are stored, additionally to the sum the preemptions. For the output, the runtime can be calculated by subtracting the first application (with *start* and *stop* as parameters) of TeSSLa's *runtime* operator from the sum of the second applications (with *preempt* and *resumse* as parameters) of this operator. If the runtime should be checked in timestamps without a *stop* event, the second

	Minimum	Maximum	Average
run 1	0.09ms	905.29ms	5.24ms
run 2	0.15ms	858.29ms	5.14ms
run 3	0.09ms	748.94ms	5.08ms
average	0.11ms	837.51ms	5.15ms

Table 5.7.: Runtime of the *ExecutionTimeConstraint* Monitor

parameter of the first application of the *runtime* operator must be replaced by a current event. In the implementation this is done by merging all include streams and the delay stream. This runtime must be smaller or equal to *upper* in any point of time and greater or equal to *lower* at *stop* events. The required delay is calculated subtracting the runtime so far from upper. All of these operations are simple arithmetic functions, therefore the algorithm is in $\mathcal{O}(1)$ in terms of time. The required storage space is fixed, therefore it is also in $\mathcal{O}(1)$ in terms of memory.
The performance test was done by using a trace that fulfill the constraint with the attributes $lower = 50$ and $upper = 75$. For each execution, between 0 and 3 preemptions were introduced. The average minimal runtime was 0.11ms, the average maximal run time 837.51ms and the overall average was 5.15ms. Therefore, 194 events were processed per second in average.

OrderConstraint

The *OrderConstraint* is defined in a way, so that the number of events on the *source* stream is equal to the number of events on the *target* stream and that the i^{th} *source* event occurs before the i^{th} *target* event. The first described property can only be checked, when it is known that no further events will occur. In TeSSLa, it is generally unknown, if further events will occur, therefore the implementation has a third input stream, which requires to have exactly one event at the end of the observation.

The implementation counts the number of events on the *source* and *target* stream and checks, if the number of *source* events is larger or equal to the number of *target* events. In the end of the streams, the number of events on both streams must be equal. Therefore, the stored state consists of two integers and the algorithm is in $\mathcal{O}(1)$ in terms of memory. The incrementations of these counters and the comparison between them are simple arithmetic operations, therefore the state transition and the output function are both in $\mathcal{O}(1)$ in terms of time. The introduction of new timestamps is not required for this constraint, except the one defining the end of the observation, therefore no delay period must be calculated.
The trace for the performance analysis of this constraint contains 5.000 *source* and 5.000 *target* events, which fulfill the *OrderConstraint*. Additionally, the trace

	Minimum	Maximum	Average
run 1	0.06ms	804.98ms	3.44ms
run 2	0.05ms	829.19ms	3.44ms
run 3	0.08ms	805.71ms	3.45ms
average	0.06ms	813.23ms	3.44ms

Table 5.8.: Runtime of the *OrderConstraint* Monitor

contains one event ($endOfObservation$) in the end, which shows, that no further events will occur. The average minimal runtime was 0.06ms, the average maximum 813.23ms and the overall average 3.44ms. Therefore, 290 events were processed per second in average.

ComparisonConstraint

The *ComparisonConstraint* defines comparisons between timestamps. These functionalities are already defined in TeSSLa, therefore no implementation is given as part of this thesis.

SporadicConstraint

The *SporadicConstraint* is defined as simple application of the *Repetition-* and the *RepeatConstraint*, therefore the *SporadicConstraint* is also implemented as application of them. The implementations of the *Repetition-* and the *RepeatConstraint* are both in $\mathcal{O}(1)$ per event in terms of time, therefore the implementation of the *SporadicConstraint* is also in $\mathcal{O}(1)$ in terms of time. The implementations of the *Repetition-* and the *RepeatConstraint* are both in $\mathcal{O}(span)$ in terms of memory. The *span* parameters of both constraints are unused in the *SporadicConstraint* and therefore set to 1. Because of this, the implementation of the *SporadicConstraint* is $\mathcal{O}(1)$ in terms of memory.
The performance analysis was done by a trace, that fulfill the constraint with the attributes $lower = 100$, $upper = 150$, $jitter = 10$ and $minimum = 10$. The average minimum run time was 0.13ms, the average maximum 854.8ms and the overall average run time 10.49ms. Therefore, 95 events were processed per second in average.

	Minimum	Maximum	Average
run 1	0.13ms	877.67ms	10.55ms
run 2	0.16ms	926.91ms	10.54ms
run 3	0.09ms	757.67ms	10.39ms
average	0.13ms	854.08ms	10.49ms

Table 5.9.: Runtime of the *SporadicConstraint* Monitor

	Minimum	Maximum	Average
run 1	0.13ms	909.67ms	10.47ms
run 2	0.15ms	891.98ms	10.54ms
run 3	0.15ms	884.42ms	10.54ms
average	0.14ms	895.36ms	10.52ms

Table 5.10.: Runtime of the *PeriodicConstraint* Monitor

PeriodicConstraint

The *PeriodicConstraint* is defined as application of the *SporadicConstraint* and is also implemented like this. Because the *SporadicConstraint* is in $\mathcal{O}(1)$ in terms of memory and time, the *PeriodicConstraint* is also. The performance analysis of the *PeriodicConstraint* was done by monitoring a trace, which fulfills the constraint with the attributes $period = 30$, $jitter = 5$ and $minimum = 1$. The performance is similar to the performance of the previous constraint, the average minimum was 0.14ms, the average maximum 895.36ms and the overall average was 10.52ms. 95 events were processed per second in average.

PatternConstraint

The *PatternConstraint* is defined as application of the *Periodic-*, *Delay-* and *RepeatConstraint*. Because of the set of unknown timestamps X, the *Periodic-* and *DelayConstraint* cannot be used for the implementation. The set X is not used in the application of the *RepeatConstraint*, therefore its implementation is used as part of the output function.
The implementation is of the *RepeatConstraint* is in $\mathcal{O}(1)$ in terms of time and in $\mathcal{O}(span)$ in terms of memory. The *span* attribute is set to 1 in the application, therefore the memory usage is also constant.
In the implementation of the *PatternConstraint*, the lower and upper bound for the current timestamp of X is stored. At every event, these bounds are further enclosed, taking the previous known bounds and the bounds implied by the current event:

$$x \in X : time(event) - offset_{count(event) \bmod |offset|} - jitter \leq x$$

	Minimum	Maximum	Average
run 1	13.02ms	1225.35ms	15.43ms
run 2	12.75ms	1198.39ms	15.36ms
run 3	12.76ms	1122.87ms	15.3ms
average	12.84ms	1182.2ms	15.27ms

Table 5.11.: Runtime of the *PatternConstraint* Monitor

$$\leq time(event) - \mathit{offset}_{count(event) \bmod |\mathit{offset}|}$$

into account. The new lower bound is set by using the maximum of the previous lower bound and the lower bound implied by the current event, the new upper bound by using the minimum of the previous upper bound and the upper bound implied by the current event. At every $|\mathit{offset}|^{th}$ event, *period* is added to the current bounds. The output function checks, if the timestamp of the current timestamp is between the lower bound + $\mathit{offset}_{count(event) \bmod |\mathit{offset}|}$ and the upper bound plus $\mathit{offset}_{count(event) \bmod |\mathit{offset}|}$ plus $jitter$. The required delay is defined by the time distance between the current timestamp and the upper bound for X, plus the allowed offset of the following event, plus the allows deviation ($jitter$).
The only state stored in the implementation are the upper and lower bound for the current x-value, therefore the implementation itself is in $\mathcal{O}(1)$ in terms of memory, but the size of the $offset$-parameter, which is a hash map, is not limited and the complete algorithm, including the parameters, is $\mathcal{O}(|\mathit{offset}|)$ in terms of memory.

The performance of the implementation of the *PatternConstraint* was tested on a trace containing 10.000 events, which fulfill the constraint with the attributes $period = 50$, $offset = \{10, 20, 30\}$, $jitter = 5$ and $minimum = 1$. The average minimum runtime per event was 12.84ms, the average maximum 1282.2ms and the overall average 15.27ms. Therefore, 65.5 events were processed per second in average.

ArbitraryConstraint

The *ArbitraryConstraint* is defined as multiple applications of the *RepeatConstraint* and is also implemented this way. The number of applications of the *RepeatConstraint* is dependent on the number of elements in the *minimum* and *maximum* parameters. The runtime of the *RepeatConstraint* is in $\mathcal{O}(1)$ per application and event, therefore it the *ArbitraryConstraint* is in $\mathcal{O}(|minimum| = |minimum|)$ in terms of time. The memory usage of the *RepeatConstraint* is in $\mathcal{O}(span)$. In the application of the *RepeatConstraint*, the *span* parameter increases for each of the $|minimum| = |maximum|$ applications. Therefore, implementation is in

	Minimum	Maximum	Average
run 1	0.17ms	914.53ms	9.95ms
run 2	0.17ms	884.79ms	10.02ms
run 3	0.15ms	867.09ms	9.84ms
average	0.16ms	888.8ms	9.93ms

Table 5.12.: Runtime of the *ArbitraryConstraint* Monitor

	Minimum	Maximum	Average
run 1	0.17ms	866.26ms	7.61ms
run 2	0.08ms	991.31ms	7.63ms
run 3	0.08ms	1029.81ms	7.61ms
average	0.11ms	926.46ms	7.62ms

Table 5.13.: Runtime of the *BurstConstraint* Monitor

$\mathcal{O}(\sum_{i=1}^{|minimum|} i) = \mathcal{O}(\frac{|minimum|^2+|minimum|}{2})$ in terms of time.
The performance test was done by running the implementation on a trace containing 10.000 events, which follow the constraint with the attributes $minimum = \{10, 20, 30\}$ and $maximum = \{15, 25, 35\}$. The average minimum run time was 0.16ms, the average maximum 888.8ms and the overall average 9.93ms. Therefore, 100 events were processed per second in average.

BurstConstraint

The *BurstConstraint* is defined as twofold application of the *RepeatConstraint* and is also implemented this way. The *RepeatConstraint* is in $\mathcal{O}(1)$ in terms of time and because the *BurstConstraint* aplies it in a fixed number (2), the *BurstConstraint* is also in $\mathcal{O}(1)$ in terms of time. The memory usage of the *RepeatConstraint* is in $\mathcal{O}(span)$. The *span* parameter is *maxOccurrences* in the first and 1 in the second application, therefore the *BurstConstraint* is in $\mathcal{O}(maxOccurrences)$ in terms of memory.
The performance test was done by monitoring an trace of 10.000 Events, which formed 4.983 Bursts of the maximal length 10. Each of the bursts are containing 1 to 3 events, therefore the trace fulfill the *BurstConstraint* with the attributes $length = 10$, $maxOccurrences = 3$ and $minimum = 1$. The average minimum time was 11ms per event, the average maximum 926.46ms and the overall average 7.62ms. Therefore, 131 events were processed per second in average.

ReactionConstraint

The implementation of the *ReactionCostraint* stores two information as state. First, a map containing the color and the timestamp of each *stimulus* event, which did not have a matching *response* event yet and second, a set that contains all colors, that previously occurred in *response*. The state is updated at every input event. *Stimulus* events are inserted into the map, *response* events are inserted into the set. Additionally, *Stimulus* events are removed from the map, if *response* event with a matching color is occurring. Similar to the *DelayConstraint*(the *Reaction-Costraint* is an extension of the *DelayConstraint*, that additionally considers the color of events), the maximal number of entries in the map is the maximal number of *stimulus* events, that could possibly occur in an interval of the length *maximum*, which is *maximum*. The maximal possible size of the set that containing all previous *response* colors is the number of event, which previously occurred in *response*. Therefore, the algorithm is in $\mathcal{O}(maximum + count(response))$ in terms of memory. The state transition (insertion in map and in set, lookup and possibly remove in map) is in $\mathcal{O}(1)$ in terms of time.
The required delay is calculated by adding *maximum* to the timestamp of the oldest entry in the map mentioned above, and subtracting the current timestamp. Because the map is an unsorted hash map, every entry in the map has to be checked. Therefore, the calculation of the required delay is in the time complexity class $\mathcal{O}(maximum)$.
The output function checks, if the oldest entry in the map is not older than *maximum* and, at timestamps containing *stimulus* events, if the set of previous *response* colors contains the color of the current *stimulus* event. The lookup in the set is done in constant time, the search for the oldest entry in the map requires to check every entry, therefore the output function is in $\mathcal{O}(|maximum|)$ in terms of time.
The performance was tested on a trace containing 5.000 *stimulus* and 5.000 *response* events, which fulfill the *ReactionConstraint* with the attributes $minimum = 5$ and $maximum = 20$. The average minimum was 4.86ms, the average maximum 992.16ms and the overall average 9.27ms per event. Therefore, 107 events were processed in one second in average.
It should be noted that the required memory is dependent on the number of events. Therefore, the performance of the monitor will drop significantly, if the RAM of the system is completely filled and a swap file needs to be used.

AgeConstraint

The implementation of the *AgeConstraint* is similar to the implementation of the *ReactionCostraint*. The only difference between the stored states is, that in the implementation of the *AgeConstraint*, *stimulus* events are removed from the map,

	Minimum	Maximum	Average
run 1	4.87ms	1064.97ms	9.35ms
run 2	4.84ms	792.38ms	9.15ms
run 3	4.88ms	1119.12ms	9.31ms
average	4.86ms	992.16ms	9.27ms

Table 5.14.: Runtime of the *ReactionCostraint* Monitor

	Minimum	Maximum	Average
run 1	4.31ms	980.6ms	8.13ms
run 2	4.19ms	1009.3ms	7.98ms
run 3	4.19ms	973.66ms	8.09ms
average	4.23ms	987.85ms	8.07ms

Table 5.15.: Runtime of the *AgeConstraint* Monitor

when they are older than *maximum*, not when a matching *response* event is found. Therefore, the algorithm also is in $\mathcal{O}(maximum + count(response))$ in terms of memory and the state transition function is in $\mathcal{O}(1)$.
The creation of new timestamps is not needed in this constraint, because only previous events need to be considered, upcoming events not.
The output function checks in timestamps, which contain a *stimulus* event it is checked, if the color is in the set of colors, that previously occurred in *response*. In timestamps, which contain a *response* event, it is checked, if a *stimulus* event with the same color is in the map and if the time distance between them is greater or equal to *minimum* and smaller or equal to *maximum*. These operation are in $\mathcal{O}(1)$.
The trace for the performance test of the *AgeConstraint* implementation fulfills the constraint with the constraint with the attributes $minimum = 5$ and $maximum = 20$. It contains 5.000 *stimulus* and 5.000 *response* events. The performance is slightly better than the performance of the *ReactionCostraint* implementation. The average minimum runtime per event was 4.23ms, the average maximum 987.85ms and the overall average 8.07ms. 123 events were processed per second in average.
Like in the previous constraint, the required memory is dependent on the number of events and the performance will drop significantly at some point in time.

OutputSynchronizationConstraint

In the *OutputSynchronizationConstraint*, for each *stimulus* event, there must be one synchronization cluster of the length *tolerance*, in which each *response* stream must have at least one event of the same color as the *stimulus* event. There is no time distance between the cluster and the *stimlus* event defined, it just has to be before the end of the streams. Therefore, a additional event, which shows the end of the observation, is needed, similar to the *OrderConstraint*.

The implementation of the *OutputSynchronizationConstraint* is storing 5 different informations as state. First, a list of every color that occurred in *stimulus*. This is updated at every *stimulus* event by appending the color to the list(run time: $\mathcal{O}(1)$, memory: $\mathcal{O}(count(stimulus))$). Second, a map containing information about all synchronization clusters, that were not finished until this point in time is stored. This map is using the color attribute as key and the start time stamp and a map as value. This inner map contains a boolean variable for each *response* stream, which shows, whether there was an event for this synchronization cluster in this stream or not. This map is updated at every *response* event. For each *response* event, that occurred in this timestamp, it is checked, if a synchronization cluster with a matching color exists, if not a new synchronization cluster with the color of the event is created. The check per event (two lookups in maps) is done in constant time, therefore this update is in $\mathcal{O}(|response|)$ in terms of time. In worst cases, each event results in the creation of a new synchronization cluster, which must be stored at least for the length of *tolerance*. The size of each information about one synchronization cluster is linear dependent on the number of *response* streams and in each interval of the length *tolerance*, $tolerance * |response|$ events can occur (and create a new synchronization cluster), therefore this information is in $\mathcal{O}(tolerance * |response|^2)$ in terms of memory. The third stored information is similar to the second, but the clusters, that either older than tolerance or fulfilled are removed from the map. Therefore, the worst case memory consumption is the also ($\mathcal{O}(tolerance * |response|^2)$). To remove fulfilled clusters, it is checked for each cluster in the map, if there was at least one event in each *response* stream of the color of the cluster. Therefore, this update is in $\mathcal{O}(tolerance * |response|^2)$ in terms of time. The fourth stored information is a list of all colors, that had an fulfilled synchronization cluster in the *response* streams until this point in time. Appending items into a list is done in constant time. The number of fulfilled synchronization clusters is at most the number events in all *response* streams, divided by the number of the *response* streams. Therefore, the required memory of this information is in $\mathcal{O}\left(\frac{\sum_i count(response_i)}{|response|}\right)$. The last stored information is a set containing each color, that previously occurred in *response*. Inserting colors into this set is done in constant time and the size of this set is limited by the number of *stimulus* events.

The maximum of the time complexity classes from above define the time complexity class of the state transition function. Therefore, the state transition function is in $\mathcal{O}(tolerance * |response|^2)$. The maximum of the memory complexity classes, which defines the memory complexity of the algorithm, is $\mathcal{O}\left(\frac{\sum_i count(response_i)}{|response|}\right)$.

The required delay is calculated by adding *tolerance* to the start time of the oldest

		Minimum	Maximum	Average
	run 1			
content...	run 2			
	run 3			
	average			

Table 5.16.: Runtime of the *OutputSynchronizationConstraint* Monitor

unfinished cluster and subtracting the current timestamp ($\mathcal{O}(tolerance{*}|response|^2)$). The output function checks three things. First, all stored synchronization clusters must be either younger than *tolerance* or fulfilled. Second, the color of each *response* event in this timestamp (if existing) must previously have occurred in *stimulus* and third, the color of the *stimulus* event in this timestamp (if existing) did not occur previously in the *response* events.
Because the entries of the map, that stores the synchronization clusters, cannot be accessed in way, that is sorted by age, every entry of the map must be checked for age (at most $tolerance * |response|$ checks). For every synchronization cluster, that is older than *tolerance*, it must be checked, if this cluster is fulfilled. The check of a single cluster requires to check the boolean variables of each stream. Per timestamp, at most $|response|$ synchronization cluster can be started, therefore at most *response* clusters grow older than *tolerance* per timestamp. Therefore, this check is in $\mathcal{O}(tolerance^2 + tolerance * |response|)$. The check, if the color of each *response* event in this timestamp previously occurred in *stimulus* requires to compare each current *response* event color with each color in the *stimulus* color list ($\mathcal{O}(|response| * count(stimulus))$). Similarly, the check, if the color of the *stimulus* event did not occur previously in the *response* streams requires the comparison of the *stimulus* color with each entry of the *response* color list ($\mathcal{O}\left(\frac{\sum_i count(response_i)}{|response|}\right)$). Therefore, the output function is in $\mathcal{O}(|response| * count(stimulus))$ at timestamps with events. At the end of the observation, it must be checked, if each *stimulus* event had a matching synchronization cluster. For each of the at most $count(stimulus)$ *stimulus* colors, a lookup in a set must be doneTherefore, the output function is in ... in terms of time. The runtime and the required memory is dependent on the number of events, which previously occurred. Therefore, the runtime will increase by every incoming event.

InputSynchronizationConstraint

The *InputSynchronizationConstraint* is very similar to the *OutputSynchronization-Constraint*. The difference is, that the synchronization occurs in a set of *stimulus* events, not in *response* events.

	Minimum	Maximum	Average
run 1	17.75ms	1218.24ms	54.28ms
run 2	22.91ms	902.81ms	54.26ms
run 3	28.55ms	1118.14ms	54.10ms
average	23.07ms	1079.73ms	54.21ms

Table 5.17.: Runtime of the *InputSynchronizationConstraint* Monitor

Despite the similarities, monitoring the *InputSynchronizationConstraint* is simpler. Two information are stored as state. First, a map that uses the numbers 1 to $|stimulus|$ as keys and as values a second map that uses colors (integer) as key and the timestamp of the latest occurrence of this color in the stream (the stream is defined by the key of the outer map). This map is updated at every *stimulus* event, at which either the timestamp of the latest occurrence of this color in this stream is updated, or a inner map is created for this color. These operations (two lookups, possibly insert into map) are in $\mathcal{O}(1)$ in terms of time. The memory size of this information is in $\mathcal{O}(|stimulus| * count(stimulus))$.
The second stored information is a set, which contains every color, that occurred in *response*. This information is in $\mathcal{O}(count(response))$ in terms of memory and the corresponding state transition is in $\mathcal{O}(1)$.
The creation of new timestamps is not needed in this constraint, because only previous events need to be considered. Therefore, the calculation of a delay span is not required.
For the output function, two checks must be done. First, none of the colors of the stimulus events in this timestamps may have occurred previously in *response*. This is checked by comparing each of the current stimulus events with all of the entries of the *response* color set mentioned above. Therefore, this check is in The second test in the output function checks at timestamps containing a *response* event, if each stream had an event with this color if the latest *stimulus* events with the same color as the current *response* event form a valid synchronization cluster. For this, the oldest and youngest event with the same color as the *response* event in the map mentioned above is searched and compared. The difference must be smaller than *tolerance*.

The performance was tested by monitoring a trace of one response and three stimulus streams, which contained 2.500 synchronization clusters. The *tolerance* attribute was set to 5. The average minimum run time per event was 23.07ms, the average maximum 1079.73ms and the overall average 54.21ms. Therefore, 18 events were processed per second in average.
Like in the previous constraint, the runtime and the required memory is dependent on the number of events, which previously occurred. Therefore, the runtime will increase by every incoming event.

Summary

Table 5 gives an overview of the complexity classes and the runtimes per Event of the implementations. Most of them have a runtime under 20ms per event in the setup that has been used, therefore more than 50 events were processed per second in average. The *InputSynchronizationConstraint*, *SynchronizationConstraint* and *OutputSynchronizationConstraint* deviate from these results, with significantly greater runtimes (54ms, 452ms per event). Additionally it should be noted that the runtime per event and the overall memory usage is dependent on the number of events in the implementation of the *Reaction-*, the *Age-*, the *OutputSynchronization-* and the *InputSynchronizationConstraint*. Therefore, the runtime per event of the monitors of these constraints will grow when monitoring larger traces.

	Complexity(Memory)	Complexity(Runtime)	avg. Runtime per Event(on Pi)
DelayConstraint	$\mathcal{O}(upper)$	$\mathcal{O}(1)$	9.7ms
StrongDelayConstraint	$\mathcal{O}(upper)$	$\mathcal{O}(1)$	
RepeatConstraint	$\mathcal{O}(span)$	$\mathcal{O}(span)$	6.02ms
RepetitionConstraint	$\mathcal{O}(span)$	$\mathcal{O}(1)$	8.91ms
SynchronizationConstraint	$\mathcal{O}(\|event\| * tolerance)$	$\mathcal{O}(\|event\| * tolerance)$	452.39ms
StrongSynchronization Constraint	$\mathcal{O}(\|event\| * tolerance)$	$\mathcal{O}(\|event\| * tolerance)$	17.15ms
ExecutionTimeConstraint	$\mathcal{O}(1)$	$\mathcal{O}(1)$	5.15ms
OrderConstraint	$\mathcal{O}(1)$	$\mathcal{O}(1)$	3.44ms
SporadicConstraint	$\mathcal{O}(1)$	$\mathcal{O}(1)$	10.49ms
PeriodicConstraint	$\mathcal{O}(1)$	$\mathcal{O}(1)$	10.52ms
PatternConstraint	$\mathcal{O}(1)$	$\mathcal{O}(\|offset\|)^2$	15.27ms
ArbitraryConstraint	$\mathcal{O}(\sum_{i=1}^{\|minimum\|} i)$	$\mathcal{O}(\|minimum\|)$	9.93ms
BurstConstraint	$\mathcal{O}(maxOccurrences)$	$\mathcal{O}(1)$	7.62ms
ReactionConstraint	$\mathcal{O}(maximum + count(response))$	$\mathcal{O}(\|maximum\| + maximum)$	9.27ms
AgeConstraint	$\mathcal{O}(maximum + count(response))$	$\mathcal{O}(1)$	8.02ms
OutputSynchronization Constraint	$\mathcal{O}\left(\frac{\sum_i count(response_i)}{\|response\|}\right)$	$\mathcal{O}(tolerance^2 + \|response\| * (tolerance + count(stimulus)) + \left(\frac{\sum_i count(response_i)}{\|response\|}\right))$	
InputSynchronizationConstraint			54.21ms

6. Zusammenfassung und Ausblick

Die Zusammenfassung greift die in der Einleitung angerissenen Bereiche wieder auf und erläutert, zu welchen Ergebnissen diese Arbeit kommt. Dabei wird insbesondere auf die neuen Erkenntnisse und den Nutzen der Arbeit eingegangen.

Im anschließenden Ausblick werden mögliche nächste Schritte aufgezählt, um die Forschung an diesem Thema weiter voranzubringen. Hier darf man sich nicht scheuen, klar zu benennen, was im Rahmen dieser Arbeit nicht bearbeitet werden konnte und wo noch weitere Arbeit notwendig ist.

A. Anhang

Dieser Anhang enthält tiefergehende Informationen, die nicht zur eigentlichen Arbeit gehören.

A.1. Abschnitt des Anhangs

In den meisten Fällen wird kein Anhang benötigt, da sich selten Informationen ansammeln, die nicht zum eigentlichen Inhalt der Arbeit gehören. Vollständige Quelltextlisting haben in ausgedruckter Form keinen Wort und gehören daher weder in die Arbeit noch in den Anhang. Darüber hinaus gehören Abbildungen bzw. Diagramme, auf die im Text der Arbeit verwiesen wird, auf keinen Fall in den Anhang.

A.2. Event Feeder

List of Figures

List of Tables

Quelltextverzeichnis

Abkürzungsverzeichnis

TDO zu erledigen *T*o *D*o

Bibliography

[AUT17] AUTOSAR: *Virtual Functional Bus, 4.3.1.* `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_VFB.pdf`. Version: December 2017

[AUT18] AUTOSAR: Specification of Timing Extensions / AUTOSAR. 2018 (4.0). – Forschungsbericht

[Ber79] Berstel, Jean: *Transductions and Context-Free Languages -*. Wiesbaden : Vieweg+Teubner Verlag, 1979. – ISBN 978–3–519–02340–1

[BFL+12] Blom, Hans ; Feng, Dr. L. ; Lönn, Dr. H. ; Nordlander, Dr. J. ; Kuntz, Stefan ; Lisper, Dr. B. ; Quinton, Dr. S. ; Hanke, Dr. M. ; Peraldi-Frati, Dr. Marie-Agnès ; Goknil, Dr. A. ; Deantoni, Dr. J. ; Defo, Gilles B. ; Klobedanz, Kay ; Özhan, Mesut ; Honcharova, Olha: TIMMO2USE Language syntax, semantics, metamodel V2 / ITEA2. 2012 (1.2). – Forschungsbericht

[CHL+18] Convent, Lukas ; Hungerecker, Sebastian ; Leucker, Martin ; Scheffel, Torben ; Schmitz, Malte ; Thoma, Daniel: TeSSLa: Temporal Stream-Based Specification Language. In: Massoni, Tiago (Hrsg.) ; Mousavi, Mohammad R. (Hrsg.): *Formal Methods: Foundations and Applications*. Cham : Springer International Publishing, 2018. – ISBN 978–3–030–03044–5, S. 144–162

[DSS+05] D'Angelo, B. ; Sankaranarayanan, S. ; Sanchez, C. ; Robinson, W. ; Finkbeiner, B. ; Sipma, H. B. ; Mehrotra, S. ; Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, 2005, S. 166–174

[LN12] Lisper, Björn ; Nordlander, Johan: A Simple and flexible Timing Constraint Logic. In: *In 5th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), 15-18 October 2012, Amirandes, Heraklion, Crete.* (2012)

[LS09] Leucker, Martin ; Schallhart, Christian: A brief account of runtime verification. In: *The Journal of Logic and Algebraic Programming 78* (2009)

[LSS$^+$18] Leucker, Martin ; Sanchez, Cesar ; Scheffel, Torben ; Schmitz, Malte ; Schramm, Alexander: TeSSLa: runtime verification of non-synchronized real-time streams, 2018, S. 1925–1933