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
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 an intervention of pacemaker, that occurred too early or too late would risk the life of the patient. In Cyber-physical Systems, e.g. the Electronic Stability Control of a vehicle, wrong timing can also lead to property damage, injuries or deaths. Also environmental aspects are affected by timing cyber-physical systems, for example combustion engines need exact timing to produce as little emissions as possible.
The interconnection of several components in cyber-physical systems makes the design and analysis of the timing behavior of these systems significant hard, because not only the components on its own, but also the complete system must be considered. In this context, testing is a major problem, because it is hard to reproduce the exact state of the system, in which the error occurred. In many cases, the error does not lay in the component where it became visible, it was carried off to other parts, which results in a malfunctioning system, where it is extremely hard to find the bug that caused the problem. Online Monitoring is the key technique to address this problem, because you can isolate the error, without the need of storing and recreating the state of the system, when searching the error.
The goal of this thesis is to create a monitoring tool for the *AUTOSAR* (**AUT**omotive **O**pen **S**ystem **AR**chitecture) Timing Extensions, which were created to increase the interoperability and exchangeability of car components.

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, to increase interoperability, exchangebility and re-usability of parts and therefore simplify 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, how application has 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, for example the latency timing constraint, that describes the amount of time between two subsequent events, or the event triggering constraints, that are used to describe the timing behavior of events, that are created in one component without getting input from another component [AUT18]. An Event is a point in time, when something, that should be considered, happens. It is possible, to add a datatype to an event.
Problematic with the AUTOSAR Timing Extensions is, that the definitions are not very formal and have room left for interpretation. Let's take a look at the *BurstPatternEventTriggering*. The BurstPatternEventTriggerings describe events clusters, with events that occur with short time distances, with large time distances between the clusters. The following attributes are needed:

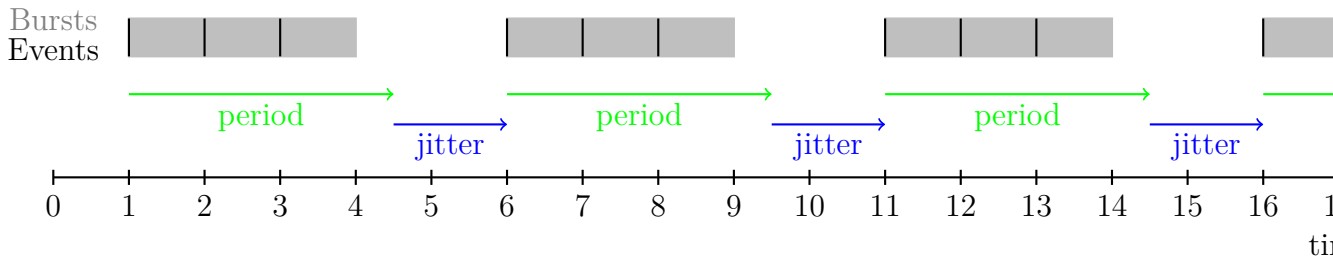

Figure 2.1.: BurstPatternEventTriggering Period-Jitter **accumulating**

Attribute	Type	Explanation
$maxNumberOfOccurrences$	PositiveInteger	maximum number of events per burst
$minNumberOfOccurrences$	PositiveInteger	(optional) minimum number of events per burst
$minimumInterArrivalTime$	TimeValue	minimum time distance between any two events
$patternLength$	TimeValue	length of each burst
$patternPeriod$	TimeValue	(optional) Time distance between the start points of two subsequent bursts
$patternJitter$	TimeValue	(optional) maximum of allowed deviation from periodic pattern

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

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 also defined as „distance between subsequent occurrences of the event“in

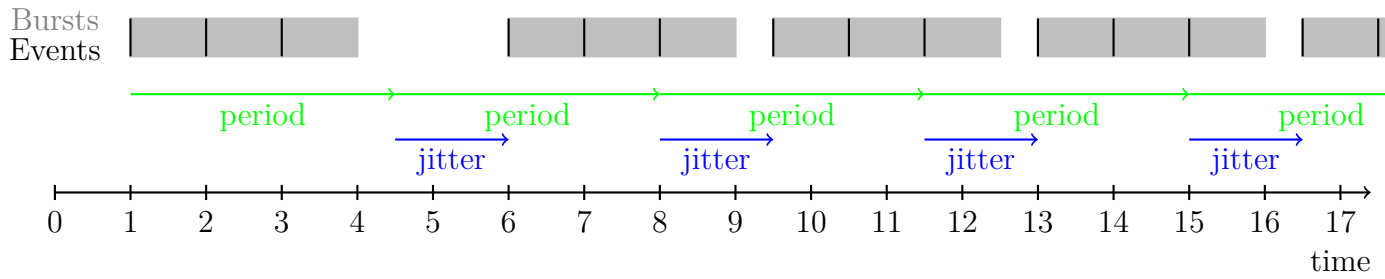

Figure 2.2.: BurstPatternEventTriggering Period-Jitter **non-accumulating**

the text, hence it is understandable the accumulating way, but there is also 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 impossible (more on monitorability in 4). As example, we will use the burst pattern again, this time using the attributes as

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

In 2.3 you see the application of the BurstPatternEventTriggering Constraint with the given parameters on a stream with events at the timestamps 3, 3.5, 4, 4.5. You can see the development of possible the burst cluster with ongoing time. The gray lines show, where the burst 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 there are 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 impossible in general, because of the strict resource limitations. More on that in 4.

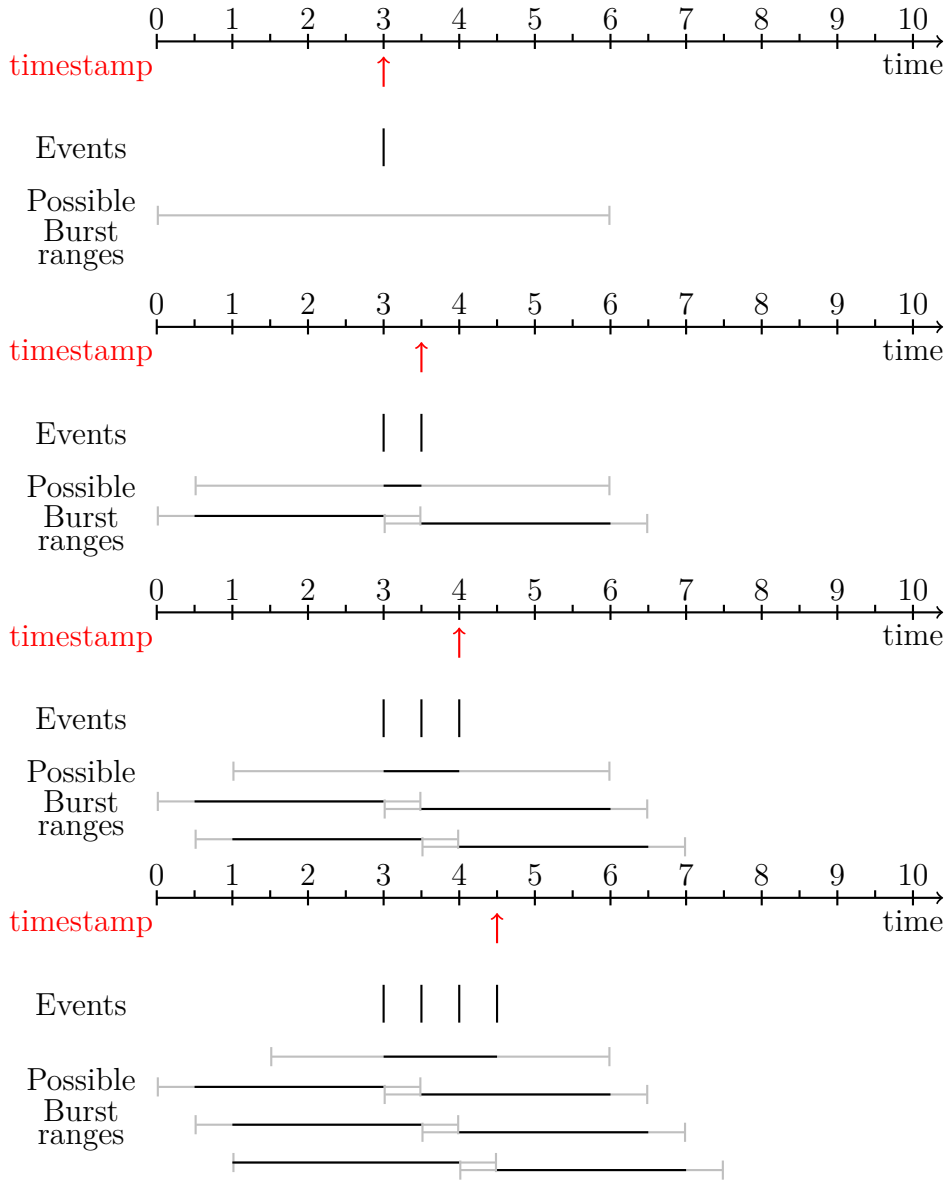

Figure 2.3.: BurstPatternEventTriggering Possible bursts, ↑ shows the current time

2.2. TADL2

Because of the problems in the definitions and monitorability of the AUTOSAR Timing Extensions, the implementation of a monitor will be done on the TADL2 (Timing Augmented Description Language Version 2) Timing Constraints, which were defined in the TIMMO-2-USE project, which was developed as part of the EAST-ADL (Electronics Architecture and Software Technology-Architecture Description Language). EAST-ADL 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 an formalized 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 define sub- or subersets of each other. Many of the AUTOSAR Timing Extensions can be defined via a combination of TADL2 Constraints, as you will see in 2.2.3.

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 value as timestamp, without the possibility of adding a data value. There are 7 syntactic categories in TiCL

$\mathbb{R}$(arithmetic constants)
$Avar$(arithmetic variables)
$AExp$(arithmetic expressions)

$Svar$(set variables)
$SExp$(set expressions)

$TVar$(time variables)
$CExp$(constraint expressions)

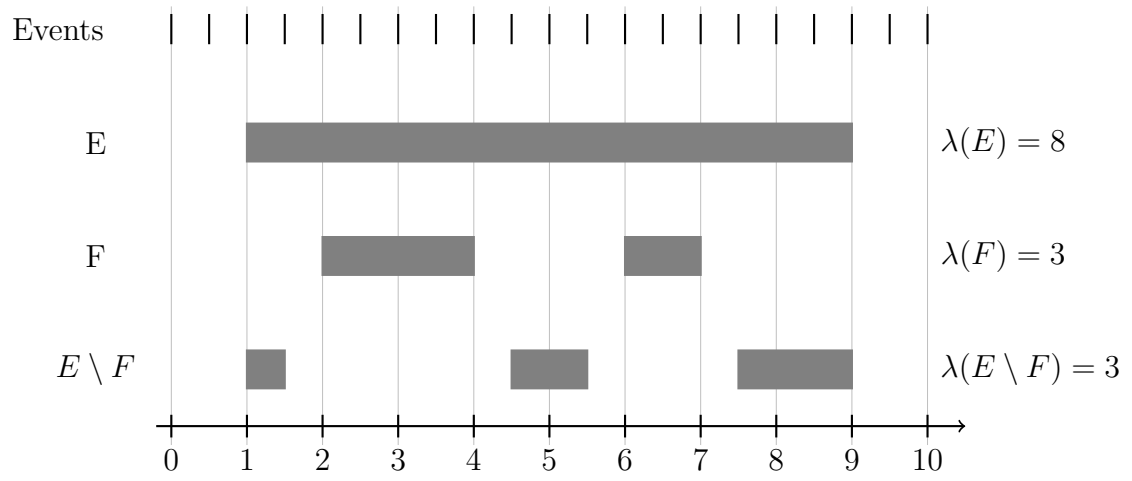

Figure 2.4.: Graphical example of $\lambda(E), \lambda(F)$ and $\lambda(E \setminus F)$

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 Lebesue measure, which is figuratively speaking, the length of all continuous intervals of E. In 2.4 you see a visualized example of the measure operator λ. 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 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, 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, including x, and before 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 $\{y : forallx \in E : x < y\}$, 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 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.

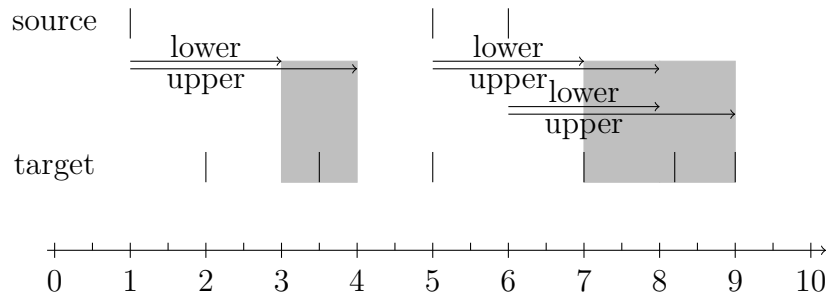

Figure 2.5.: Example DelayConstraint - $lower = 2$, $upper = 3$

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* are allowed.
In 2.5 you see 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	$\mathbb{T}$ (time expression)

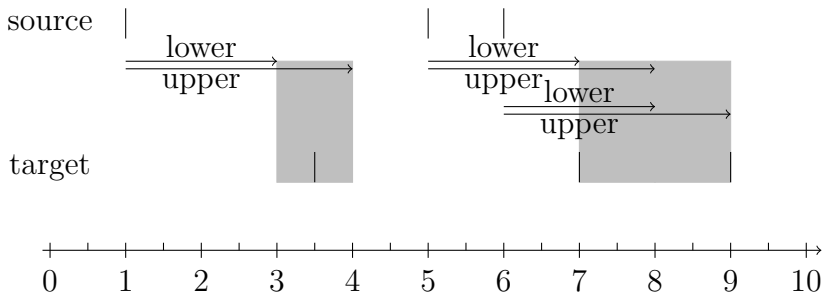

Figure 2.6.: Example StrongDelayConstraint - $lower = 2$, $upper = 3$

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. In 2.6 you see 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	$\mathbb{T}$ (time expression)
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. The definition specifies that the length of each time interval containing

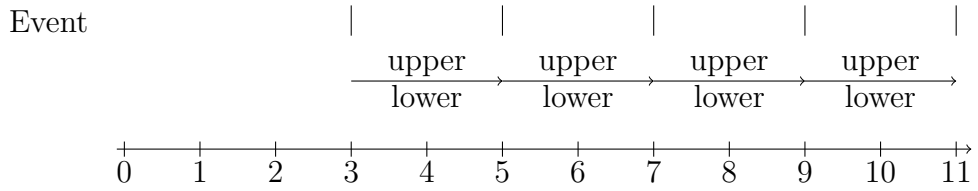

Figure 2.7.: Example RepeatConstraint - $lower = 2$, $upper = 2$, $span = 1$

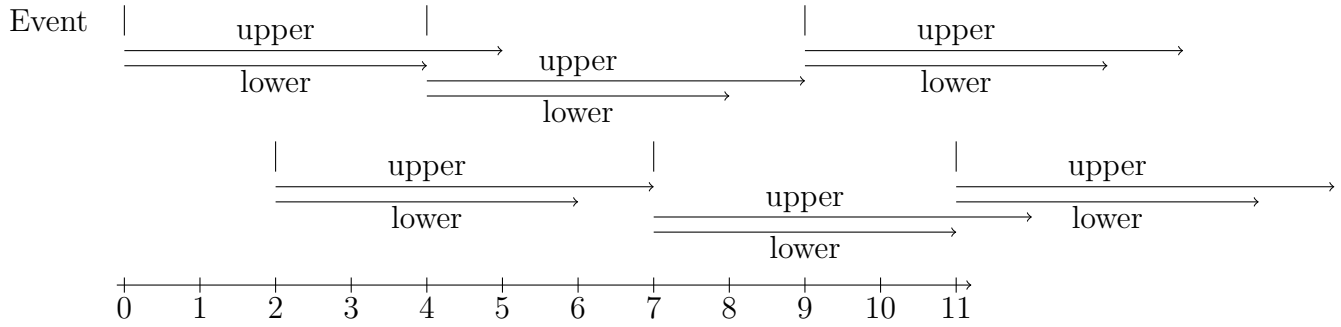

Figure 2.8.: Example RepeatConstraint - $lower = 4$, $upper = 5$, $span = 2$

$span + 1$ 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.
In 2.7 you see an example of the RepeatConstraint with the attributes $event = \{1, 3, 5, ...\}$, $lower = upper = 2$ and $span = 1$. Because *lower* is equal *upper* and *span* is 1, the events are following a strict periodic pattern after the first event. 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	$\mathbb{T}$ (time expression)
upper	$\mathbb{T}$
span	*integer*

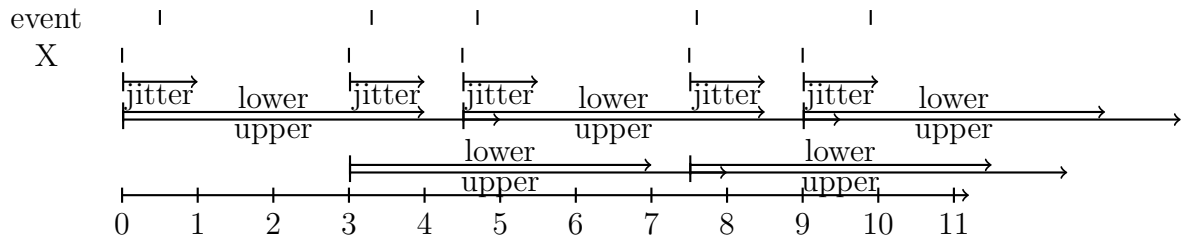

Figure 2.9.: Example RepetitionConstraint - $lower = 4$, $upper = 5$, $span = 2$, $jitter = 1$

$$\begin{array}{rl} jitter & \mathbb{T} \end{array}$$

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

$$\begin{aligned} &\exists X : RepeatConstraint(X, lower, upper, span) \wedge \\ &\quad StrongDelayConstraint(X, event, 0, jitter) \end{aligned}$$

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. In 2.9 you see 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

$$\begin{array}{rl} event & \text{set of event sets, } |event| \geq 2 \\ tolerance & \mathbb{T} \end{array}$$

and is defined via the *DelayConstraint* as

$$\begin{aligned} &\exists X : \forall i : DelayConstraint(X, event_i, 0, tolerance) \wedge \\ &\quad DelayConstraint(event_i, X, -tolerance, 0) \end{aligned}$$

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

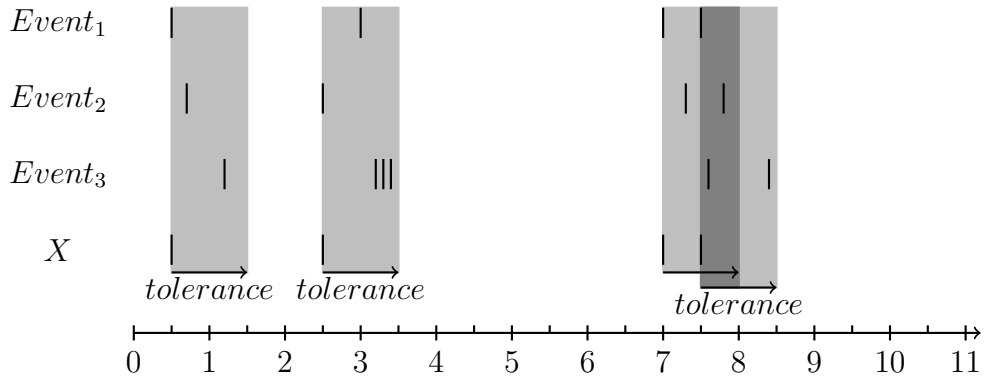

Figure 2.10.: Example SynchronizationConstraint - $tolerance = 1$

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. In the second cluster of events you see 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 *SynchronizationConstraint*, 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 you can see in 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.

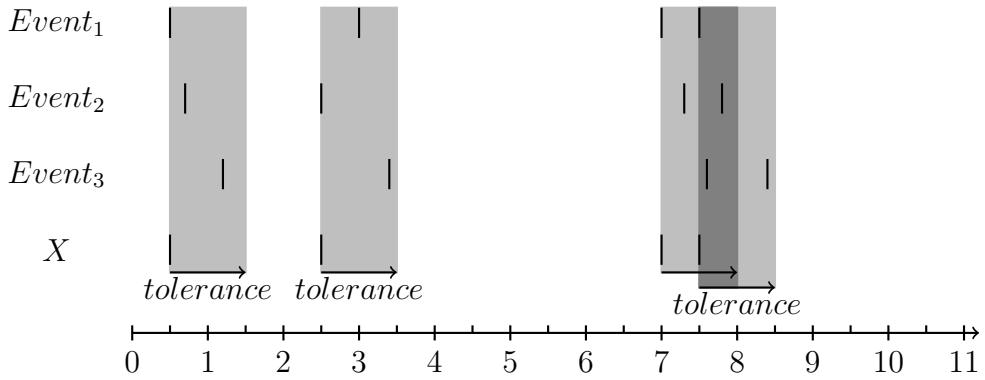

Figure 2.11.: Example StrongSynchronizationConstraint - $tolerance = 1$

ExecutionTimeConstraint

The *ExecutionTimeConstraints* takes 4 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. 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.

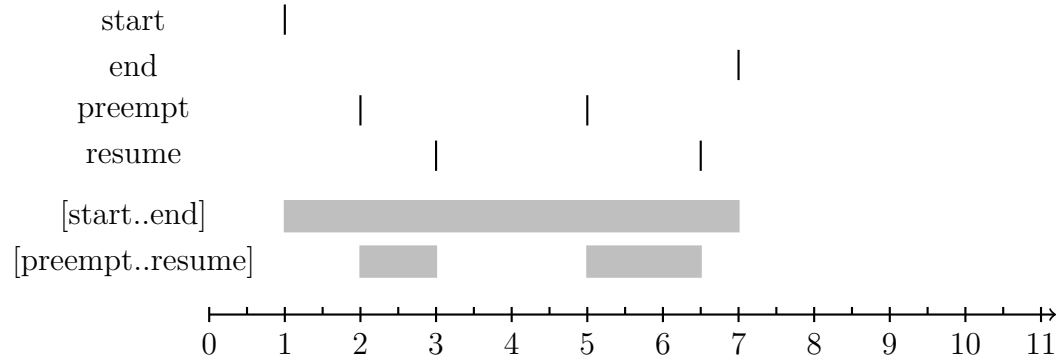

Figure 2.12.: Example ExecutionTimeConstraint

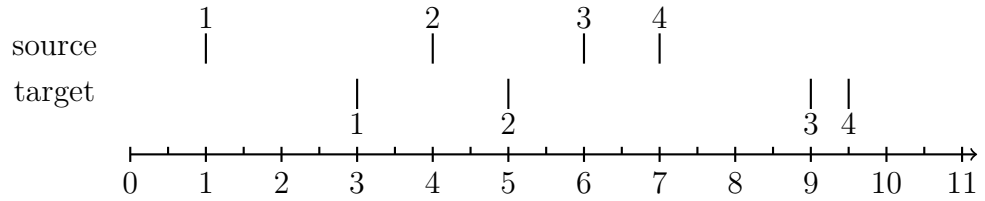

Figure 2.13.: Example OrderConstraint

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.
In 2.13 you see 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 for both sets 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 timing expressions. It takes 3 attributes

$$\begin{aligned} \mathit{leftOperand} &\quad \mathbb{T} \\ \mathit{rightOperand} &\quad \mathbb{T} \\ \mathit{operator} &\quad \text{comparisonOperator}(\in \{\mathit{LessThanOrEqual}, \mathit{LessThan}, \\ &\qquad\qquad \mathit{GreaterThanOrEqual}, \mathit{GreaterThan}, \mathit{Equal}\}) \end{aligned}$$

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

$\mathit{ComparisonConstraint}(\mathit{leftOperand}, \mathit{rightOperand}, \mathit{LessThanOrEqual})$
$\Leftrightarrow \mathit{leftOperand} \leq \mathit{rightOperand}$

$\mathit{ComparisonConstraint}(\mathit{leftOperand}, \mathit{rightOperand}, \mathit{LessThan})$
$\Leftrightarrow \mathit{leftOperand} < \mathit{rightOperand}$

$\mathit{ComparisonConstraint}(\mathit{leftOperand}, \mathit{rightOperand}, \mathit{GreaterThanOrEqual})$
$\Leftrightarrow \mathit{leftOperand} \geq \mathit{rightOperand}$

$\mathit{ComparisonConstraint}(\mathit{leftOperand}, \mathit{rightOperand}, \mathit{GreaterThan})$
$\Leftrightarrow \mathit{leftOperand} > \mathit{rightOperand}$

$\mathit{ComparisonConstraint}(\mathit{leftOperand}, \mathit{rightOperand}, \mathit{Equal})$
$\Leftrightarrow \mathit{leftOperand} = \mathit{rightOperand}$

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

SporadicConstraint

The *SporadicConstraint* takes 5 attributes

$$\begin{aligned} \mathit{event} &\quad \text{set of events} \\ \mathit{lower} &\quad \mathbb{T} \\ \mathit{upper} &\quad \mathbb{T} \\ \mathit{jitter} &\quad \mathbb{T} \\ \mathit{minimum} &\quad \mathbb{T} \end{aligned}$$

and is defined via the *Repetition-* and the *RepeatConstraint* as

$\mathit{RepetitionConstraint}(\mathit{event}, \mathit{lower}, \mathit{upper}, 1, \mathit{jitter})$

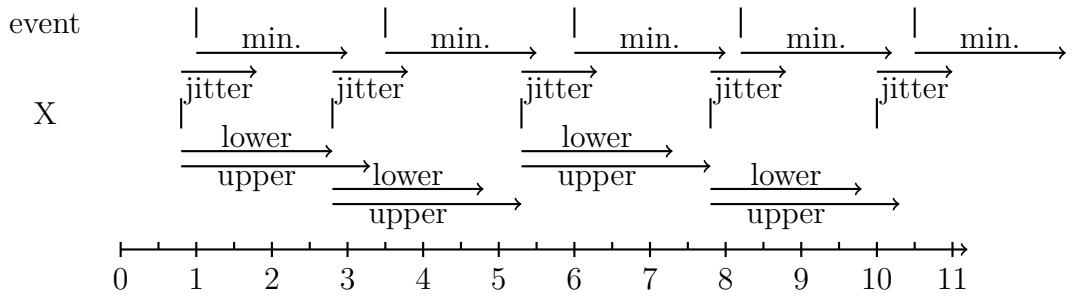

Figure 2.14.: Example SporadicConstraint - $lower = 2$, $upper = 2.5$, $jitter = 1$, $minimum = 2$

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

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. For each point of time in X, there must be exactly one timestamp in *event*, that is not before the corresponding entry of X and not later than *jitter* after that.
2.14 shows a possible utilization of the *SporadicConstraint* with the attributes $lower = 2$, $upper = 2.5$, $jitter = 1$, $minimum = 2$ and $event = \{1, 3.5, 6, 8.2, 10.5, ...\}$. As 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 bijective mapping between the elements of *event* and X.
In 2.15, the *PeriodicConstraint* with the attributes $period = 3$, $jitter = 1$, $minimum =$

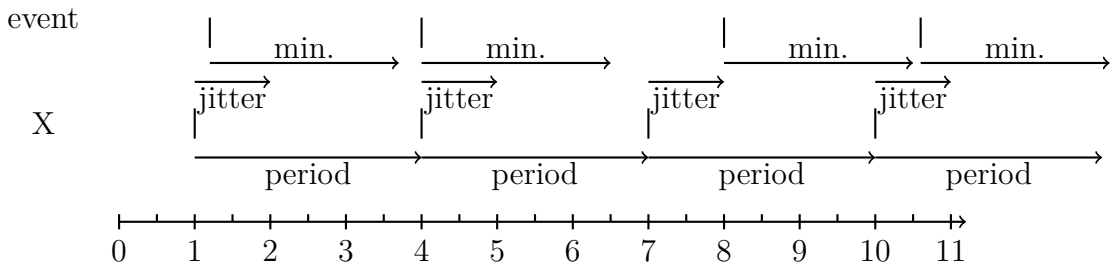

Figure 2.15.: Example PeriodicConstraint - $period = 3$, $jitter = 1$, $minimum = 2.5$

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	$setof\mathbb{T}$
jitter	$\mathbb{T}$
minimum	$\mathbb{T}$

and is defined as

$$\exists X : PeriodicConstraint(X, period, 0, 0)$$
$$\wedge \forall i : DelayContraint(X, event, offset_i, offset_i + jitter)$$
$$\wedge RepeatConstraint(event, minimum, \infty, 1)$$

It can be understood as a modification of the *PeriodicConstraint*, as it describes periodic behavior, but not from single events, but from groups of $|offset|$ events, that follow specific time distances (specified by $offset$) after the strictly periodic timestamps of X.
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*.

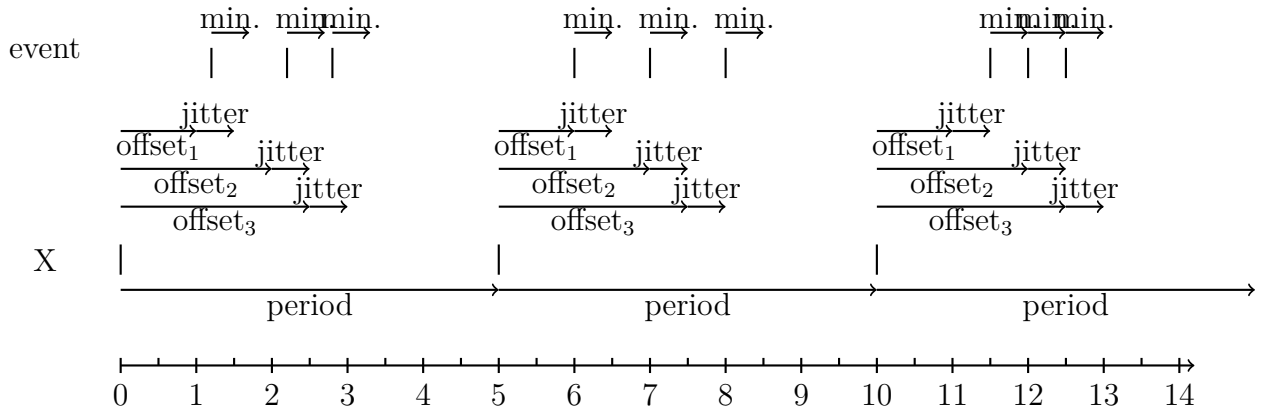

Figure 2.16.: Example PatternConstraint - $period = 5$, $offset = \{1, 2, 2.5\}$, $jitter = 0.5$, $minimum = 0.5$

ArbitraryConstraint

The *ArbitraryConstraint* takes 3 attributes

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

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 several subsequent events. The first entries of *minimum* and *maximum* define the distance between each event and it direct successor. The second entries, where the *span* attribute of the *RepeatConstraint* is 2, set the distance between the one event and its next but one successor and so on.
2.17 show 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 2.1, the relevant distances are written in **bold** font. Apparently, the time distances math the ranges, given by the *minimum*- and *maximum* attribute.

	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 2.17

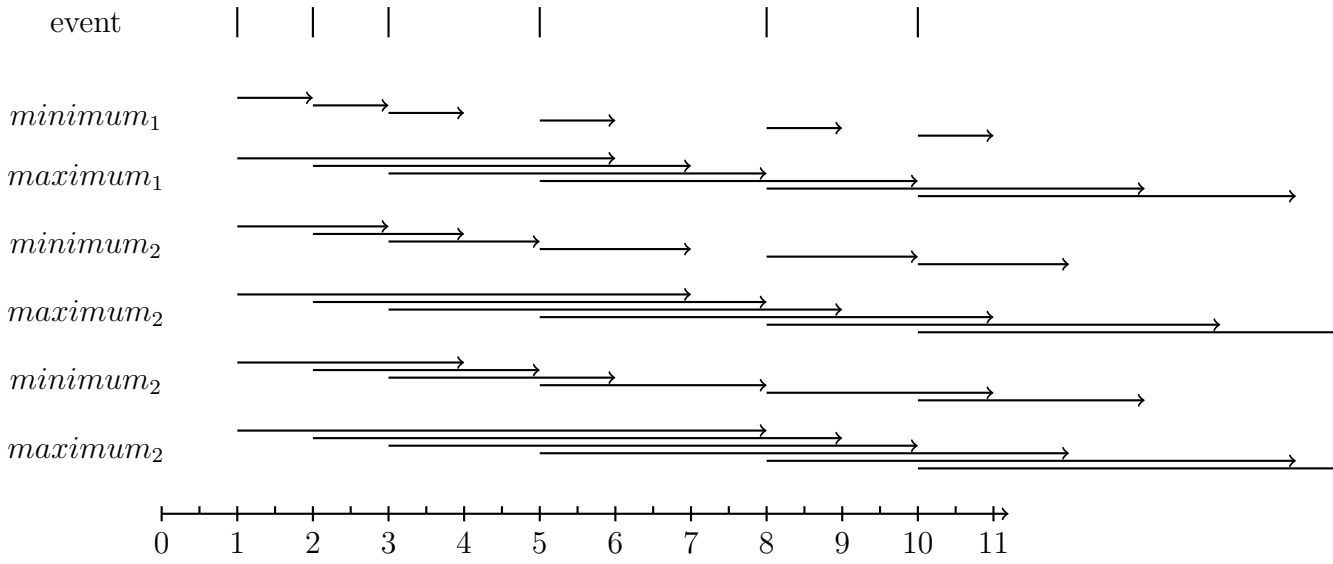

Figure 2.17.: Example ArbitraryConstraint - $period = 5$, $offset = \{1, 2, 2.5\}$, $jitter = 0.5$, $minimum = 0.5$

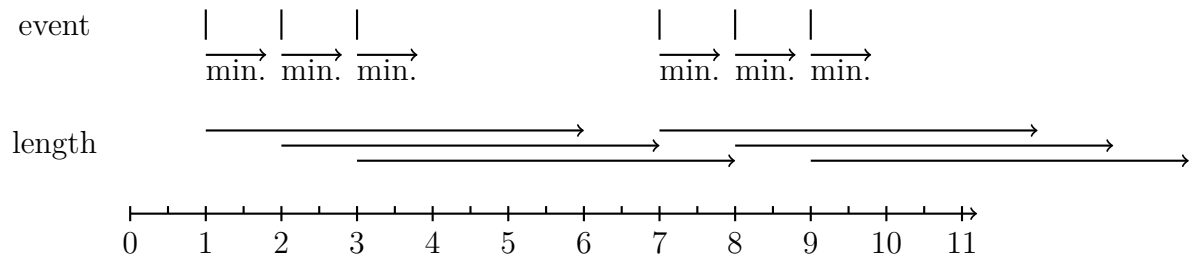

Figure 2.18.: Example BurstConstraint - $length = 5$, $maxOccurences = 3$ $minimum = 0.8$

BurstConstraint

The *BurstConstraint* takes 4 attributes

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

and is defined as

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

The idea of this constraint is to describe the maximum number of events, that may occur in an time interval of the given *length* and 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 2.2.3.
In 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

scope	*EventChain*
minimum	$\mathbb{T}$
maximum	$\mathbb{T}$

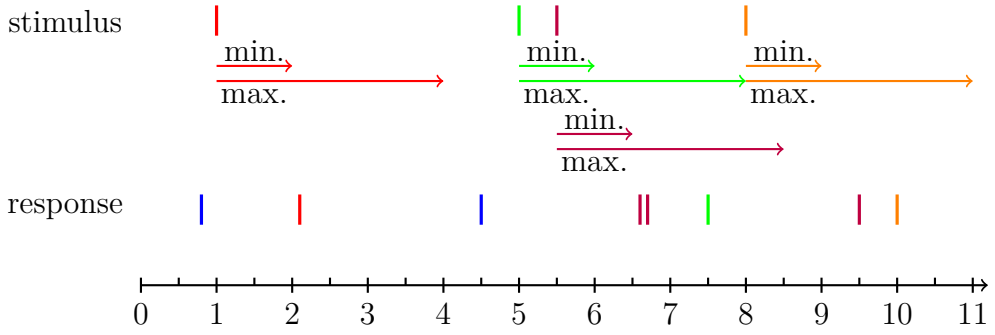

Figure 2.19.: Example ReactionConstraint - $minimum = 1$, $maximum = 3$

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 all events 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 each 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 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

$scope$	$EventChain$
$minimum$	$\mathbb{T}$
$maximum$	$\mathbb{T}$

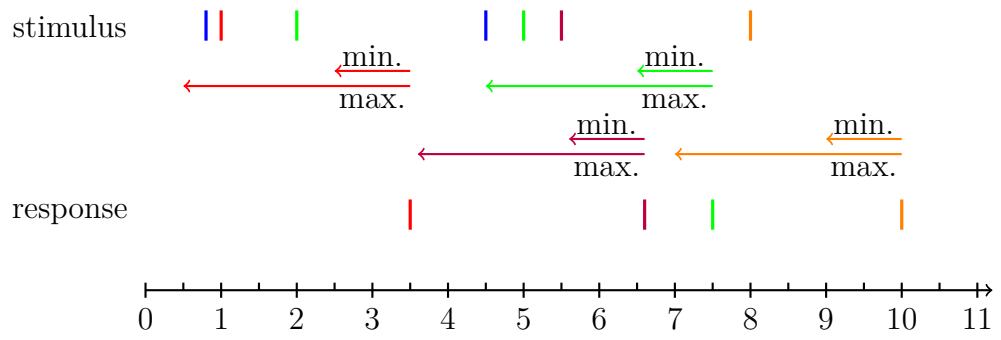

Figure 2.20.: Example AgeConstraint - $minimum = 1$, $maximum = 3$

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 alternative 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. 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*.

OutputSynchronizationConstraint

The *OutputSynchronizationConstraint* takes 2 attributes

$$\begin{aligned}
scope &\quad SetofEventChain \\
tolerance &\quad \mathbb{T}
\end{aligned}$$

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

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

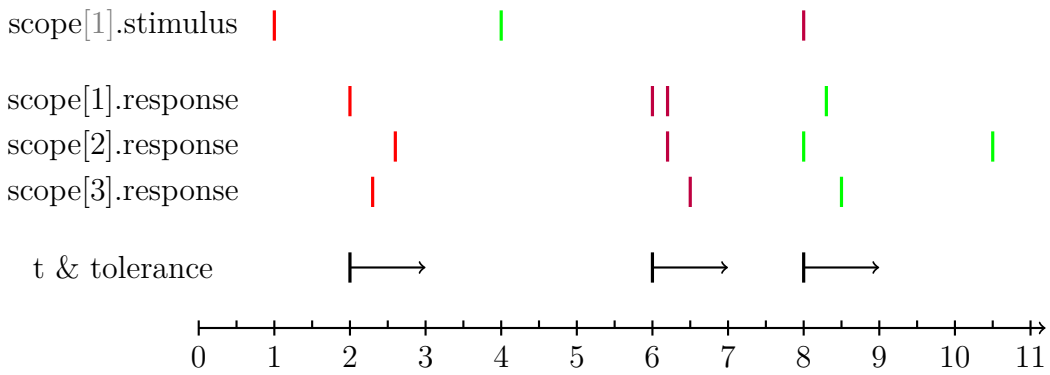

Figure 2.21.: Example OutputSynchronizationConstraint - $tolerance = 1$

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. 2.21 shows an example of the *OutputSynchronizationConstraint* with the attributes $tolerance = 1$,
$scope[1].stimulus = scope[2].stimulus = scope[3].stimulus = \{(1, red), (4, green), (8, purple)\}$,
$scope[1].response = \{(2, red), (6, purple), (6.2, purple), (8.2, green)\}$,
$scope[2].response = \{(2.6, red), (6.2, purple), (8, green), (10.5, green)\}$,
$scope[3].response = \{(2.3, red), (6.5, purple), (8.5, green)\}$.
As you can see, only time distance between the *response* events are relevant, the distance between *stimulus* and *response* is not observed and may also be negative, as you can see in the purple events.

InputSynchronizationConstraint

The *InputSynchronizationConstraint* takes 2 attributes

$$\begin{aligned} scope &\quad SetofEventChain \\ tolerance &\quad \mathbb{T} \end{aligned}$$

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.

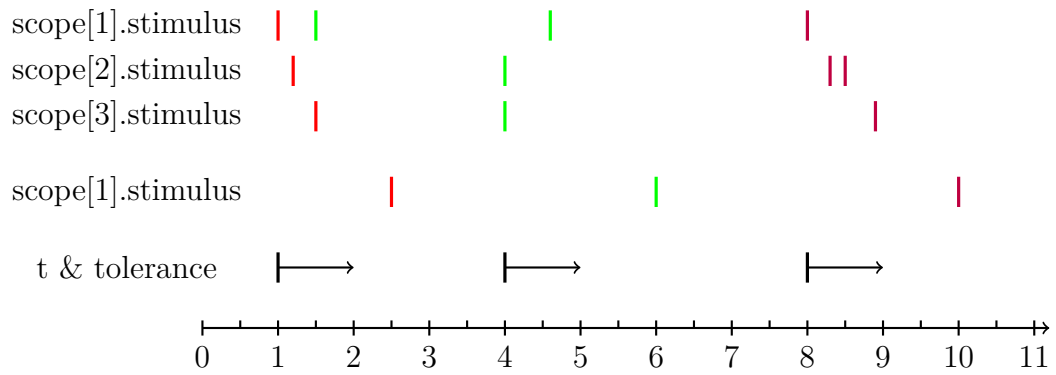

Figure 2.22.: Example InputSynchronizationConstraint - $tolerance = 1$

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.

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 *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 are in *BurstPatternEventTriggering* and not in *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* is the same for the *LatencyTimingConstraint* with the *latencyConstraintType age*, but scope is defined with the type *TimingDescriptionEvent*, not as *TimingDescriptionEventChain*. The reason for this is, that it can be used in earlier development phases, as it is an abstract type. There is no differentiation like this in TADL2 and the *AgeConstraint* is semantically the same in AUTOSAR and TADL2.

SynchronizationTimingConstraint

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

SynchronizationPointConstraint

The *SynchronizationPointConstraint* describes, that a list of executables and a list 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 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

event Occurrence-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) ∧ SynchronizationConstraint (scope.response, tolerance)
single Occurrences	scope	response Synchronization	tolerance	OutputSynchronizationConstraint (scope, tolerance) ∧ StrongSynchronizationConstraint (scope.response, tolerance)
multiple Occurrences	scope	stimulus Synchronization	tolerance	InputSynchronizationConstraint (scope, tolerance) ∧ SynchronizationConstraint (scope.stimulus, tolerance)
single Occurrences	scope	stimulus Synchronization	tolerance	InputSynchronizationConstraint (scope, tolerance) ∧ SynchronizationConstraint (scope.stimulus, tolerance)

Table 2.2.: SynchronizationTimingConstraint ⇔ TADL2 Constraints

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

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, but the TADL2 constraint is capable of not counting interruptions in the execution. On the other hand, the AUTOSAR constraint offers the possibility, to determine if external calls should be included in the run time or not. Also, 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 times need to be turned into their obvious counterparts. If the run time of external calls should not be counted (the *executionTimeType* attribute is set to *net*), the start and stop points of these external calls must be moved into the *preempt* and *resume* event sets. If the *executionTimeType* attribute is set to *gross*, external calls are ignored and the complete time from the start to the end of the execution is counted, therefore the *preempt* and *resume* event remain empty.

3. Monitor Models

3.1. Runtime Verification

Monitoring the AUTOSAR Timing Extensions is the goal of this thesis. As monitoring plays a major role in runtime verification, a short overview of this will be given. The definitions of [Run] 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. 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 4.

3.2. TeSSLa

TeSSLa (**Te**mporal **S**tream-based **S**pecification **La**nguage) is a functional programming language, build for runtime verification of streams. In TeSSLa, **streams** are defined as traces of events, each event consists of one data value from a data set $\mathbb{D}$ and a time value from a discrete time domain $\mathbb{T}$. This time domain needs a total order and subsequent timestamps must have increasing time values. A TeSSLa Specification can have several streams with different data sets, but each of these streams must use the same time domain $\mathbb{T}$, which timestamps are increasing over all streams. Each stream can have only one event per timestamp, but it is possible to have events on different streams at the same timestamp.
A distinction between synchronous and asynchronous streams is made. A set of synchronous streams have events in the exact same time stamps, events in asynchronous streams do not have this restriction. It is easy to see, that synchronous streams are

a subset of the asynchronous ones, therefore we will only use asynchronous streams from now on.
In TeSSLa, calculations 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. This possibility will take a large role in this thesis, more on that later.
At the timestamps, in which events arrived and calculations are done, you only have direct access to the youngest event of each stream, but with the use of the *last*-operator, which can be used recursively, the event before that can be accessed. The *lift*-operator applies a function, which is defined on data values $\mathbb{D}$, on each event of one or more streams. Similar to this, the *slift*-operator (signal lift) first applies the given function, when there was at least one event of each input stream. The *time*-operator returns the time value of an event.

3.3. Finite Transducers

4. Monitorability

Because of time and memory restrictions, online monitoring on a possible infinite run of a system is only reasonable, if several constraints are fulfilled. Therefore, the term of *Finite Monitorability* will be introduced, which ensures, that a property can be supervised using an online monitor.

4.1. Finite Monitorability

4.1.1. Timestamps

As we consider streams that can be infinite, 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 meet, especially not in the context of online monitoring. Therefore, the time domain $\mathbb{T}$ must be 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 constraints.

4.1.2. Finite Monitorability

For the definitions of streams and functions defined on them, TeSSLa-like syntax is used. Also, some standard TeSSLa functions are used in the definitions.

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

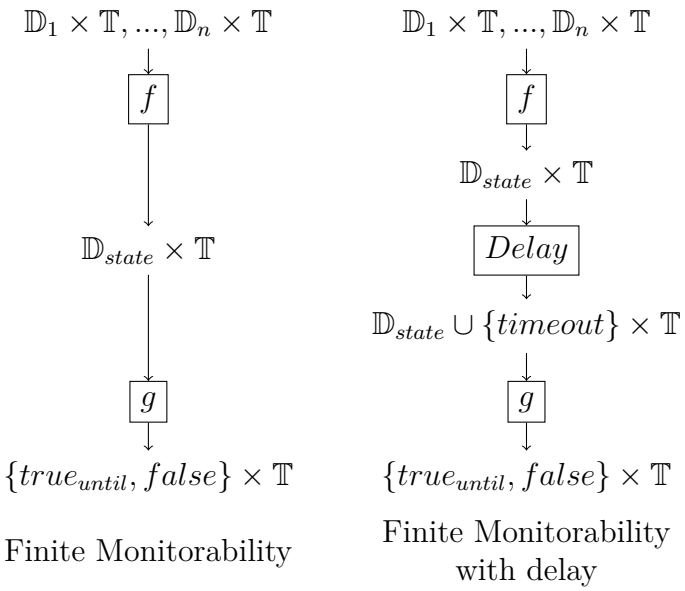

Figure 4.1.: Overview Finite Monitorability - with or without *delay*

Input Streams

Let $S_1, S_2, ..., S_n$ be input streams with
$\forall i : S_i = (\mathbb{T} \cdot \mathbb{D}_i)^\omega \cup (\mathbb{T} \cdot \mathbb{D}_i)^+ \cup (\mathbb{T} \cdot \mathbb{D}_i)^* \cdot (\mathbb{T}_\infty \cup \mathbb{T} \cdot \{\bot\})$ and
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 finite size.
Further let $f : S_1 \times S_2 \times ... \times S_n \times S_{state} \to S_{state} \times \mathbb{T}$ a state transition function, which defines the state stream in an incremental fashion:
$\forall t \in \mathbb{T} \exists i \in \{1, 2, ..., n\} : S_i(t) \in \mathbb{D}_i$
$\to S_{state}(t) = f(S_1(t), S_2(t), ..., S_n(t), last(S_{state}, merge(S_1, S_2, ..., S_n))(t))$
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

$g : \mathbb{D}_{state} \times \mathbb{T} \rightarrow \{true_{until}, false\} \times \mathbb{T}$
The runtime of g is in $\mathcal{O}(1)$.

Evaluation

A property of a set of streams is called *Finite Monitorable*, if a function f with type $\mathbb{D}_{state}$ 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. It should be noted that these definitions are *timestamp conservative*, because the streams S_{state} and S_{output} can only change their data value at the timestamps of input events.

Equivalences

4.1.3. Finite Monitorability with Delay

Not all of the TADL2 constraints can be monitored in a *timestamp conservative*. 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 still be $true_{until}$, 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.

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.

Delay

A *delay generator*, which copies the states produced by the function f and inserts a *timeout* state into the state stream, if there was no input event for a specific period of time, is added to the definition. The length of this period depends on the current state of the monitor.

Output Stream

The output function g is expanded by the *timeout* value:
$g : (\mathbb{D}_{state} \cup \{timeout\}) \times \mathbb{T} \rightarrow \{true_{until}, false\} \times \mathbb{T}$
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 with 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.

4.1.4. Non-Finite Monitorability

Not all TADL2 constraints are finite monitorable, because a monitor would require infinite memory and/or time resources. In a theoretical view, this makes online monitoring on infinite traces impossible, because a machine with infinite resources does not exist in the real world. In a practical view, many of these problems are solved by using a system with finite memory, with the hope that this finite resources would be enough, to cover the inputs of the ”real world”. In these cases, a distinction is useful, as some constraints have resource requirements, that grow continuously with every input event, and others constraints only require infinite resources in worst case scenarios. 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.

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

5. Timmo2Use Constraints

5.1. DelayConstraint

5.2. StrongDelayConstraint

5.3. RepeatConstraint

5.4. RepetitionConstraint

5.5. SynchronizationConstraint

5.6. StrongSynchronizationConstraint

5.7. ExecutionTimeConstraint

5.8. OrderConstraint

5.9. ComparisonConstraint

5.10. SporadicConstraint

5.11. PeriodicConstraint

5.12. PatternConstraint

5.13. ArbitraryConstraint

5.14. BurstConstraint

5.15. ReactionConstraint

5.16. AgeConstraint

5.17. OutputSynchronizationConstraint

6. Implementierungen

In der Evaluierung wird das Ergebnis dieser Arbeit bewertet. Eine praktische Evaluation eines neuen Algorithmus kann zum Beispiel durch eine Implementierung geschehen. Je nach Thema der Arbeit kann sich natürlich auch die gesamte Arbeit eher im praktischen Bereich mit einer Implementierung beschäftigen. In diesem Fall gilt es am Ende der Arbeit insbesondere die Implementierung selber zu evaluieren. Wesentliche Fragen dabei können sein:

– Was funktioniert jetzt besser als vor meiner Arbeit?
– Wie kann das praktisch eingesetzt werden?
– Was sagen potenzielle Anwender zu meiner Lösung?

6.1. Implementierungen

Wenn Implementierungen umfangreich beschrieben werden, ist darauf zu achten, den richtigen Mittelweg zwischen einer zu detaillierten und zu oberflächlichen Beschreibung zu finden. Eine Beschreibung aller Details der Implementierung ist in der Regel zu detailliert, da die primäre Zielgruppe einer Abschlussarbeit sich nicht im Detail in den geschriebenen Quelltext einarbeiten will. Die Beschreibung sollte aber durchaus alle wesentlichen Konzepte der Implementierung enthalten. Gerade bei einer Abschlussarbeit am Institut für Softwaretechnik und Programmiersprachen lohnt es sich, auf die eingesetzten Techniken und Programmiersprachen einzugehen. Ich würde in einer solchen Beschreibung auch einige unterstützende Diagramme erwarten.

7. 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.

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

[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

[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)

[Run]