UNIVERSITÄT ZUM BEISPIEL
INSTITUT FÜR BEISPIELE

Monitoring of the AUTOSAR Timing Extensions with TeSSLa

Überwachung der AUTOSAR Timing Extensions mittels TeSSLa

Bachelor Thesis

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

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

(Hendrik Streichhahn)
Lübeck, den 1.1. 1970

Abstract Satisfying given timing requirements is essential for the correct behavior of embedded real-time systems. In the automotive domain, the AUTOSAR timing extensions are a widely used and accepted standard for specifying timing requirements. Previous work, such as the TIMMO-2-USE project, has focused on formalizing the AUTOSAR timing model and timing extensions in a mathematically rigorous way, in order to make them amenable for offline system analysis tools such as automated model-checking and verification.
Because of computational problems, model-checking and offline verification is limited to relatively small-scale systems. Furthermore, not all types of specification violations can be detected at system development time, and sporadic, rare events typically require a capability for long-term observations. Run-time verification is a more lightweight method that lies at the boundary between formal verification and testing. Run-time verification checks properties, expressed in temporal logic, on-the-fly during the operation of the system using finite-state monitors generated from logical specifications. In this thesis, an analysis of the 18 TADL2 timing constraints defined in the TIMMO-2-USE project is made to examine, whether they can be expressed as finite-state monitors, thus making them monitorable by run-time verification. Further, a monitor for each of the TADL2 timing constraint is implemented in the temporal stream-based specification language TeSSLa.

Kurzfassung Die Einhaltung von Zeitschranken ist essentiell wichtig für das korrekte Verhalten von eingebetteten Echtzeitsystemen. In der Automobilindustrie werden in breiter Masse die AUTOSAR Timing Extensions (etwa *AUTOSAR Zeiterweiterungen*) verwendet, mit denen das das Zeitverhalten von Hard- und Softwarekomponenten beschrieben werden kann. Andere Arbeiten, etwa das TIMMO-2-USE Projekt, haben daran gearbeitet, die AUTOSAR Timing Extensions zu formalisieren und somit einen Grundbaustein dafür zu legen, diese Definitionen vom Zeitverhalten automatisiert zu kontrollieren, etwa durch Model Checking. Ein Problem von Model Checking und ähnlichen Ansätzen ist, dass diese aufgrund der extrem großen Laufzeit auf kleinere Systeme beschränkt sind. Runtime Verification ist eine leichtgewichtigere Methode der Analyse von Systemkomponenten, die einen Mittelweg zwischen formaler Analyse und Testen geht, wobei formal definierte Eigenschaften des Systems während der Laufzeit geprüft werden.

Im Rahmen dieser Arbeit werden die 18 TADL2 Timing Constraints, welche im Rahmen des TIMMO-2-USE Projekt erarbeitet wurden, dahingehend überprüft, ob sie in mittels Runtime Verification auf unendlichen Strömen überwacht werden können. Darauf aufbauend wird für jeden dieser Constraints ein Monitor in der Sprache TeSSLa, welche für die Überwachung von Zeiteigenschaften auf Strömen entwickelt wurde, implementiert.

Contents

1 Introduction

Timing behavior is one of the most important properties of computer systems. Especially in safety-critical applications, a wrong timed action or 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 [AUT]. With AUTOSAR, development processes and components are standardized, which increases productivity, interoperability and exchangeability of these components.
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 described as part of this thesis. This is problematic for monitoring, because the implementation of a monitor must not be based on unambiguous definitions. To solve this problem, the timing constraints defined in the **T**iming **A**ugmented **D**escription **L**anguage Version **2** (*TADL2*) are used as basis for the monitoring tool. *TADL2* was created as part of the TIMMO project, which had similar goals to AUTOSAR, but the definitions are written in a more formal way. The AUTOSAR Timing Extensions are comparable and partly compatible to the TADL2 timing constraints. Most of the constraints defined in the AUTOSAR standard can be described as equivalent combination of TADL2 timing constraints and vice versa.
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 of 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 *simple monitorability* is introduced, which ensures that a property on infinite streams can always be monitored with finite time and memory resources. Then, each of the TADL2 timing constraint is checked, if it *simple monitorable* or not. After

that, the TeSSLa implementations of these constraints are 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 to increase the interoperability, exchangebility and re-usability of parts and therefore simplifying development and production.
The AUTOSAR Timing Extension are describing timing constraints for actions and reactions of components. The constraints are defined via *events*, which consists of a time value and, if needed, a data value of an arbitrary type. To describe the logical relationship between groups of events, *event chains* are defined, which consists of *stimulus* and *response* events, in which 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 events with the same time distance and offers the possibility to set an allowed deviation from this pattern. Additionally 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 non overlapping event clusters with a minimal and maximal number of events. Optionally periodic repetitions of these clusters can also be described.

5 The **ArbitraryEventTriggering** defines the distance between subsequent events 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 kind, 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** defines the order, in which a list of executables must start and finish.
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 constraints are not formally defined and have room left for different interpretations. 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. These following attributes define, how the events may occur:

- ***maxNumberOfOccurrences*** (positive integer)
  Maximal number of events per burst
- ***minNumberOfOccurrences*** (positive integer)
  Minimal number of events per burst (optional)
- ***minimumInterArrivalTime*** (time value)
  Minimal distance between subsequent events

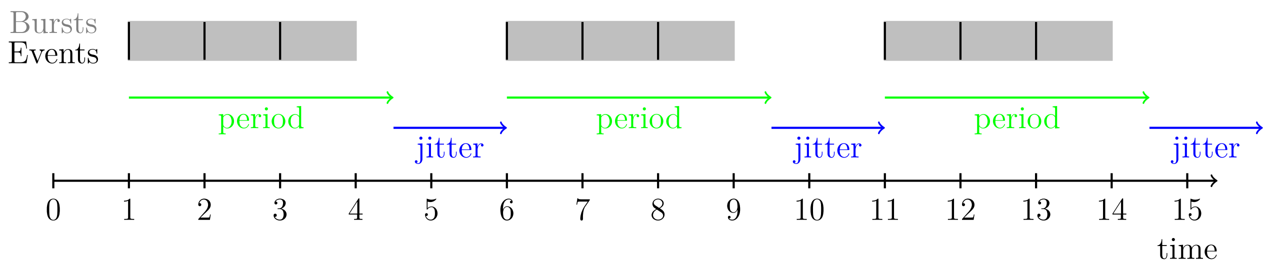

Figure 2.1: BurstPatternEventTriggering *patternPeriod* and *patternJitter* **accumulating**

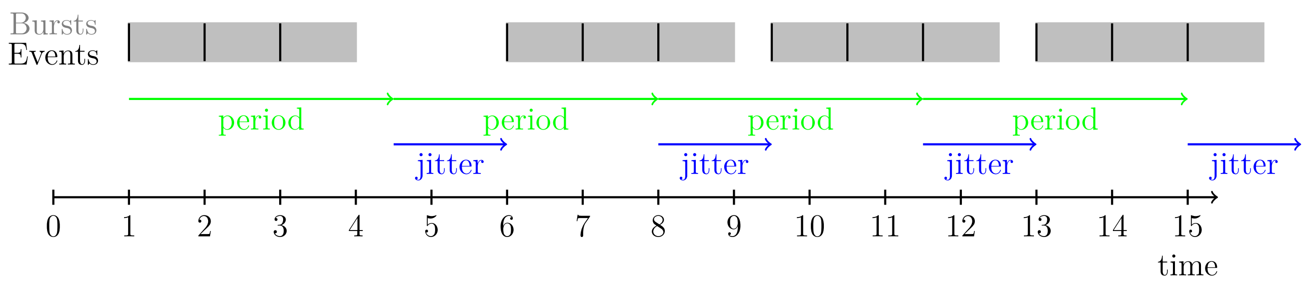

Figure 2.2: BurstPatternEventTriggering *patternPeriod* and *patternJitter* **non-accumulating**

- ***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 way, as seen in 2.1, or in a non-accumulating way, as seen in 2.2. 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.

With the definition of $patternPeriod$ ("time distance between the beginnings of subsequent repetitions of the given burst pattern"[AUT18]) you would think, that the accumulating variant is meant. Against that, the period attribute in *PeriodicEvent-Triggering*-Constraint is defined as "distance between subsequent occurrences of the event"[AUT18] 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 a monitor may need time and memory resources, which continuously grow with every input event. This makes online monitoring unsuitable in nearly all scenarios (more on monitorability in 3). As example, we will use the *BurstPatternEventTriggering* again. This time we use the attributes

- $maxNumberOfOccurrences = INT\_MAX$ or any significant large number
- $minNumberOfOccurrences = 1$
- $minimumInterArrivalTime = 0$
- $patternLength = 3$
- $patternPeriod$ 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 clusters with ongoing time is visualized. The gray bars show the range, in which 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. Because the minimal distance between subsequent events is not specified, an arbitrary large number of events can be placed in any interval with the length *patternLenth*.
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, because unlimited memory and time is needed to check the constraint in following events. Therefore, online monitoring this constraint is unsuitable in most cases.

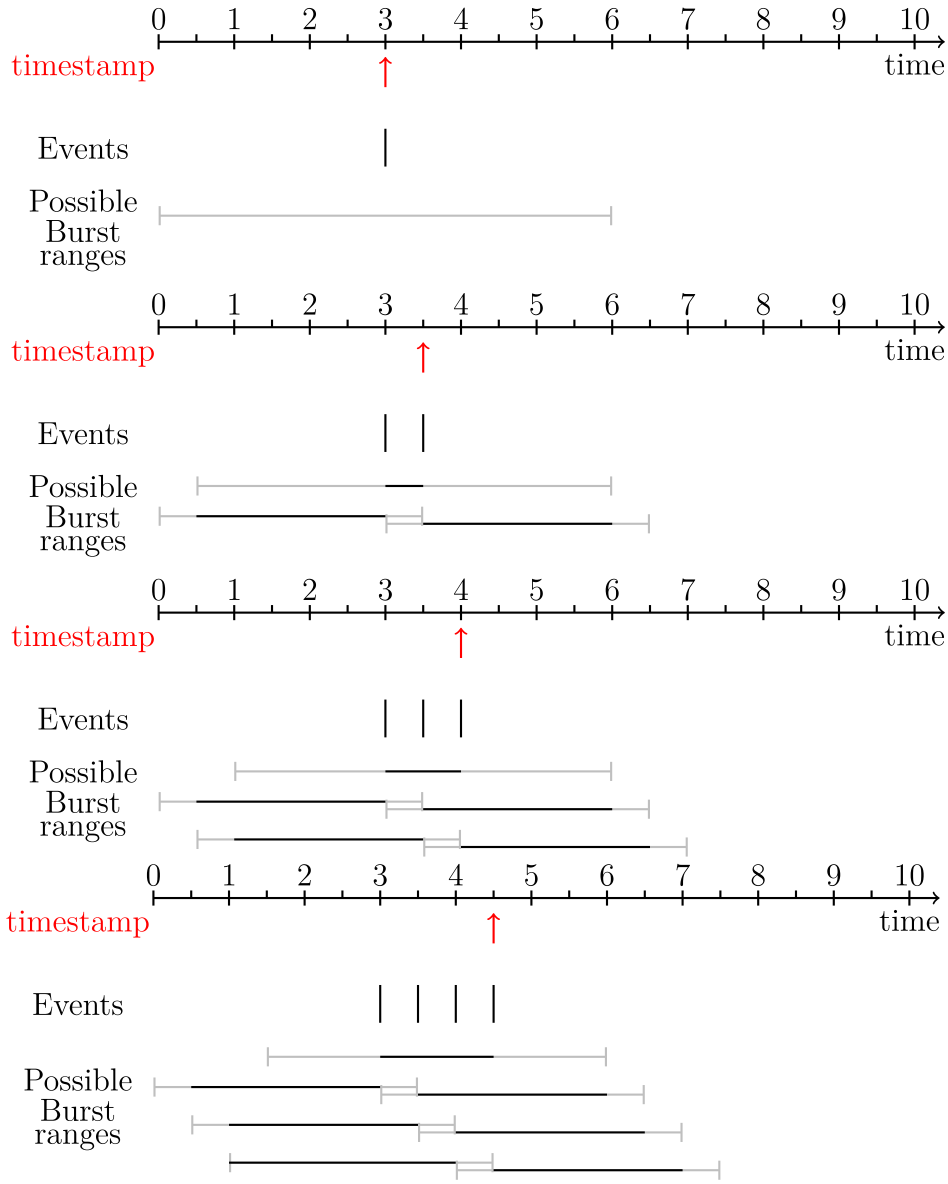

Figure 2.3: BurstPatternEventTriggering Possible bursts, ↑ shows the current time

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 the AUTOSAR Timing Extensions, 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 the timing constraint logic TiCL [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 associated events. This attribute is defined as abstract and possibly infinite datatype. The only restriction is, that an equality test on these color values must be defined. 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 counting 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 and 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 paper 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 every event only consists of a real number as timestamp, without the possibility of adding a data value. There are 7 syntactic categories in TiCL

$$\begin{aligned} \mathbb{R}&(\text{arithmetic constants}) \\ Avar&(\text{arithmetic variables}) \\ AExp&(\text{arithmetic expressions}) \end{aligned}$$

$$\begin{aligned} Svar&(\text{set variables}) \\ SExp&(\text{set expressions}) \end{aligned}$$

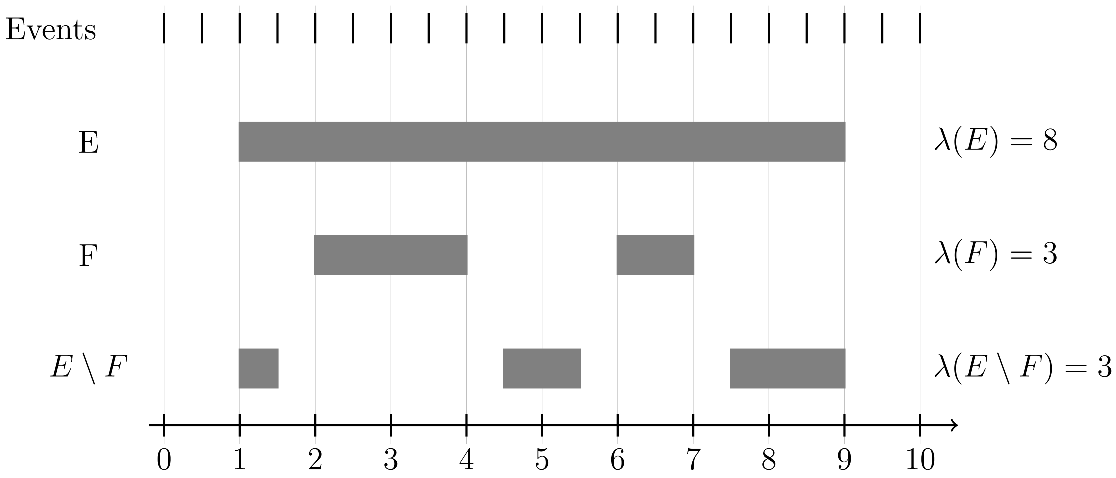

Figure 2.4: Graphical example of $\lambda(E), \lambda(F)$ and $\lambda(E \setminus F)$

$TVar$(time variables)

$CExp$(constraint expressions)

Arithmetic expressions can be defined as arithmetic constants, as arithmetic variables, as application of $+, -, *, /$ on arithmetic expressions, as 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). Consequently 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 ($\wedge$) on constraint expressions, the negation of constraint expressions and the $\forall$-Quantifier on arithmetic, set and time variables over a 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.

Let $x, y \in Tvar$ and $E, F \in SExp$.

The interval constructor $[x <]([x \leq])$ is defined as $\{y : x < y\}(\{y : x \leq y\})$, therefore the interval contains all points in time laying behind of x (including 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 points in 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. Please note the use of $\forall$ instead $\exists$ in the definition.

$[\leq E]$ ($[< E]$) 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) x 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.

$E(i)$ is i^{th} timestamp in E, starting by zero.

$E \leq F$ describes, that E is a sub sequence of F, which means that between the earliest and latest element in E all elements of F are in 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 a data value. The range of the data values are arbitrary, infinite data types are possible as well as empty data types, when only the point in time is relevant for the constraint. 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 the datatype of the color must be defined. It is used to check, which events of an *EventChain* are directly related.

DelayConstraint

The *DelayConstraint* has 4 attributes

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

and is defined as

$$DelayConstraint(source, target, lower, upper)$$
$$\Leftrightarrow \forall x \in source : \exists y \in target : lower \leq y - x \leq upper.$$

For all events x in *source*, there must be an event y in *target*, so that the time distance between x and y is between *lower* and *upper*. 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.

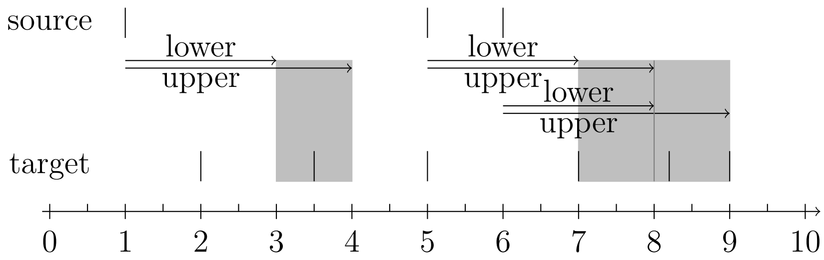

Figure 2.5: Example DelayConstraint - *lower* = 2, *upper* = 3

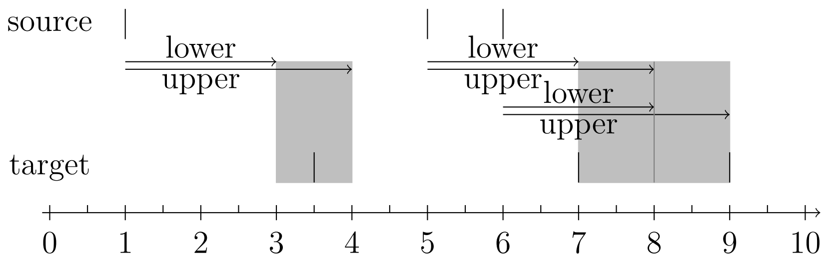

Figure 2.6: Example StrongDelayConstraint - *lower* = 2, *upper* = 3

StrongDelayConstraint

The *StrongDelayConstraint* has 4 attributes

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

and is defined as

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

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.

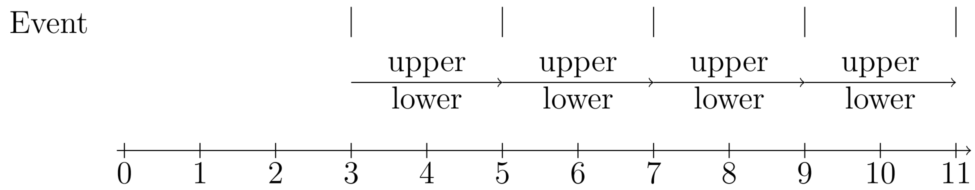

Figure 2.7: Example RepeatConstraint - $lower = 2$, $upper = 2$, $span = 1$

RepeatConstraint

The *RepeatConstraint* also has 4 attributes

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

and is defined as

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

As reminder, the $\leq$-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 $[A]$ is the time interval between the oldest and newest event in A.

The definition of the *RepeatConstraint* specifies that the length of each time interval containing $span + 1$ subsequent 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 then 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 to *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 distances between all subsequent events with an even index are considered, as well as the distances between subsequent events with an uneven index.

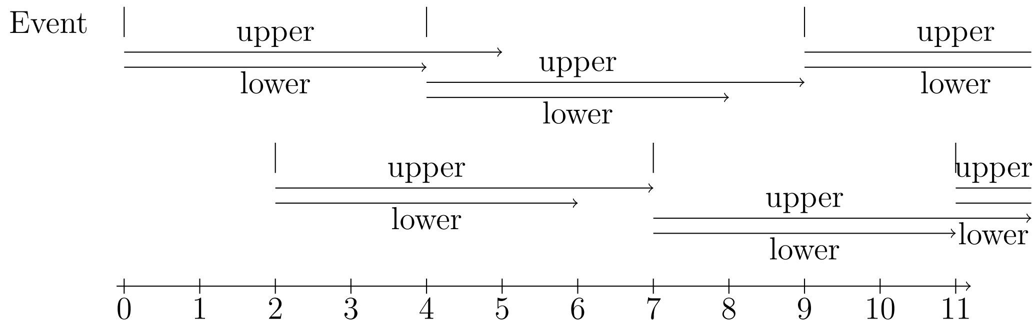

Figure 2.8: Example RepeatConstraint - *lower* = 4, *upper* = 5, *span* = 2

RepetitionConstraint

The *RepetitionConstraint* has 5 attributes

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

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

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

where X is a set of arbitrary time stamps, that follow the structure of the *Repeat-Constraint*(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 exactly 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	$\mathbb{T}$

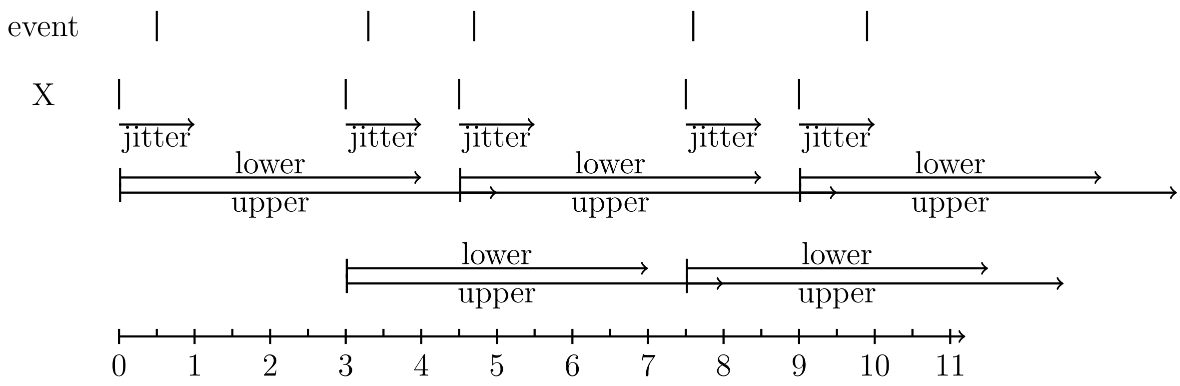

Figure 2.9: Example RepetitionConstraint - *lower* = 4, *upper* = 5, *span* = 2, *jitter* = 1

and is defined via the *DelayConstraint* as

$$
\begin{aligned}
&SynchronizationConstraint(event_1, ..., event_n, tolerance)\\
&\Leftrightarrow \exists X : \forall i : DelayConstraint(X, event_i, 0, tolerance) \land\\
&\qquad DelayConstraint(event_i, X, -tolerance, 0)
\end{aligned}
$$

X is a set of timestamps and 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 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 be 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, $\lvert event \rvert \geq 2$
tolerance	$\mathbb{T}$

and is defined as

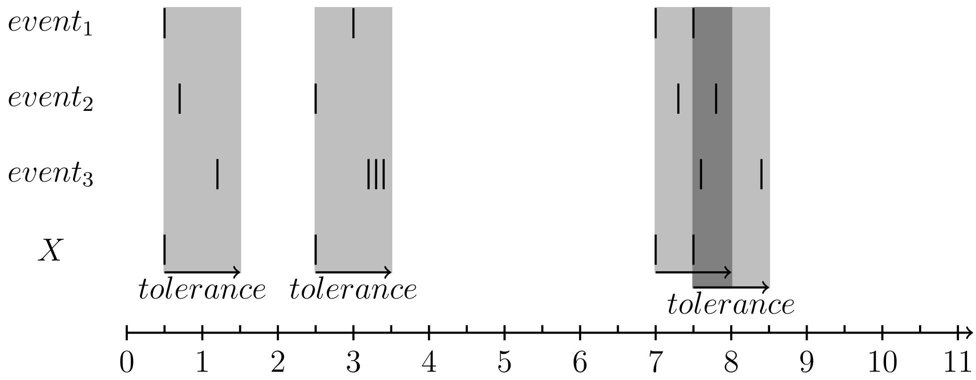

Figure 2.10: Example SynchronizationConstraint - $tolerance = 1$

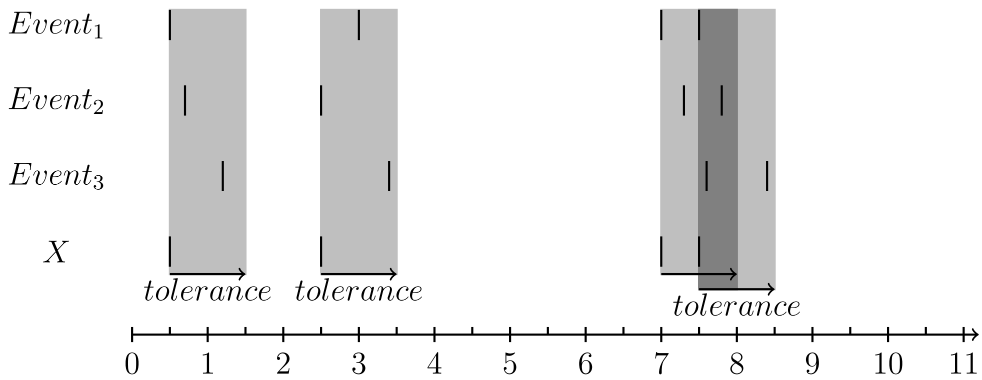

Figure 2.11: Example StrongSynchronizationConstraint - $tolerance = 1$

$StrongSynchronizationConstraint(event_1, \ldots, event_n, tolerance)$
$\Leftrightarrow \exists X : \forall i : StrongDelayConstraint(X, event_i, 0, tolerance)$

This constraint 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 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.

ExecutionTimeConstraint

The *ExecutionTimeConstraints* takes 6 attributes

start set of events

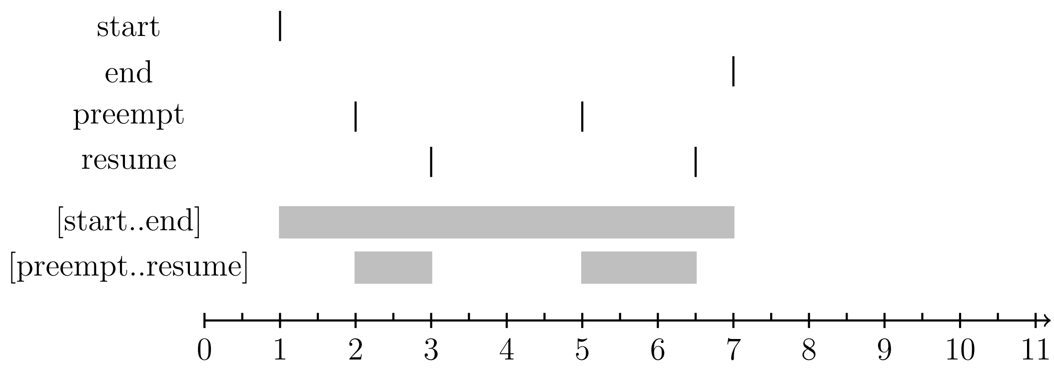

Figure 2.12: Example ExecutionTimeConstraint

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

and is defined as

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

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. $[preempt..resume]$ defines the intervals between each element of preempt until the next timestamp of resume and is removed from the considered interval length.
The Idea behind this constraint is to define 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.

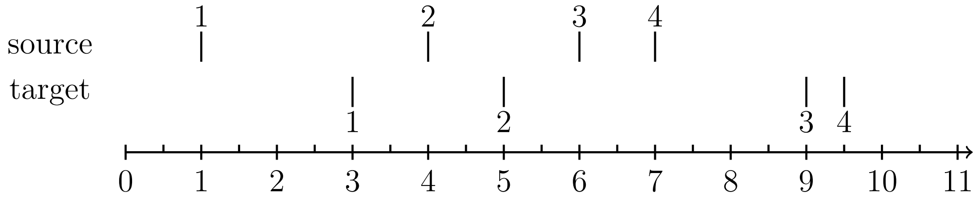

Figure 2.13: Example OrderConstraint

OrderConstraint

The *OrderConstraint* takes two attributes

source	set of events
target	set of events

and is defined as

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

This constraints ensures the order of events, so that the i-th event of *target* occurs 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

$$\begin{aligned}&SporadicConstraint(event, lower, upper, jitter, minimum)\\ \Leftrightarrow\ &RepetitionConstraint(event, lower, upper, 1, jitter)\\ &\wedge RepeatConstraint(event, minimum, \infty, 1)\end{aligned}$$

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 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*.

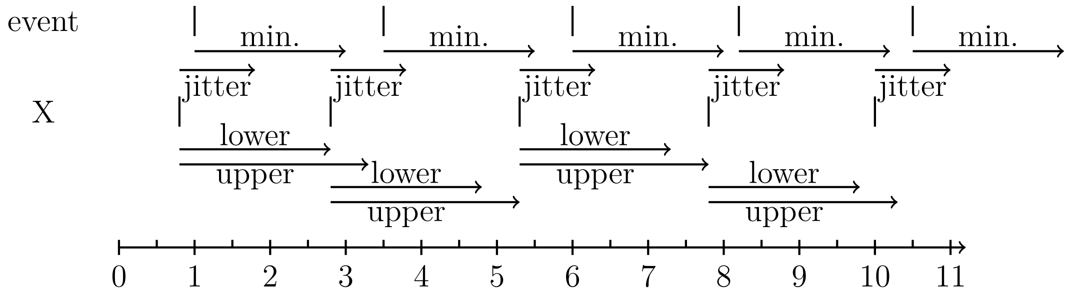

Figure 2.14: Example SporadicConstraint - $lower = 2$, $upper = 2.5$, $jitter = 1$, $minimum = 2$

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*

$$PeriodicConstraint(event, period, jitter, minimum)$$
$$\Leftrightarrow SporadicConstraint(event, period, period, jitter, minimum)$$

The variable timestamps in the set X are 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 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	$\mathbb{T}$

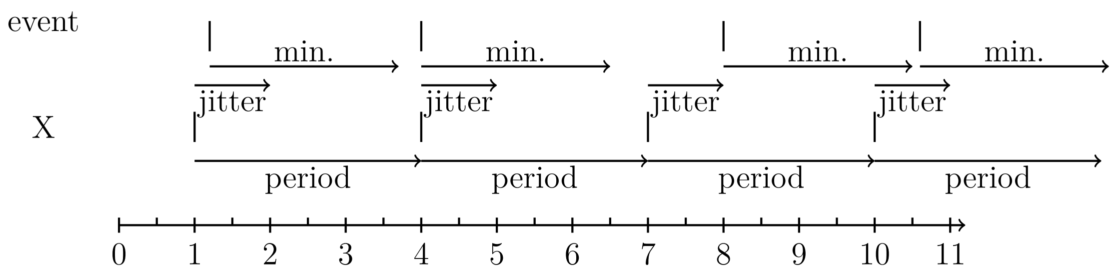

Figure 2.15: Example PeriodicConstraint - *period* = 3, *jitter* = 1, *minimum* = 2.5

offset	set of $\mathbb{T}$
jitter	$\mathbb{T}$
minimum	$\mathbb{T}$

and is defined as

$$
\begin{aligned}
&PatternConstraint(event, period, \mathit{offset}_1, ..., \mathit{offset}_n, jitter, minimum)\\
&\Leftrightarrow \exists X : PeriodicConstraint(X, period, 0, 0)\\
&\quad \wedge \forall i : DelayContraint(X, event, \mathit{offset}_i, \mathit{offset}_i + jitter)\\
&\quad \wedge RepeatConstraint(event, minimum, \infty, 1)
\end{aligned}
$$

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 $|\mathit{offset}_i|$ 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 events in *event*. Therefore, the *PatternConstraint* only ensures the minimal distance 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

$$
\begin{aligned}
&PatternConstraint(event, period, \mathit{offset}_1, ..., \mathit{offset}_n, jitter, minimum)\\
&\Leftrightarrow \exists X : PeriodicConstraint(X, period, 0, 0)
\end{aligned}
$$

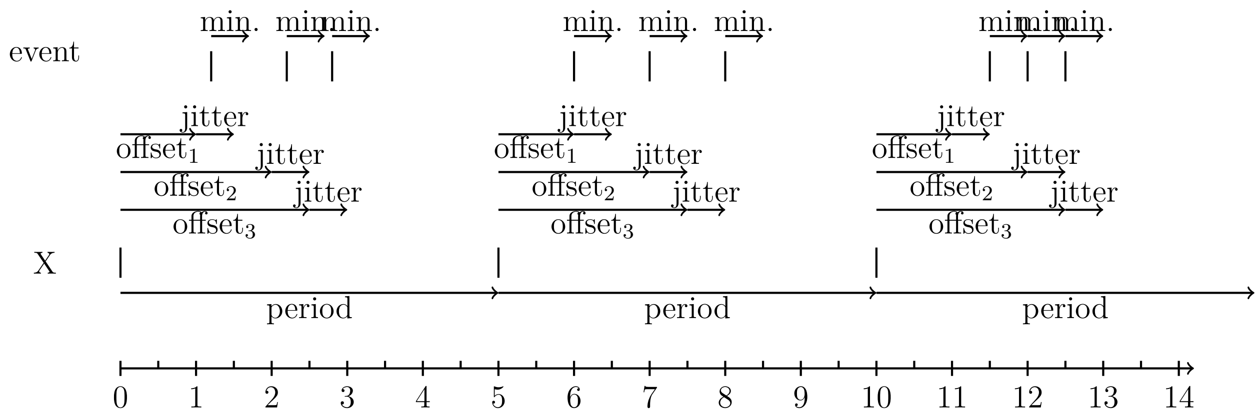

Figure 2.16: Example PatternConstraint - $period = 5$, $offset = \{1, 2, 2.5\}$, $jitter = 0.5$, $minimum = 0.5$

$$\land \forall i : \boldsymbol{Strong}DelayContraint(X, event, offset_i, offset_i + jitter)$$
$$\land RepeatConstraint(event, minimum, \infty, 1)$$

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[BFL+12]. 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 the 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*.

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

	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

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

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, defines 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, 2 and more 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*
minimum	$\mathbb{T}$

and is defined as

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

The idea of this constraint is to describe the maximum number of events that may

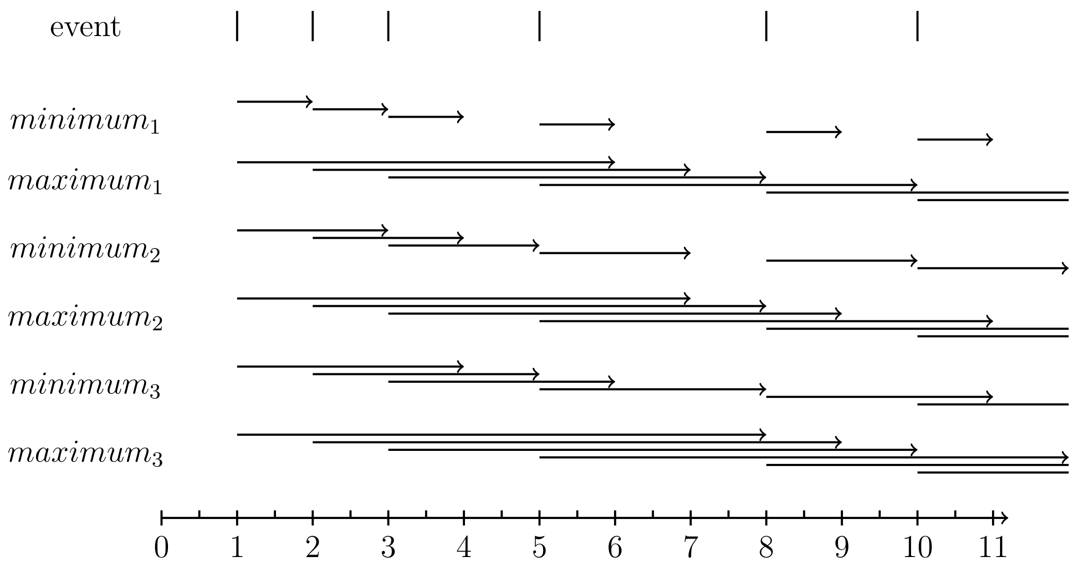

Figure 2.17: Example ArbitraryConstraint - *minimum* $= \{1, 2, 3\}$ and *minimum* $= \{4, 5, 6\}$

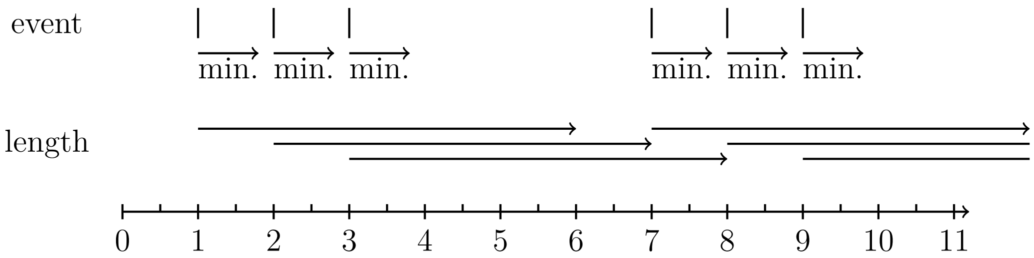

Figure 2.18: Example BurstConstraint - *length* $= 5$, *maxOccurences* $= 3$ *minimum* $= 0.8$

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, an application of 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

$$\begin{array}{rl} scope & EventChain \\ minimum & \mathbb{T} \\ maximum & \mathbb{T} \end{array}$$

and is defined as

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

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 $scope.stimulus$.
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

$$\begin{array}{rl} scope & EventChain \\ minimum & \mathbb{T} \\ maximum & \mathbb{T} \end{array}$$

and is defined as

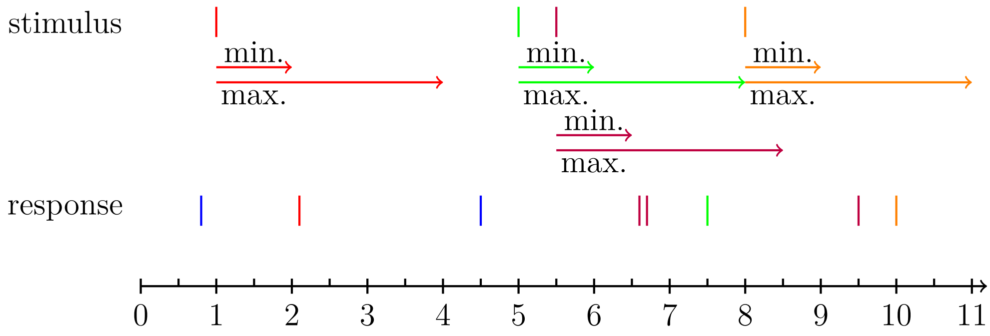

Figure 2.19: Example ReactionConstraint - $minimum = 1$, $maximum = 3$

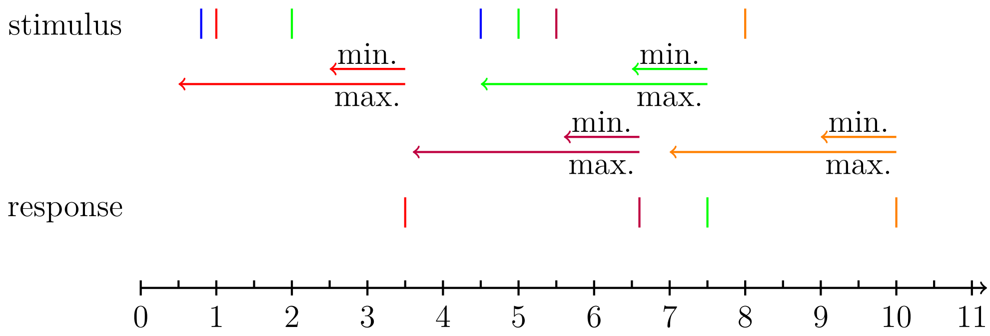

Figure 2.20: Example AgeConstraint - $minimum = 1$, $maximum = 3$

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

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, which is implied by the correctness of the event chain.
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*.

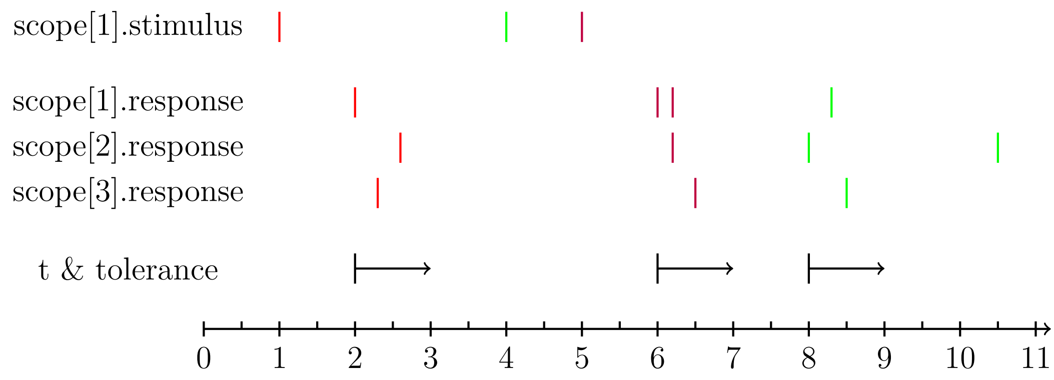

Figure 2.21: Example OutputSynchronizationConstraint - $tolerance = 1$

OutputSynchronizationConstraint

The *OutputSynchronizationConstraint* takes 2 attributes

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

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

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

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 $tolerance = 1$,
$scope[1].stimulus = scope[2].stimulus = scope[3].stimulus = \{(1, red), (4, green), (5, purple)\}$,
$scope[1].response = \{(2, red), (6, purple), (6.2, purple), (8.2, green)\}$,
$scope[2].response = \{(2.6, red), (6.2, purple), (8, green), (10.5, green)\}$,
$scope[3].response = \{(2.3, red), (6.5, purple), (8.5, green)\}$.

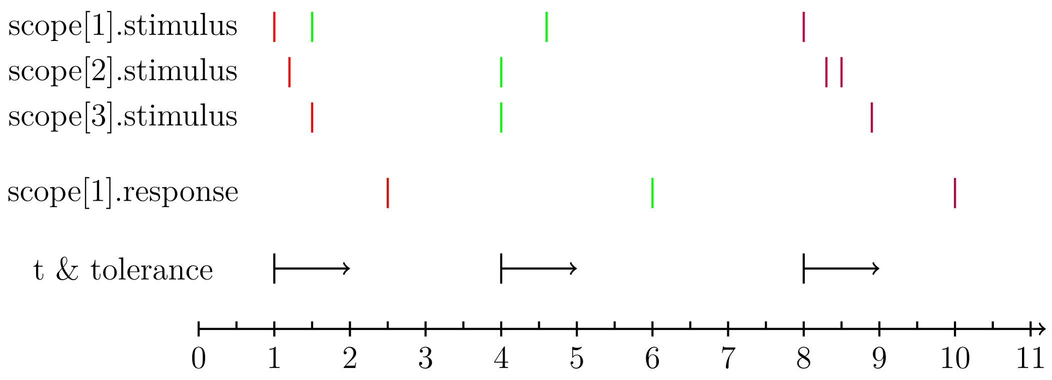

Figure 2.22: Example InputSynchronizationConstraint - $tolerance = 1$

InputSynchronizationConstraint

The *InputSynchronizationConstraint* takes 2 attributes

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

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

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

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 Extensions* 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. Some of the AUTOSAR Timing Extensions are defined on *Executable Entities*, describe things, that can be executed, for example a function. In the analysis of their timing, only striking points in times of these entities are relevant, for example the start or end points, therefore *Executable Entities* can be transformed into events if needed.

It should be noted, that the set of TADL2 timing constraints are not equal to the AUTOSAR Timing Extension and that there are constraints, 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:
$lower \widehat{=} period$

$upper \widehat{=} maximumInterArrivalTime$
$jitter \widehat{=} jitter$
$minimum \widehat{=} minimumInterArrivalTime$

ConcretePatternEventTriggering

The idea behind the *ConcretePatternEventTriggering* from $AUTOSAR$ is the same as behind *TADL2s PatternConstraint*, but some details 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 from the periodic repetitions, but 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

$$\begin{aligned} & 0 \leq max(\mathit{offset}) \leq patternLength \\ \wedge \quad & patternLength + patternJitter < patternPeriod \end{aligned}$$

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 *BurstPattern-EventTriggering* those can start in any point of time.

ArbitraryEventTriggering

AUTOSARs *ArbitraryEventTriggering* is similar to the *ArbitraryConstraint* as defined in TADL2, but the *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 defined in the $TADL2$ constraint, if this attribute is not required for the specification, the *LatencyTimingConstraint* can be expressed with the *AgeConstraint* defined in TADL2, if the $latencyConstraintType$ is *age*. If the $latencyConstraintType$ is *reaction*, it can be expressed by 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 TADL2s *SynchronizationConstraint*, *StrongSynchronizationConstraint*, *OutputSynchronizationConstraint*, *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).

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 the 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

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	Strong-SynchronizationConstraint (scopeEvent, tolerance)
multiple Occurrences	scope	response Synchronization	tolerance	Output-SynchronizationConstraint (scope, tolerance) ∧ SynchronizationConstraint (scope.response, tolerance)
single Occurrences	scope	response Synchronization	tolerance	Output-SynchronizationConstraint (scope, tolerance) ∧ Strong-SynchronizationConstraint (scope.response, tolerance)
multiple Occurrences	scope	stimulus Synchronization	tolerance	Input-SynchronizationConstraint (scope, tolerance) ∧ SynchronizationConstraint (scope.stimulus, tolerance)
single Occurrences	scope	stimulus Synchronization	tolerance	Input-SynchronizationConstraint (scope, tolerance) ∧ SynchronizationConstraint (scope.stimulus, tolerance)

Table 2.2: SynchronizationTimingConstraint ⇔ TADL2 Constraints

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 (which can be set in *AUTOSARs ExecutionTimeConstraint*), 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 at least as many events as occur in the stream in a specific amount of time, not every property can be monitored in an online monitoring setting. In this chapter, the term of *Simple Monitorability* will be introduced, which ensures that monitoring a property on infinite streams is possible with finite memory resources and finite run time 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 a sequence of 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 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 a 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 a possibly infinite sequence $s = a_0a_1... \in \mathcal{S}_D = (\mathbb{T}\cdot\mathbb{D})^\omega \cup (\mathbb{T}\cdot\mathbb{D})^+ \cup (\mathbb{T}\cdot\mathbb{D})^* \cdot (\mathbb{T}_\infty \cup \mathbb{T}\cdot\{\bot\})$ 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 in timestamps, in which 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. If the *delay*-operator is not used in a specification, the output streams only contains events in timestamps, which also had events in the input streams. These specifications are called *timestamp conservative*.
In a memory perspective, streams may be understood as *piece wise constant signals*. Only the timestamp and the data value of the youngest event of one stream 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

[DSS+05] introduces *LOLA*, a specification language for the observation of synchronous event streams, comparable to TeSSLa. The main difference between these languages is, that TeSSLa is designed to monitor input streams, which are not synchronized, which means their events may occur independently from each other. Because the events of the timing constraints defined in TADL2 and AUTOSAR are

also not synchronized, TeSSLa is more suitable for monitoring them.
[DSS+05] 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.

Semantics of LTL$_3$[ABLS05] RV-LTL[BLS07]

In Runtime Verification, the output of a monitor is based on a finite prefix on a possibly infinite trace of system states. On many of these prefixes, it can't be finally decided, if this run the system fulfills a given property or not. Because of this, a binary boolean output of a monitor may be misleading, because it cannot express, if the output is final or may change due to upcoming system states.

Definition 1 (LTL$_3$[ABLS05])**.** *[ABLS05] introduces a three valued semantics for Runtime Verification and LTL (Lineal Temporal Logic). This semantic, LTL$_3$ called, is defined on a finite trace $u = a_0 ... a_{n-1} \in \Sigma^*$ of length n. A LTL$_3$ formula ϕ on trace u at position $i < n$ is defined as:*

$$[u, i \models \varphi]_3 = \begin{cases} \top & \textit{if } \forall \sigma \in \Sigma^\omega : u\sigma, i \vDash \varphi \\ \bot & \textit{if } \forall \sigma \in \Sigma^\omega : u\sigma, i \nvDash \varphi \\ ? & \textit{otherwise} \end{cases}$$

If the prefix σ fulfills the formula ϕ, $[u, i \models \phi]$ is $\top$. If it doesn't fulfill the formula, $[u, i \models \phi]$ is $\bot$. In any other case, $[u, i \models \phi]$ is defined as ?.

[BLS07] introduces a four valued semantics called RV-LTL. It is defined as extension of LTL$_3$, where the ?-value is further splitted.

Definition 2 (RV-LTL)**.** *Again, let $u = a_0 ... a_{n-1} \in \Sigma^*$ be a finite trace of length n, $i < n$, and φ be a RV-LTL formula. The truth value of this formula is defined as:*

$$[u, i \models \varphi]_{RV} = \begin{cases} \top & \textit{if } [u, i \vDash \varphi']_3 = \top \\ \bot & \textit{if } [u, i \vDash \varphi']_3 = \bot \\ \top^p & \textit{if } [u, i \vDash \varphi']_3 =? \wedge [u, i \vDash \varphi]_F = \top \\ \bot^p & \textit{if } [u, i \vDash \varphi']_3 =? \wedge [u, i \vDash \varphi]_F = \bot \end{cases}$$

where ϕ' is a modified form of ϕ, in which the weak next-state operator is replaced by the strong next-state operator. $[u, i \vDash \varphi]_F$, as defined in [LPZ85], is the binary (two valued boolean) truth value of the formula φ over u at position i.

$\top$ and $\bot$ are final truth values, which will not change due to upcoming symbols of u. $\top^p$ and $\bot^p$ may change based on upcoming symbols of u, but on the prefix u of the length i, φ is (not) fulfilled.

3.1.1 Transducer Models

In section 3.2, some transducer models are used, which will be introduced next.

Definition 3 (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*

The run of a DFST for an input word $w = w_0w_1w_2... \in \Sigma^\infty$ *is a sequence* $s_0 \xrightarrow{w_0/o_0} s_1 \xrightarrow{w_1/o_1} s_2...$*, where* $s_0 = q_0$*,* $\delta(s_i, w_i) = (s_{i+1}, o_i), i \geq 0$ *and the output word* $o = o_0o_1o_2... \in \Gamma^\infty$*.*

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 states of a DFST are not accepting. The transducer *transduces* an input word, not accepting or rejecting it.

Timed Deterministic Finite State Transducer(TDFST) are an extension of DFST. The extension from DFST to TDFST is done analogous to the extension of automata to timed automata in [AD94].

Definition 4 (Timed Deterministic Finite State Transducer)**.** *Timed Deterministic Finite State Transducers(TDFST) are a 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, where for all $(q_a, \sigma_a, \vartheta_a, q'_a, R_a, \gamma_a), (q_b, \sigma_b, \vartheta_b, q'_b, R_b, \gamma_b) \in \delta$ the conjunction $\vartheta_a \wedge \vartheta_b$ is unsatisfiable.*

Let $v_i : C \to R$ be functions that map each clock to its current value.

The run of a TDFST for an input word $w = (w_0, t_0)(w_1, t_1)(w_2, t_2)..., w_i \in \Sigma^\infty$ is a sequence $s_0, v_0 \xrightarrow[o_0]{(w_0,t_0),\vartheta_0,r_0} s_1, v_1 \xrightarrow[o_1]{(w_1,t_1),\vartheta_1,r_1} s_2, ...$ with output $o = o_0 o_1 o_2 ... \in \Gamma^\infty$, if, and only if,

- $s_0 = q_0$
- $\forall c \in C : v_0(c) = 0$
- $\forall i \geq 0 :$
  - $\delta(s_i, w_i, \vartheta_i) = (s_{i+1}, r_i, o_i)$
  - $\forall c \in r_i : v_{i+1} = v_i[c \leftarrow t_i]$
  - $t_i, v_i \models \vartheta_i$

In addition to DFSTs, 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 *Simple Monitorability* is introduced. It represents a stricter alternative to *Efficiently Monitorable Specifications* mentioned above, by also restricting the allowed run time per timestamp with events. *Simple Monitorability* ensures, that the worst case memory consumption and the worst case run time per input event of a monitor is bounded independently of the input streams.

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 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:

1. The first used timestamp has the value $t_0 = 0$

2. All used timestamps must be smaller than t_{max}.
   t_{max} must be big enough, so it is not reached in practical use [1].
3. The distance between two subsequent time values is predetermined, but arbitrary small.
4. The number of possible timestamps is significantly larger than the number of events.

Because of the 2., 3. and 4., a limitation of the number of events in a specific time interval through these restrictions is invalid.

3.2.1 Simple Monitorability

The concept behind the definition of *Simple Monitorability* is that a monitor for event streams is defined by three parts, a 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.

Definition 5 (Simple Monitorability)**.** *A property is called Simple Monitorable, if a monitor, which outputs $\top$, $\top^p$, $\bot^p$ and $\bot$ as defined for RV-LTL, can be constructed in the following way:*

Input Streams *Let $S_1, S_2, ..., S_n$ be the input streams with*
$\forall i : S_i = (\mathbb{T} \cdot \mathbb{D}_i)^\omega \cup (\mathbb{T} \cdot \mathbb{D}_i)^+ \cup (\mathbb{T} \cdot \mathbb{D}_i)^* \cdot (\mathbb{T}_\infty \cup \mathbb{T} \cdot \{\bot\})$

State *Let S_{state} with $S_{state} = (\mathbb{T} \cdot \mathbb{D}_{state})^+ \cup (\mathbb{T} \cdot \mathbb{D}_{state})^*$ be the state stream of the monitor. The cardinality of $\mathbb{D}_{state}$ is finite and the worst case memory requirement is bounded independently of the input streams.*
Further let $f : S_1 \times S_2 \times ... \times S_n \times S_{state} \rightarrow S_{state}$ be a state transition function, which defines the state stream in an incremental fashion:
$\forall t \in \mathbb{T} \exists i \in \{1, 2, ..., n\} : S_i(t) \in \mathbb{D}_i :$
$S_{state}(t) = f(S_1(t), S_2(t), ..., S_n(t), last(S_{state}, merge(S_1, S_2, ..., S_n))(t))$
The worst case run time of f is bounded independently of the input streams.

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

Output Stream *Let* $S_{output} = (\mathbb{T} \cdot \{\top, \top^p, \bot^p, \bot\})^+ \cup (\mathbb{T} \cdot \{\top, \top^p, \bot^p, \bot\})^*$
be the output stream of the monitor, which is defined via a function
$g : S_{state} \times S_{output} \rightarrow S_{output}$
which defines the output stream in an incremental fashion:
$\forall t \in \mathbb{T} \exists S_{state}(t) \in \mathbb{D}_i :$
$S_{output}(t) = g(S_{state}(t), last(S_{output}, S_{state})(t))$
The worst case run time of g *is bounded independently of the input streams.*

In every timestamp with input events, the state transition function f is applied to the current youngest input events and the previous event of the state stream S_{state}. The output of f, combined of the timestamp of the latest input event, defines the new event in S_{state}. The output of function g is applied to the current state and the previous output and produces the new output. It should be noted that a monitor, which follows this scheme is *timestamp conservative*.
For any monitor, which is created in the way described above, a Deterministic Finite State Transducer (DFST) can be constructed, which is equivalent to the combination of a finite state and the state transition function. For that, let

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

For the output stream in combination with the output function, an equivalent DFST can be constructed too. For that, let

- $Q' = \{\top, \top^p, \bot^p, \bot\}$ be the states with initial state $q \in Q'$
- $\Sigma' = \mathbb{D}_{state} \times \mathbb{T}$ be the input alphabet
- $\Gamma' = \{\top, \top^p, \bot^p, \bot\}$ be the output alphabet and
- $\delta' : Q' \times \Sigma' \rightarrow Q' \times \Gamma'$ be the transition function.

It should be noted, that both transducers could be combined into one transducer without changing the expressiveness. This is not done to keep analogies to the following definition.

3.2.2 Simple Monitorability With Delay

Most of the TADL2 constraints can not be monitored correctly 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 *Simple Monitorability* is expanded by the ability of introducing exactly one new timestamps. The following definitions are visualized in the right half of figure 3.1.

Definition 6 (Simple Monitorability With Delay)**.** *A property is called Simple Monitorable With Delay, if a monitor, which outputs* $\top$*,* $\top^p$*,* $\bot^p$ *and* $\bot$ *as defined for RV-LTL, can be constructed in the following way:*

Input Streams *Let* $S_1, S_2, ..., S_n$ *be the input streams with*
$\forall i : S_i = (\mathbb{T} \cdot \mathbb{D}_i)^\omega \cup (\mathbb{T} \cdot \mathbb{D}_i)^+ \cup (\mathbb{T} \cdot \mathbb{D}_i)^* \cdot (\mathbb{T}_\infty \cup \mathbb{T} \cdot \{\bot\})$

State *Let* S_{state} *with* $S_{state} = (\mathbb{T} \cdot \mathbb{D}_{state})^+ \cup (\mathbb{T} \cdot \mathbb{D}_{state})^*$ *be the state stream of the monitor. The cardinality of* $\mathbb{D}_{state}$ *is finite and the worst case memory requirement of the state is bounded independently of the input streams.*
Further let $f : S_1 \times S_2 \times ... \times S_n \times S_{state} \to S_{state}$ *be a state transition function, which defines the state stream in an incremental fashion:* $\forall t \in \mathbb{T} \exists i \in \{1, 2, ..., n\} : S_i(t) \in \mathbb{D}_i :$
$S_{state}(t) = f(S_1(t), S_2(t), ..., S_n(t), last(S_{state}, merge(S_1, S_2, ..., S_n))(t))$
The worst case run time of f *is bounded independently of the input streams.*

State$_{timeout}$ *Let* $S_{state_{timeout}}$ *with* $S_{state_{timeout}} = (\mathbb{T} \cdot (\mathbb{D}_{state} \cup \{timeout\}))^+ \cup (\mathbb{T} \cdot (\mathbb{D}_{state} \cup \{timeout\}))^*$ *be the a second state stream, which is defined via a delay generator* $DelayGen : S_{state} \to S_{state_{timeout}}$*. DelayGen has two tasks. First, it copies each input event to the output. Second, a timer is started at every input timestamp. The duration of this timer is dependent of the input. If the next input comes before the timer runs out, the timer is resetted and started again. If the timer runs out once, the Delay Generator outputs the timeout signal, which is repeated at every following input and the timer is not started again. The worst case run time for the calculation of the required delay must be bounded independently of the input streams.*

Output Stream *Let* $S_{output} = (\mathbb{T} \cdot \{\top, \top^p, \bot^p, \bot\})^+ \cup (\mathbb{T} \cdot \{\top, \top^p, \bot^p, \bot\})^*$
be the output stream of the monitor, which is defined via a function
$g : S_{state} \times S_{output} \to S_{output}$
which defines the output stream in an incremental fashion:
$\forall t \in \mathbb{T} \exists S_{state}(t) \in \mathbb{D}_i :$

$S_{output}(t) = g(S_{state_{timeout}}(t), last(S_{output}, S_{state_{timeout}})(t))$
The worst case run time of g is bounded independently of the input streams.

A monitor, which is *Simple Monitorability With Delay*, is not *timestamp conservative* anymore, because one new timestamp can be created. Because of this characteristic, the monitor cannot be described via (Timed) Deterministic Finite State Transducers. To solve this problem, a modification to TDFST is done, which allows ε-transitions, which are guarded by a clock constraint, but do not consume an input symbol to perform a state transition.

Definition 7 (Delay Generator)**.** *Let* $tmr : \mathbb{D} \to \mathbb{T}$ *a function, which determines the required delay period.*
A Delay Generator is a 6-Tuple $(\Sigma, \Gamma, Q, q_0, C, \delta)$*, where*

- $Q = \{q_{start}, q_{timeout}\} \cup \{q_{wait,i} | \forall i \in \mathbb{D}_0\}$ *is a finite set of states with initial state* q_{start}
- $\Sigma = \mathbb{D}_{state}$ *is an input alphabet*
- $\Gamma = \mathbb{D}_{state} \cup \{timeout\}$ *is an output alphabet*
- $C = \{c\}$ *is a set of exactly one clock and*
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Theta(C) \to Q \times 2^C \times \Gamma$ *a state transition function.* δ *is defined as:*

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

The definition of the *Delay Generator* is visualized in figure 3.2.2. On the left side is the initial state q_{start}. The first input leads to a transition to the wait state of the corresponding input symbol. 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 $i \in \mathbb{D}_{state}$ is used as output and clock c is resetted. In the timeout state, each input symbol leads to a repetition of the *timeout* symbol.

A monitor, which monitors a property that is *Simple Monitorability With Delay*, is equivalent to a combination of two DFSTs and a *Delay Generator*. The first DFST depicts, like before, the combination of state and state transition function. For this transducer, let

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

The output of this DFST is the input of the *Delay Generator* introduced above. The output of the *Delay Generator* is the input of the second DFST, which represents the output stream in combination with the output function, which is defined in the following way:

- $Q' = \{\top, \top^p, \bot^p, \bot\}$ are the states with initial state $true_{until}$
- $\Sigma' = (D_{state} \cup \{timeout\}) \times \mathbb{T}$ is the input alphabet
- $\Gamma' = \{\top, \top^p, \bot^p, \bot\}$ is the output alphabet and
- $\delta' : Q' \times \Sigma' \rightarrow Q' \times \Gamma'$ is the transition function, where $delta(q, (timeout, t)) = false$ for each possible q and t.

The output transducer is nearly the same as before, the difference is, that it additionally takes the *timeout* symbol as input and in this case always returns $\bot$.

3.2.3 Not Simple Monitorable

Disproving one of the characteristics of *Simple Monitorability* does not necessarily mean, that a property of infinite streams can not be monitored with finite resources. For example, a property, where all parts of *Simple Monitorability* are fulfilled, except the worst case run time of the state transition function, which is dependent on the input streams, but the average run time of this function over every possible trace of this function is not[2]. In these cases, a monitor with finite resources can be constructed, which can observe arbitrary long input traces.
If you can prove, that a limitation of the memory consumption, which is required to monitor a property correctly, cannot be given independently of the input streams and their events, it can be safely said, that a property cannot be monitored correctly on arbitrary input traces with finite resources. This is, because the memory of a computation system is always finite. If the required storage space is dependent on

[2]In other words, the run time over the entire trace is linear dependent on the length of the trace

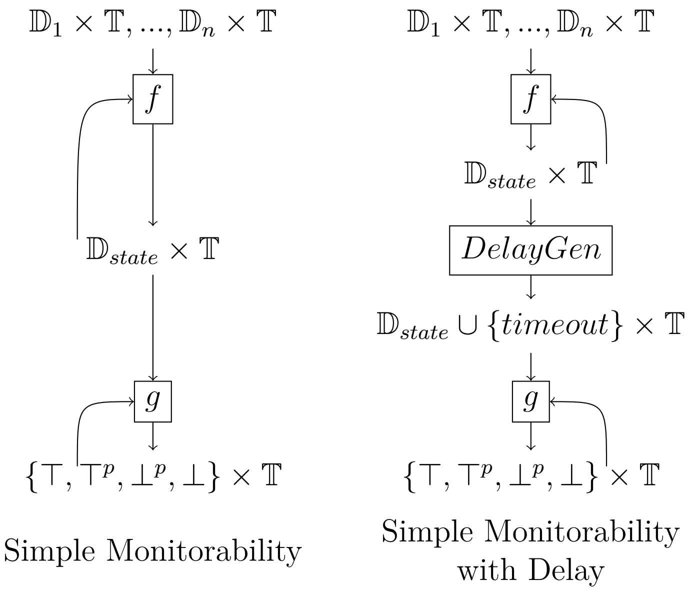

Figure 3.1: Overview Simple Monitorability - with or without *delay*

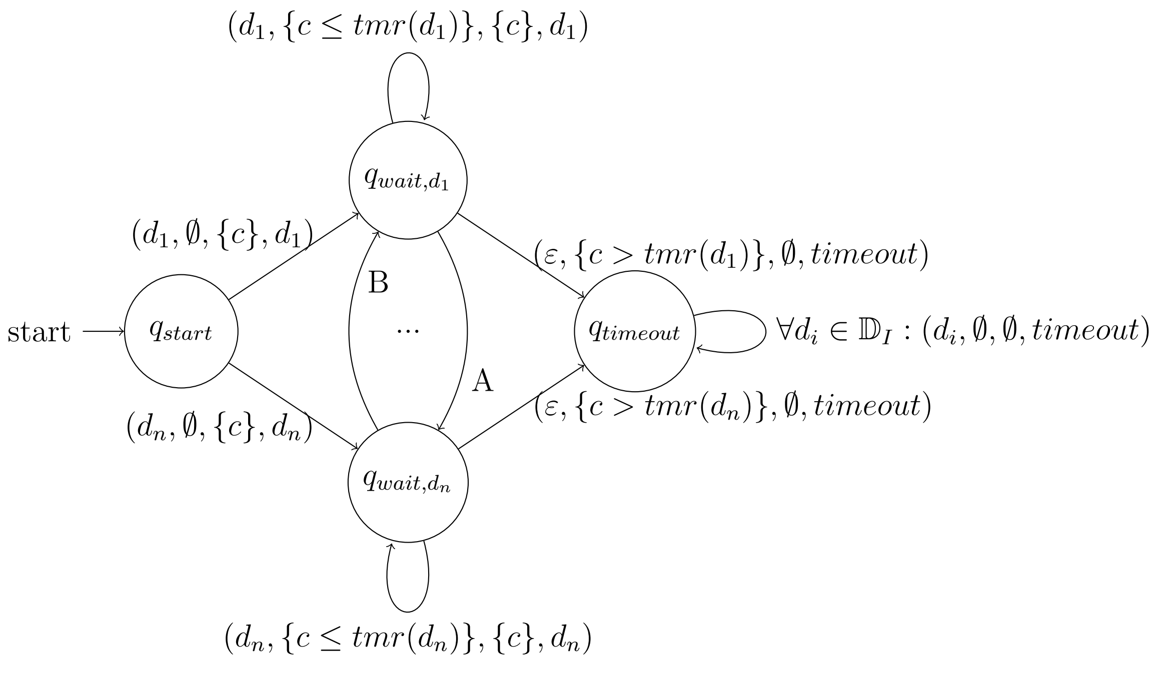

Figure 3.2: Visualization of the Delay Generator. Description A means $(d_n, \{c < tmr(d_1)\}, \{c\}, d_n)$ and description B means $(d_1, \{c < tmr(d_n)\}, \{c\}, d_1)$.

the input trace, a set of input streams can always be constructed, which requires an arbitrary large amount of storage space, which is larger than the available memory.

Not all TADL2 constraints are simple monitorable properties, even with delay, because they may require memory resources, which are not independent from the events of the observed trace. Like stated before, correct online monitoring of these constraints is impossible for arbitrary traces, because infinite memory resources may be required. On the other hand, many of these problems are solved by using finite resources, with the hope, that the available resources are enough to cover the inputs of the ”real world”. In these cases, a distinction is useful, because the memory or time 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 Not Simple Monitorable* and the others *Worst Case Not Simple Monitorable* for the rest of this thesis.
Obviously, the constraints with continuous resource requirement growth cannot be monitored infinitely, but the constraints that only need infinite resources in worst cases can be monitored in many cases.

4 Analysis of the Monitorability of Timing Constraints

4.1 Monitorability of the TADL2 Timing Constraints

In this chapter, each of the TADL2 constraints will classified into the classes *Simple Monitorable*, *Simple Monitorable with Delay* and *Not Simple Monitorable*, like defined in chapter 3. For the last class, it will be demonstrated, if the constraint is not simple monitorable in any cases or just in worst case scenarios.

4.1.1 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 at least one *target* event. Therefore, the state that need to be stored to monitor the *DelayConstraint* is the set of *source* events, that are younger than *upper* and did not have a matching *target* event yet. If this information is not stored, the constraint cannot be monitored correctly. Updates to this state and output of the monitor are done at *source* and *target* events and at delay timestamps *upper* after the oldest stored *source* 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* events occur in the timestamps $\{1, 1.1, ..., 5.9\}$ and *target* events in the timestamps $\{6, 6.1, ..., 11\}$. At timestamp 6, all 49 *source* events must be stored, because they are all required to generate the correct output in this and in following timestamps. At this timestamp, the oldest *source* event can be removed from the storage, because the matching *target* event occurs in this timestamp. Other timestamps cannot be removed from the storage, because they are younger than *lower*. With every following *target* event, the oldest event can be removed from the storage, until every *source* had its matching *target* event at

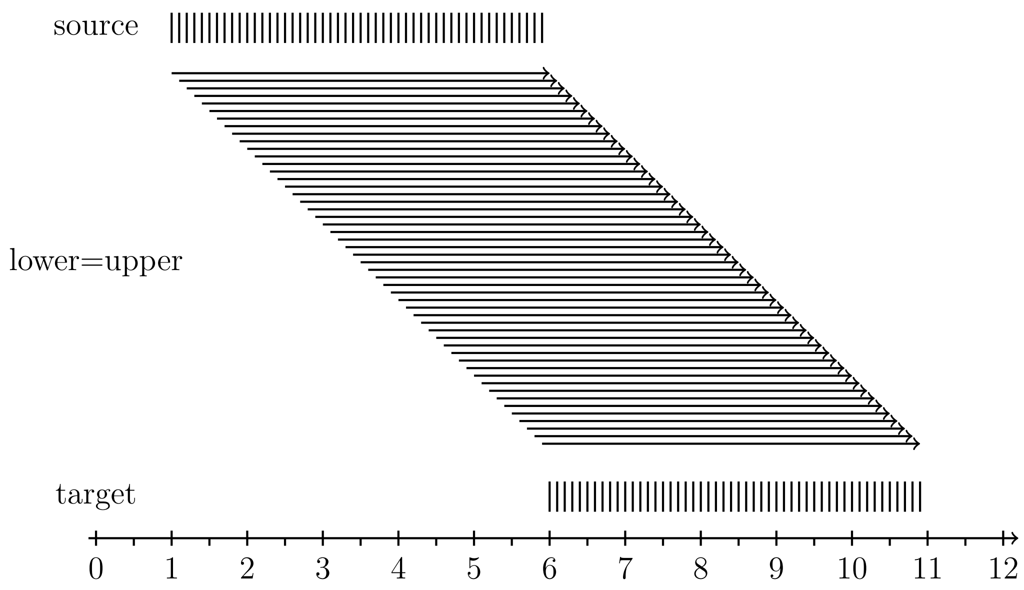

Figure 4.1: *DelayConstraint* or *StrongDelayConstraint* with $lower = upper = 5$

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*. Which means that the required storage space can grow infinitely, therefore, the worst case memory requirement is dependent of the events in the trace. Consecutively, the *DelayConstraint* is not *simple monitorable*.
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 not simple monitorable*.

4.1.2 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. Like before, all *source* events, that did not have a matching *target* event yet, must be stored, but at matching *target* events, only one *source* event can be removed from the storage. The worst case memory requirement remains unchanged and is still dependent of the number and the placement of the input events, therefore the *StrongDelayConstraint* is *worst case not simple monitorable* with the same argumentation as for the *DelayConstraint*.

4.1.3 RepeatConstraint

The *RepeatConstraint* defines the time distance between each event and its $span^{th}$ successor. Therefore, the state, that must be stored for monitoring, 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 otherwise. The delay offset can be calculated by the time distance between the current timestamp and the timestamp, that lays *upper* behind the oldest stores event.
Because 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 $\mathcal{O}(1)$ (e.g. if double linked lists are used), the *RepeatConstraint* is *simple monitorable with delay.*

4.1.4 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 the 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 in s 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
$upperBound := List\_append(last(List\_tail(UpperBound), s), upperBoundNow + upper)$
for the upper bound.

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 at the lists end. $lowerBoundNow$ 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 they are independent from the trace and the *RepetitionConstraint* is a property, which is *simple monitorable with delay.*

4.1.5 SynchronizationConstraint

The *SynchronizationConstraint* describe groups of streams, which events occur in common clusters. Each of these streams 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 section 4.1.1, an arbitrary number of events can be placed in this interval and the memory requirements are dependent of the input streams. The required storage space is not growing continuously, because the stored events can be removed at the end of the *tolerance* interval, therefore the *SynchronizationConstraint* is *worst case not simple 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.

4.1.6 StrongSynchronizationConstraint

The difference between the *StrongSynchronizationConstraint* and the *SynchronizationConstraint* is, that in the StrongSynchronizationConstraint, only one event per stream is allowed per synchronization cluster. Overlapping of these clusters is still

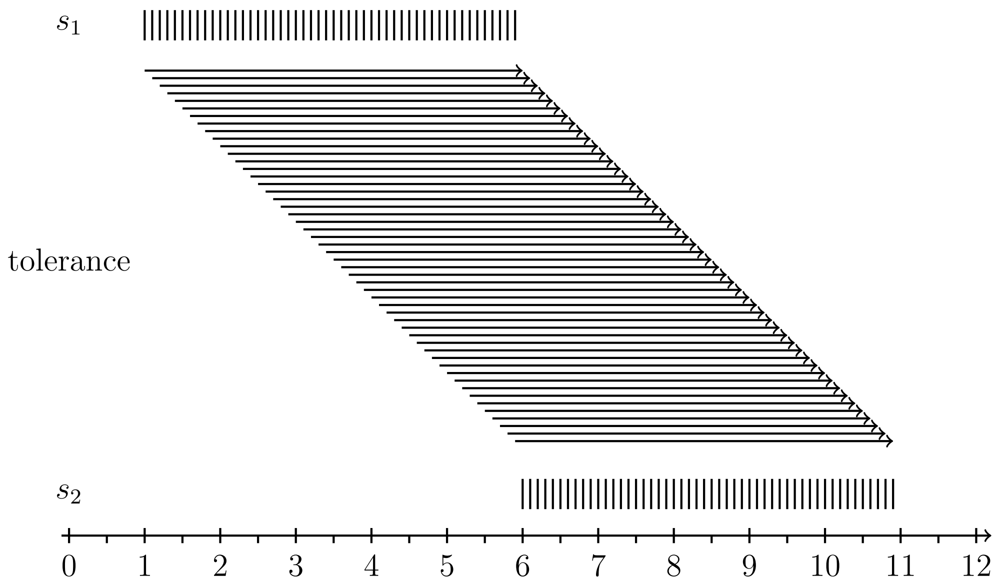

Figure 4.2: *SynchronizationConstraint* or *StrongSynchronizationConstraint* with $tolerance = 5$

possible. Therefore, this constraint can be classified as *worst case not simple monitorable* with the same argumentation as the previous constraint.

4.1.7 ExecutionTimeConstraint

The *ExecutionTimeConstraint* ensures that the time distance between *stop* and *start* events, not counting interruptions (which are specified by *preempt* and *resume* events) is between *lower* and *upper*.
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 followed by an *resume* event with no other event in between 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 the time distances between the *resume* and *preempt* events. This sum is resetted at every *start* event.
These values are updated on events in *start*, *preempt* and *resume*.
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 events in *stop*, additionally the runtime must be greater or equal to *lower*.
To monitor this constraint correctly, a delay is required, when a *stop* event is late

or missing. The delay duration is the distance between the current timestamp and *upper* minus *runtime* after the current timestamp.
The required storage space is fixed(remind, that we limited the memory size of timestamps in 3.2), also the runtime of the state transition and output function can be implemented with constant run time, consecutively the *ExecutionTimeConstraint* is *simple monitorable with delay.*

4.1.8 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. In a finite setting, it must also be checked that the number of *source* and *target* events is equal in the end of the observation. Because it is possible that an arbitrary large number of *source* events occur before the first *target* occurs, a possibly arbitrary large number must be stored, and the required storage space is dependent on the input streams. Because 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 not simple monitorable.*

4.1.9 ComparisonConstraint

The *ComparisonConstraint* defines an ordering relation between two single events and does not describe relations of streams or their events. Therefore, the definition of *simple monitorability* is not applicable. But because of the restrictions to timestamps made in section 3.2, the maximal required storage space and the run time of the operators $\leq, <, \geq, >, =$ have a fixed upper limit.

4.1.10 SporadicConstraint

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

4.1.11 PeriodicConstraint

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

4.1.12 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 input events occur after the strictly periodic timestamps of X. The distances between the elements of X and the following events is defined by *offset*.
This constraint can be monitored by storing the upper and lower limit of the current latest element of X and the number of event occurrences, reset to 0 at every $|offset|^{th}$ event. The limits of the elements of X can be narrowed down at 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 correct distance to the limitations of the current latest element of X. To be able to recognize late or missing events, a delay is required.The timestamp, where the delay must occur, can be calculated by adding *jitter* and the entry of *offset* for the next expected event to the current upper limit of latest X.
Because the memory requirements (two timestamps and an integer) are constant and the mentioned state transition, delay calculation and evaluation functions can be implemented in constant time, the *PatternConstraint* is *simple 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. Because the *RepeatConstraint* is *simple monitorable with delay* and the *upper* parameter is ∞, the constraint is *simple monitorable* (without delay) in this variant.

4.1.13 ArbitraryConstraint

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

$$\begin{aligned}&ArbitraryConstraint(event, minimum_1, ..., minimum_n, maximum_1, ..., maximum_n)\\&\Leftrightarrow \forall i \in 1, ..., n : RepeatConstraint(event, minimum_i, maximum_i, i).\end{aligned}$$

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

4.1.14 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 *simple monitorable with delay*. The *upper* parameter in both application of the *RepeatConstraint* is ∞, therefore the timeout always occurs infinite timestamps after the latest input events, which means, the timeout is dispensable and the *BurstConstraint* can be monitored without any delay or timeout operator. Consecutively, the *BurstConstraint* is a *simple monitorable* property.

4.1.15 EventChains

The *ReactionConstraint* and the following Constraints are defined on *EventChains*, which are defined as *stimulus* and *response* stream. Each event of these streams has a color attribute, which describes the causal connection of individual events. It is required, that any *stimulus* event of 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 must exist.
Monitoring this property is difficult, because it is required to store every color which 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 of the color black. After that, not black is allowed in *stimulus*. To check this properly for all colors, the color of all events, which occurred in *response* must be stored until the end of the observation.
The memory consumption to monitor this property is growing continuously with any event that introduces a new color in *response*, therefore the correctness of *EventChains* is a *always not simple monitorable* property.

4.1.16 ReactionConstraint

If we assume the correctness of the *EventChains*, a monitor would be similar to a monitor of the *DelayConstraint*. The only difference is that the color attribute of

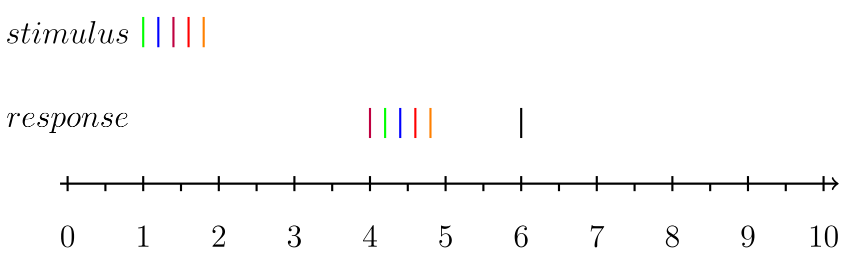

Figure 4.3: Event Chain example

the stored *stimulus* events must also be stored. Removing events from the storage is only possible, when the time distance and the color between the *stimulus* and *response* events is correct. Like for the *DelayConstraint*, the required worst case storage space is dependent on the input streams, therefore the *ReactionConstraint* is *worst case not simple monitorable*, if the correctness of the *EventChain* is assumed.

4.1.17 AgeConstraint

Similar to the analysis of the *ReactionConstraint*, we assume the correctness of the *EventChain*.
The *AgeConstraint* is very similar to the *ReactionConstraint*, the main difference is that every *response* event requires a *stimulus* event in a matching color in the right distance, not the other way around. Because the *response* events always occur after the *stimulus* event(s) in the same color, no delay is required, but the number of events in *stimulus*, that must be stored in worst cases, remains the same as in the *ReactionConstraint*. Therefore, the *AgeConstraint* is *worst case not simple monitorable*, if the correctness of the *EventChain* is assumed.

4.1.18 OutputSynchronizationConstraint

Again, the correctness of the *EventChain* is assumed in the analysis.
The definition of the *OutputSynchronizationConstraint* does not limit the time distance between *stimulus* events and their associated synchronization clusters in the *response* streams. If we only consider infinite streams, a missing synchronization cluster cannot make the constraint unsatisfied, therefore, only the correctness of these clusters may be false and lead to a negative output. The correctness of these synchronization clusters is not *simple monitorable*, which is argued the same way as in the simple *SynchronizationConstraint*. An arbitrary large number of new synchronization clusters can be placed in a time interval with the length *tolerance*, which has to be stored until the all *response* streams have fulfilled this cluster. A

key different to the *SynchronizationConstraint* is that only the first occurrences of each color must form a synchronization cluster. After this cluster, events of this color may occur independently. Because of this characteristic, the color of any finished synchronization cluster must be stored for the entire rest of the observation, which means, the *OutputSynchronizationConstraint* is a *always not simple monitorable* property.
If finite streams are considered, the color of all *stimulus* events must be stored, until the matching synchronization cluster occurs. At the end of the observation, it must be checked, if there was a synchronization cluster for all colors, which occurred in *stimulus*. The classification as *always not simple monitorable* is not affected by this.

4.1.19 InputSynchronizationConstraint

Like before, the correctness of the *EventChains* is assumed.
In the *InputSynchronizationConstraint*, synchronization clusters in the *stimulus* streams must only be fulfilled, if the associated events are the last of their color in their streams before an associated *response* event. This means, at least some information for every color, which occurred in the *stimulus* streams, must be stored, until the *response* color with the same color. Because several *response* events of this color may occur, the information about a fulfilled or unfulfilled synchronization cluster may not be removed from the storage. Otherwise it could not be checked correctly, if there was a synchronization cluster with a matching color. Consecutively, the required storage space grows continuously with every new stimulus color and the constraint is *always not simple monitorable*.

4.2 Conclusion

Figure 4.4 gives an overview, which TADL2 timing constraints are *simple monitorable* and which are not. The *ComparisonConstraint* is not defined on streams, therefore the definition of *simple Monitorability* is not applicable. All simple monitorable constraints, except the *BurstConstraint* require the creation of new timestamps. The other constraints are not simple monitorable, of which the *Input-* and *OutputSynchronizationConstraint* are always not simple monitorable. If the correct order of the *EventChains* is assumed, the *Reaction-* and *AgeConstraint* are only not simple monitorable in worst cases, like the other not simple monitorable constraints. 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 not simple monitorable constraints,

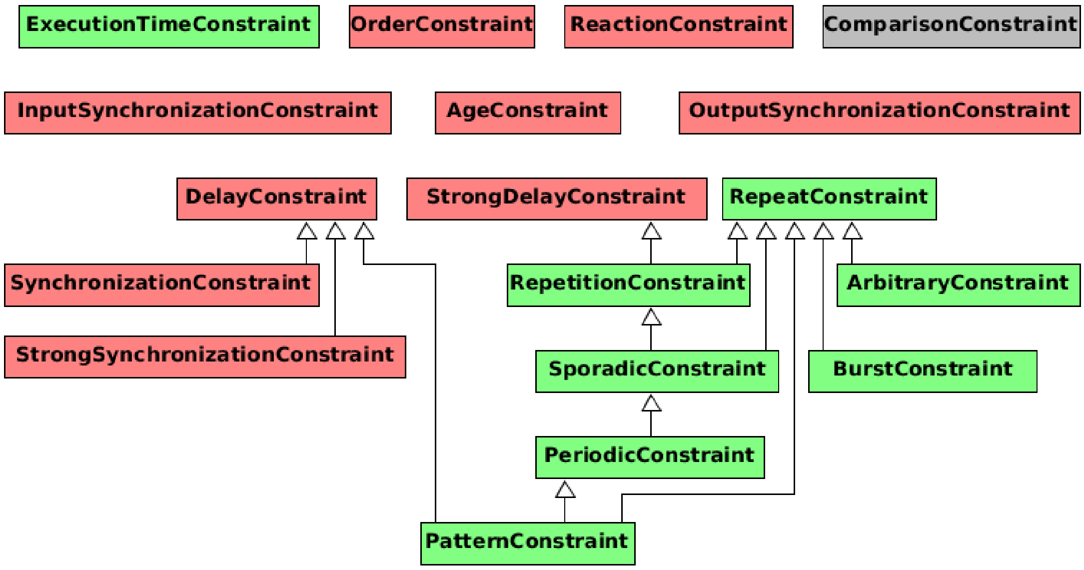

Figure 4.4: Overview over constraints - Simple Monitorable - Not Simple Monitorable

can be simple monitorable, because of further restrictions, which limit the required storage space or runtime.

5 Implementation

5.1 Implementation Of The TADL2 Constraints

In this chapter, the implementation of the monitor of each constraint will be explained. This is done by giving a short documentation of each monitor is given. Additionally, the worst case memory usage and the worst case and average run time per event is shown. In section 5.1.21, each monitor is run on traces, which were generated to match the constraints with specific parameters, to evaluate which performance can be expected in a practical usage of the implementation.
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 state transition function, the output function and the calculation of the required delay define the required time per timestamp with input events.
The implementations are programmed and tested for version 1.2.2 of the TeSSLa interpreter.

Output of the monitors

The monitors are outputting RV-LTL truth values ($\top, \bot, \top^p, \bot^p$), which are represented as two boolean variables. One of these variables is showing the truth value on the prefix, which was processed until this point in time, and the other variable shows, if the output possibly changes in upcoming timestamps. These two variables are packed inside of the type *fourValuedBoolean*. The individual values are mapped in the following way:

fourValuedBoolean.value	*fourValuedBoolean.final*	RV-LTL
true	true	$\top$
true	false	$\top^p$
false	false	$\bot^p$
false	true	$\bot$

Additionally to the state of the monitor, the previous output of the monitor is stored. If the previous output was $\perp$, the new output of the monitor is ignored and the output stays $\perp$. This is done to simplify the state and state transition of the monitor. For example in the *OrderConstraint*, the number of events, which occurred in the input streams, are stored as state. In the individual input timestamps, the correct order of the events can be checked in combination of the previous output of the monitor. If the previous output is unknown, the state must be defined more complex.

5.1.1 DelayConstraint

The implementation of the *DelayConstraint* monitor stores a 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 4.1.1, this list can grow infinitely long in worst cases, when the time domain is defined in an uncountable way. In these worst cases, an infinite number of *source* events may occur, before any event can be removed from the list, when a matching *target* event occurs.
The used TeSSLa version is using integer values as time domain, therefore it is countable and the list cannot grow infinitely, because at most *upper stimulus* events need to be stored and the largest possible length of the list is linear dependent on the parameter *upper*. Because this list is the only growable memory usage, the algorithm is in $\mathcal{O}(upper)$ in terms of memory.
In timestamps with a *target* event, all events in the list, which are in the right time distance, are removed from the list. This means that in worst cases, all events in the list must be checked and removed, which means, the worst case run time of the state transition is linear dependent of the length of the list and therefore is $\mathcal{O}(upper)$. 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*. In the first case, there is no *source* event without matching *target* event, therefore the output is $\top^p$. If list is not empty and the entry in the head of the list is younger than *upper*, the constraint is currently unsatisfied, but a satisfying state can still be reached. In this case, the output is $\perp^p$. If the entry in the head of the list is older than *tolerance*, there can't be a matching *target* event, therefore the output of the monitor is $\perp$. All these checks are done in constant time, therefore the output function 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* events, subtracted by the timestamp of the current event ($\mathcal{O}(1)$).

5.1.2 StrongDelayConstraint

The *StrongDelayConstraint* is implemented very similarly to the *DelayConstraint*. The only difference in the state transition is that exactly one event, which is the head of the list of unmatched *source* events, is removed, when a matching *target* event occurs. Therefore, the maximal memory usage is the same ($\mathcal{O}(upper)$), but the run time of the state transition is constant per input timestamp, because only the head of the list has to be considered in the transition.
The output function is nearly the same as in the previous constraint. The only difference is that in timestamps containing *target* events, it is checked if this event has a *source* event in the right distance. If not, the output is $\bot$. This check is also done in constant time, therefore the output function still is in $\mathcal{O}(1)$. The calculation of the delay period remains unchanged.

5.1.3 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 run time of the state transition function is linear dependent of *span* in terms of time and the memory usage is likewise.
The required delay is calculated by adding *upper* to the $span^{th}$ oldest event (or the first event, if there has been less than *span* events before) minus the current timestamp. Therefore, the time for the calculation is linear dependent on the *span* parameter, because the entire recursive definition of the state mentioned above must be walked trough.
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*. If this property is fulfilled, the output is $\top^p$ and in any other case it is $\bot$. Like in the calculation of the required delay, the entire recursive definition of the considered for the evaluation. Therefore, the output function is in $\mathcal{O}(span)$.

5.1.4 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. Relevant for the monitoring are the upper and lower 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 growing storage, therefore the algorithm is in $\mathcal{O}(span)$ in terms of memory. The run time of the state transition function is constant (removing the lists head and appending an entry to the lists).
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 $\top$, in any other case, it is $\bot$. 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 $\mathcal{O}(1)$.

5.1.5 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.
Because TeSSLa does not allow to define macros or functions with a variable number of input streams, events of each input timestamp must be placed into an integer list, which contains the index (starting at 1) of all streams, which have an event in this timestamp. This is list then used as a parameter to the implementation. The creation of this list is already implemented for up to 10 streams.
The implementation of the *SynchronizationConstraint* stores all events that occured not longer than *tolerance* ago in a list. In each entry, this list 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 in every input timestamp 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 the events older and one with the 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 occurrences. 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 the used TeSSLa version. 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 $\mathcal{O}(|event| * tolerance)$, because at most $|event|$ events must be appended to the list and the list has the maximum length *tolerance*. In worst cases, every event in the list(which is in ascending order) is older than tolerance, therefore the worst case runtime of 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 first, if there are any entries in the list of stored events, which were not part of a synchronization cluster yet. If this is not the case, the output is $\top^p$, because there are no unsatisfied synchronization clusters in this case. If there are entries without a synchronization cluster so far, it is checked, if all list entries, which were removed in this timestamp (and therefore are older than *tolerance*) had a synchronization cluster. If one of these removed entries didn't have a synchronization cluster, the constraint is unsatisfied and the output is $\perp$. If all of them were part of at least one cluster, the output is $\perp^p$, because there are still entries without cluster in the list (see first check), but they still can be satisfied. Because the list can have the size $|event| * tolerance$ and all of the entries are considered in the first check of the output function, the output function 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. The list is in ascending order, but the only unsatisfied events are relevant for the delay, which means, the entire list must be checked in worst cases. Therefore, the calculation of the required delay is in $\mathcal{O}(|event| * tolerance)$.

5.1.6 StrongSynchronizationConstraint

The *StrongSynchronizationConstraint* is defined as application of the *StrongDelay-Constraint*, but this application cannot be used for the implementation, like in the

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

previous constraint. Similar to the *SynchronizationConstraint*, the events of the input streams must be merged into a list.
The difference between the *Synchronization-* and the *StrongSynchronizationConstraint* is that each event is part of exactly one synchronization cluster in the *StrongSynchronizationConstraint*. Therefore, the implementation is different to the implementation of the previous constraint. Not every event is stored separately, but information about synchronization clusters, containing their start time and in which stream an event occurred in this cluster, are stored.
The information about synchronization clusters are stored in a list, which contains a time expression, which marks the starting point and a map, containing a boolean variable for every input stream, which shows, if there already was an event in this stream for this cluster. At event occurrences, the event is either added to a existing synchronization cluster, or a new cluster with the start time of the event is added to the list. For the search of a matching cluster, each event of the list is considered in worst cases, therefore the run time of this part of the state transition is linear to the number of active clusters. In worst cases this number is *tolerance*, when one event occurs in every timestamp in always the same stream. In the second step of the state transition, for every stored cluster is checked, if it is fulfilled. If so, it is removed from the list. To check, if a cluster is fulfilled, one boolean check must be done for every input stream, therefore at most boolean $tolerance * |event|$ checks must be done and the worst case run time of the state transition is in $\mathcal{O}(tolerance * |event|)$. When the events occur in timewise separated synchronization clusters, the list is significantly shorter than *tolerance* and the run time may be expected to be linear to the number of input streams.
The list storing the clusters is at most *tolerance* long and the size of individual entries of the list is linear dependent on the number of streams, because they store a boolean variable for every stream. Because of these length restrictions of the list, the algorithm is in $\mathcal{O}(|event| * tolerance)$ in terms of memory.
The output function checks first, if the list of stored synchronization clusters is empty. If this is the case, the output is $\top^p$, because there are no unsatisfied synchronization clusters. If the list isn't empty, it is checked, if the oldest unsatisfied cluster , which is in the head of the list, is younger than tolerance. If so, the output is $\bot^p$, because the constraint is unsatisfied, but can be satisfied by upcoming events. If the oldest cluster is older than tolerance, the constraint is unsatisfied and cannot be satisfied by upcoming events, therefore the output us $\bot$. All these checks are done in constant 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 ($\mathcal{O}(1)$).

5.1.7 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 this operator and then summed up. The sum of these interruptions is reseted by every *start* event. The calculation of this sum with resets, a macro called *resetSum* was programmed, which is a modified version of TeSSLas *resetCount* operator.
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 second application (with *preempt* and *resumse* as parameters) of TeSSLa's *runtime* operator from the sum of the first applications (with *start* and *stop* as parameters) of this operator. If the runtime should be checked in timestamps without a *stop* event, the second 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 input streams and the delay stream together.
The resulting runtime must be smaller or equal to *upper* in any point of time and greater or equal to *lower* at *stop* events. If this is the case, the output is $\top^p$, in any other case it is $\bot$. The required delay is calculated subtracting the runtime so far from upper. All of these operations are simple arithmetic functions on timestamps, 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.

5.1.8 OrderConstraint

The implementation counts the number of events in the *source* and *target* stream and stores these numbers as state. This update is done in constant time and the required storage space is also constant. The output function compares the number of *source* and *target* events. If the number is equal, the constraint is fulfilled until this point in time and the output is $\top^p$. If the number *source* events is larger, the constraint is unsatisfied, but can be satisfied by upcoming events, therefore $\bot^p$ is the output. If the number of *target* events is larger, the order of the events is invalid, the constraint is unsatisfied and cannot be satisfied anymore. Therefore, the output is $\bot$ in these cases. The checks of the output function are also done in constant time.
The introduction of new timestamps is not required for this constraint, therefore no delay period must be calculated.

5.1.9 ComparisonConstraint

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

5.1.10 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 $\mathcal{O}(span)$ in terms of time and memory. Because *span* is fixed to 1 in the *SporadicConstraint*, the implementation is in $\mathcal{O}(1)$ in terms of memory and time.

5.1.11 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.

5.1.12 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 of the *RepeatConstraint* is in $\mathcal{O}(span)$ in terms of time memory. The *span* attribute is set to 1 in the application, therefore the run time and memory usage is constant in this part.
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

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

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 access of the map entries is done in constant time, therefore the calculation of these new borders is also done in constant time and the state transition function in $\mathcal{O}(1)$ in terms of time.
The output function checks, if the timestamp of the current event is between the lower bound plus $\mathit{offset}_{count(event) \bmod |\mathit{offset}|}$ and the upper bound plus $\mathit{offset}_{count(event) \bmod |\mathit{offset}|}$ plus $jitter$. If so, the output is $\top^p$, if not it is $\perp$. The previous defined output is conjuncted with the output of the application of the *RepeatConstraint*. The comparisons of timestamps are done in constant time and monitoring the *RepeatConstraint* with $\mathit{span} = 1$ if likewise. Therefore, the output function is in $\mathcal{O}(1)$.
The required delay is defined by the time distance between the current timestamp and the upper bound for X, plus the expected offset of the following event, plus the allowed 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 map, is not limited in size and the complete algorithm, including the parameters, is $\mathcal{O}(|\mathit{offset}|)$ in terms of memory.

5.1.13 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 $\mathcal{O}(|minimum|)$ in terms of time. The memory usage of the *RepeatConstraint* is in $\mathcal{O}(\mathit{span})$. In the application of the *RepeatConstraint*, the *span* parameter increases for each of the $|minimum| = |maximum|$ applications. Therefore, the implementation is in $\mathcal{O}(\sum_{i=1}^{|minimum|} i) \widehat{=} \mathcal{O}(|minimum|^2 + |minimum|)$ in terms of time.

5.1.14 BurstConstraint

The *BurstConstraint* is defined as twofold application of the *RepeatConstraint* and is also implemented this way. The *RepeatConstraint* is in $\mathcal{O}(\mathit{span})$ in terms of time and memory. Because the *span* attribute is set to 1 and $maxOccurrences$ in the applications of the *RepeatConstraint*, the implementation of the *BurstConstraint* is in $\mathcal{O}(maxOccurrences)$ in terms of memory and time.

5.1.15 ReactionConstraint

The correctness of the *EventChain* is assumed in the implementation. If this property is unknown, it must be checked individually.
The implementation of the *ReactionCostraint* stores a map, which maps the color of *stimulus* events, which did not have a matching *response* event yet, to their timestamps. This state is updated at every input event. *Stimulus* events are inserted into the map, *response* events remove, if possible, a event from the map called above. Similar to the *DelayConstraint*(the *ReactionCostraint* can be seen 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*. Therefore, the algorithm is in $\mathcal{O}(maximum)$ in terms of memory. The state transition (insertion, 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 unsorted, every entry of the map must be considered for this. Therefore, the calculation of the required delay is in the time complexity class $\mathcal{O}(maximum)$.
The output function first checks, if the map of unmatched *stimulus* events is empty. If so, the constraint is satisfied and the output is $\top^p$. If there are entries in the map and the oldest entry is older than *tolerance*, the constraint is unsatisfied and cannot be satisfied by upcoming events. In this case, the output is $\bot$. If the oldest entry is younger than *tolerance*, the constraint is currently unsatisfied, but can be satisfied by upcoming events. Therefore, the output is $\bot^p$. To find the oldest entry in the map, all entries must be considered. Therefore, the output function is linear dependent on the size of this map, which is at most *maximum*.

5.1.16 AgeConstraint

Like before, the correctness of the *EventChain* is assumed in the implementation. If this property is unknown, it must be checked individually.
Similar to the implementation of the *ReactionCostraint*, the *AgeConstraint* monitor stores a map containing the latest *stimulus* event, which are younger than *maximum*. The *color* value is used as map key and the timestamp is used as map value. This map has the maximal size *maximum* and is updated at every input event. *Stimulus* events are inserted or updated, and entries, that are older than *maximum* are removed. To make this update faster, a list containing the colors of the events in the map is stored additionally. The maximal size of this list is also *maximum* and the colors are stored in chronological order, so that the color, that occurred the longest time ago, is in the head of the list. The update is done by looking at the head of the list and removing this entry from the list and the corresponding entry

with the same color from the map, if the entry is older than *maximum*. These operations are done in constant time, but need to be repeated, as long as the color in the head of the map is too old, so at most *maximum* times. Inserting or updating the *stimulus* event to the map is done in effectively constant time, but inserting or updating the list requires to remove any previous entry with the color of the current event. For this, every entry in the map has to be processed, which means this operation takes *maximum* steps in worst cases. Consecutively, the state and the state transition is in $\mathcal{O}(maximum)$ in terms of memory and time. The creation of new timestamps is not needed in this constraint, because only previous events need to be considered, upcoming events not.
In timestamps containing a *response* event, the output function checks, 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*. if so, the output is $\top^p$, if not, it is $\bot$. Timestamps without *response* events cannot lead to a violation of the constraint. The lookup in the map and the comparisons are done in constant time.

5.1.17 OutputSynchronizationConstraint

Similar to the *Synchronization*- and *StrongSynchronizationConstraint*, the input streams cannot be directly used as parameter. For the *OutputSynchronizationConstraint*, a stream of maps must be created, which represents the events of each timestamp. The key of each entry is the index of the stream (0 for the *stimulus* stream, 1, 2, ... for the *response* streams), in which the event occurred and the value is the color of the event. Again, the creation of this map is already implemented for up to 10 *response* streams.
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 this cluster and the *stimlus* event defined, it just has to be before the end of the streams.
The implementation of the *OutputSynchronizationConstraint* is storing three different information as state. First, a set of the stimulus colors, which did not have a *response* event in the same color yet. This set is updated at every input event, the color of *stimulus* events is inserted and the colors of the *response* events in the current timestamp are removed from the set. These updates are done in constant time, in worst cases, where no matching *response* events are occurring, the required storage space is linear dependent on the number of *stimulus* events.
The second information is a map, which is containing information about all synchronization clusters that were not finished before this point in time. This map is using the color attribute as key and the start timestamp and a map as value. This inner

map uses the indices of the *response* streams as keys and a boolean variable as value. This value 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 of these *response* events, 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, if the color of this event was in the set of stimulus colors of the previous timestamp. The check per event (two lookups in maps, one on set) is done in constant time, therefore the entire update of this map is in $\mathcal{O}(|response|)$ in terms of time per input timestamp. 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 are 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.
In combination, the run time of the state transition is in $\mathcal{O}(tolerance * |response|^2)$ and the memory usage is in $\mathcal{O}(count(stimulus) + tolerance * |response|^2)$.
The required delay is calculated by adding *tolerance* to the start time of the oldest unfinished cluster and subtracting the current timestamp. To get the oldest unfinished synchronization cluster, all map currently active clusters must be considered, which means the run time is linear dependent on the number number currently active clusters (at most $tolerance * |response|$).
The output function checks first, if the set of unmatched *stimulus* and the list of map of stored stored synchronization clusters are empty. If this is the case, not *stimulus* events had a fulfilled synchronization and the constraint is fulfilled until this point in time. The output is $\top^p$. If the set or the map are not empty it is checked, if the all synchronization clusters are younger than *tolerance*. If so, the constraint is currently unsatisfied, but can be satisfied by future events. In this case, the output is $\bot^p$. If the oldest synchronization cluster is older than *tolerance*, the constraint is unsatisfied and no future events can't change this. Therefore, the output is $\bot$ in this case. The first check is done in constant time, the second check requires to consider each stored synchronization cluster. Therefore, the run time of the output is linear dependent on the number of stored events, which is at most $tolerance * |response|$.

5.1.18 InputSynchronizationConstraint

The input streams must be transformed into a $map[Int, Int]$ stream, similar to the previous constraint, but this time the index 0 indicates the *response* stream and the indices 1, 2, ... are indicating the *stimulus* streams.
The *InputSynchronizationConstraint* is defined very similar to the *OutputSynchronizationConstraint*. The difference is, that the synchronization occurs in a set of *stimulus* events, not in *response* events.
Despite the similarities, the implementation of the *InputSynchronizationConstraint* is different to the implementation of the *OutputSynchronizationConstraint*. As state, 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 as value. 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 new inner map entry is created for this color. The lookup, if there already is a matching entry in the map for this color in this stream and possibly its update is done in constant time, but the time for initializing a new entry is linear dependent on the number of *stimulus* streams. Because $|stimulus|$ events may occur and introduce a new color in each timestamp, the state transition is in $\mathcal{O}(|stimulus|^2)$ in terms of time. The worst case memory size of this information is in $\mathcal{O}(|stimulus| * count(stimulus))$, because the map described above possibly stores every input event of the *stimulus* streams, when they introduce a new color and therefore a new entry in the inner map of the stream must be created. *Response* events are not considered for the state of the monitor.
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.
In timestamps containing a *response* event the output function checks,if the last occurrences of the corresponding color in the *stimulus* stream form a valid synchronization cluster. This is done by searching the youngest and oldest event with this color in the map of latest *stimulus* events. If a event of this color is missing, the age is interpreted as ∞ or $-\infty$, which leads to a length of the synchronization cluster that is definitely longer than $tolerance$. If the synchronization cluster is longer than $tolerance$, the constraint is violated and the output is $\perp$. If the cluster is not longer than $tolerance$, the output is $\top^p$. In timestamps without a *response*, the output remains unchanged. Because the color value is the key of the inner map, the time for searching the oldest and youngest event of this color is linear to the number of *stimulus* streams. Therefore, the output function is in $\mathcal{O}(|stimulus|)$ in terms of time.

5.1.19 EventChain

Additionally to the 18 TADL2 timing constraints, a monitor, which checks the correctness of *EventChains* was implemented. A *EventChain* is defined on a *stimulus* and a *response* stream as:

$\forall x \in stimulus : \forall y \in response : x.color = y.color \Rightarrow x < y$

As a state, a set, which contains all colors that previously occurred in *reponse* is stored. This set is updated at each *response* event by a an insertion into a set ($\mathcal{O}(1)$). The maximal size of this map is the number of events in *response*, therefore the state is in $\mathcal{O}(count(response))$ in terms of memory.
The output function checks, if the color every occurring *stimulus* event is not in the set of *response* colors, which is checked in constant time. If the color is in the set of *response* colors, the output is $\perp$. Otherwise, it is $\top^p$.

5.1.20 Conclusion

Table 5.1.20 gives an overview of the worst case memory consumption and the worst case run time per input timestamp. The worst case memory requirement and the runtime per input timestamp of the *Repeat-*, *Repetition-*, *ExecutionTime-*, *Sporadic-*, *Periodic-*, *Pattern-*, *Arbitrary-* and *BurstConstraint*, which are the *simple monitorable* constraints, are either constant, or they are only limited by the parameters of the constraint, not by the input traces. The implementations of the *Delay-*, *StrongDelay-*, *Synchronization-*, *StrongSynchronization-*, *Reaction-* and *AgeConstraint* are limited by the events, which may occur in time intervals of a specific length. Monitoring the correctness of *EventChains*, the *OutputSynchronization-* or the *InputSynchronizationConstraint* with these implementations require continuously growing memory resources and in the *OutputSynchronizationConstraint*, the run time per input timestamp is continuously growing too. The implementation of the *OrderConstraint* is in $\mathcal{O}(1)$ in terms of memory and time per event, although it is classified as *Not simple monitorable*. This is, because integers of a fixed length are used for the implementation of the constraint and only a finite subset of all streams that fulfill the constraint can be monitored correctly.

	Memory	Run Time per Input Timestamp
DelayConstraint	$\mathcal{O}(upper)$	$\mathcal{O}(upper)$
StrongDelayConstraint	$\mathcal{O}(upper)$	$\mathcal{O}(1)$
RepeatConstraint	$\mathcal{O}(span)$	$\mathcal{O}(span)$
RepetitionConstraint	$\mathcal{O}(span)$	$\mathcal{O}(1)$
SynchronizationConstraint	$\mathcal{O}(\|event\| * tolerance)$	$\mathcal{O}(\|event\| * tolerance)$
StrongSynchronizationConstraint	$\mathcal{O}(\|event\| * tolerance)$	$\mathcal{O}(\|event\| * tolerance)$
ExecutionTimeConstraint	$\mathcal{O}(1)$	$\mathcal{O}(1)$
OrderConstraint	$\mathcal{O}(1)$	$\mathcal{O}(1)$
SporadicConstraint	$\mathcal{O}(1)$	$\mathcal{O}(1)$
PeriodicConstraint	$\mathcal{O}(1)$	$\mathcal{O}(1)$
PatternConstraint	$\mathcal{O}(1)$	$\mathcal{O}(1)$
ArbitraryConstraint	$\mathcal{O}(\|minimum\|)$	$\mathcal{O}(\|minimum\|^2 + \|minimum\|)$
BurstConstraint	$\mathcal{O}(maxOccurrences)$	$\mathcal{O}(maxOccurrences)$
ReactionConstraint	$\mathcal{O}(maximum)$	$\mathcal{O}(maximum)$
AgeConstraint	$\mathcal{O}(maximum)$	$\mathcal{O}(maximum)$
OutputSynchronizationConstraint	$\mathcal{O}(count(stimulus) + tolerance * \|response\|^2)$	$\mathcal{O}(tolerance * \|response\|^2)$
InputSynchronizationConstraint	$\mathcal{O}(\|stimulus\| * count(stimulus))$	$\mathcal{O}(\|stimulus\|^2)$
EventChain	$\mathcal{O}(count(response))$	$\mathcal{O}(1)$

Table 5.1: Worst Case Run Times of the Implementations

5.1.21 Performance Analysis

To get an overview of the capabilities of the monitor implementations, each of them were run on at least 100 traces with 10.000 events, which were generated by following specific parameters to show, which of these parameters result in faster or slower run times. For this evaluation, the TeSSLa interpreter version 1.2.2 were used and it was run on a computer with a i5-6600k processor running on 4.3 GHz. The operating system was Windows 10.0.19041.0.
The run times were measured as time between the input of all events of all timestamps and the associated output of the TeSSLa interpreter. For that, a program[2] was written, which generates traces for each constraint and then measures the time between the input of the events of one timestamp and the output of the TeSSLa interpreter. The communication between the test program and the TeSSLa interpreter is done via the *standard input* and *standard output stream* of the interpreter. The time is measured by the java function *System.nanoTime()* immediately before the events of one timestamps are written into the input stream and immediately after an reaction was received on the output stream. It must be noted that this time measurement is not completely accurate, because neither the used java runtime environment nor the operating system were build to fulfill real time requirements. Therefore, unpredictable delays may occur in the test program, in the java interpreter or between them, but the averages of the results show, what the monitors are capable of and on which input parameters the run time significantly rises.
A shortened version of the results will be shown here, the complete results of the runtime measurement can be accessed at https://github.com/HendrikStreichhahn/TeSSLa-Autosar-Timing-Extensions/tree/master/traceGenerator/results.

DelayConstraint

The *DelayConstraint* was evaluated with 100 Traces of 10.000 events. The traces fulfilled the constraint with the parameters $lower \in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$ and $upper = lower$. The distance of subsequent *source* event were 2^i, with $i \in \{0, 1, ..., 10\}$, while the distance between subsequent *source* events in each trace was smaller than $2 * lower$. The shorter the distances between the *source* event are, the more (at most *upper*, when the distance is 1) events are stored as state.
Figure 5.1 shows the average run time of the monitor in dependency of $lower$ and *upper* for traces with event distances of 1, which means that *upper* events are stored as state of the monitor. The run time is nearly constant, because the trace generator does not create worst case scenarios and only one event must be removed from the

[2]This program can be found at https://github.com/HendrikStreichhahn/TeSSLa-Autosar-Timing-Extensions/tree/master/traceGenerator

list at every *target* event.
Figure 5.2 shows the average run times for this constraint with the parameters $lower = upper = 800$ in dependency of the distance of subsequent *source* events. Two clusters can be observed. The average run times of traces with event distances of $2^0, 2^1, 2^2, 2^3, 2^4, 2^5$ are higher than the run times of the other traces. This is, because in the first six traces, there are timestamps with two events, and in the traces, each timestamp has at most one event. This can be shown by a simple equation

Proof. Let $lower = upper$ be the distance between *source* events and their associated *target* event.
Let $s \in \mathbb{N}_0$ be the first timestamp with an *source* event in the trace.
Let $dist \in \mathbb{N}$ be the distance between subsequent *source* events.
The placement of all *source* events is given by: $s + x * dist$ with $x \in \mathbb{N}$
The placement of all *target* events is given by: $s + y * dist + upper$ with $y \in \mathbb{N}, y < x$
All placements of *source* and *target* events, which occur in common timestamps, fulfilled the equation:

$$\begin{aligned} s + x * dist &= s + y * dist + upper \\ x * dist &= y * dist + upper \\ x &= y + \frac{upper}{dist} \end{aligned}$$

When $upper = 800$, there is no integer solution for x and y for $dist \in 64, 128, 256, 512, 1024$, all events occur in individual timestamps for these distance between *source* events. When $dist \in \{1, 2, 4, 8, 16, 32\}$, there is an integer solution for x and y, so there multiple events in individual timestamps. □

StrongDelayConstraint

The traces for the evaluation of the *StrongDelayConstraint* were generated with the same parameters as for the previous constraint. In figure 5.3 the average run times with fixed *source* event distances is shown. The results are nearly constant. Figure 5.4 shows the average run times for traces, where *lower* and *upper* is fixed at 700 and the distance between subsequent *source* events is varying. It can be seen that the run times for the traces is separated into two areas, one cluster containing the traces with a *source* event distance of $2^0, 2^1$ and 2^2 and one containing the other traces. This clustering has the same reason as in the *DelayConstraint*, it occurs, because in some traces, there are many timestamps with multiple events and in some are not.

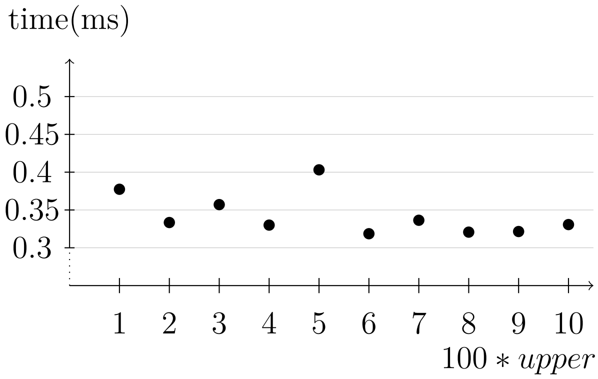

Figure 5.1: Average run times of the *DelayConstraint* with event distances of $2^0 = 1$

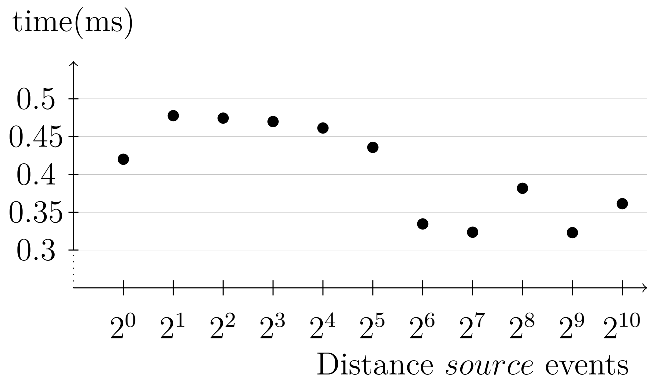

Figure 5.2: Average run times of the *DelayConstraint* with the parameters $lower = upper = 800$

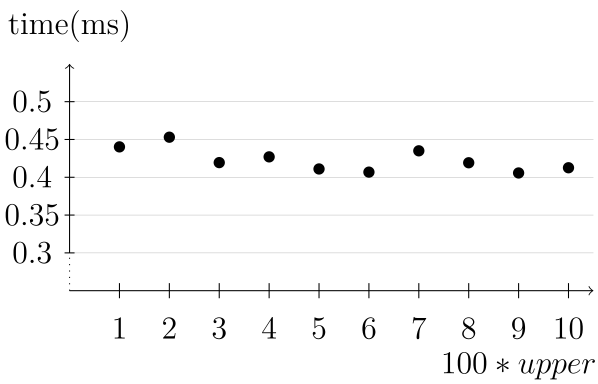

Figure 5.3: Average run times of the *StrongDelayConstraint* with event distances of $2^0 = 1$

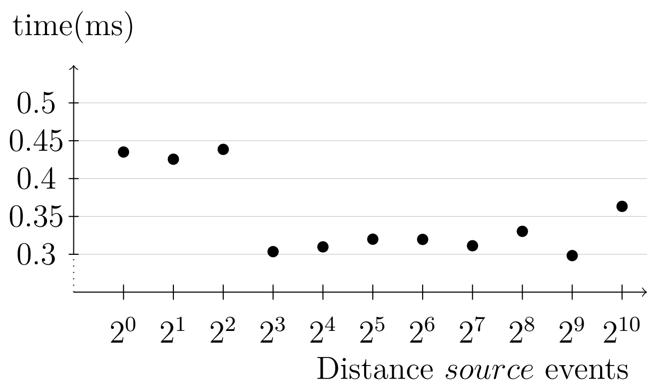

Figure 5.4: Average run times of the *StrongDelayConstraint* with the parameters $lower = upper = 700$

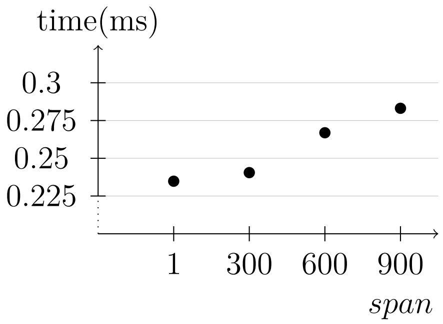

Figure 5.5: Average run times of the *RepeatConstraint* with the parameters $lower = 5000, upper = 7000$

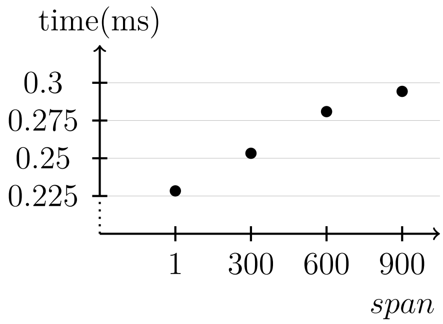

Figure 5.6: Average run times of the *RepeatConstraint* with the parameters $lower = 8000, upper = 9000$

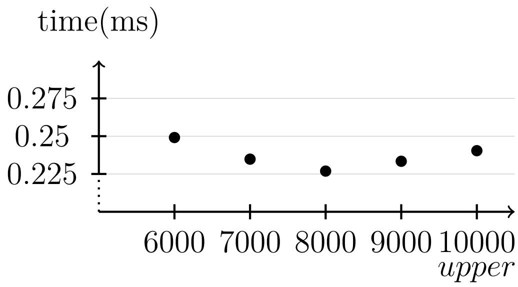

Figure 5.7: Average run times of the *RepeatConstraint* with the parameters $span = 1, lower = 5000$

RepeatConstraint

The *RepeatConstraint* was evaluated with 100 Traces of 10.000 events. The traces were created with the attributes $span \in \{1, 300, 600, 900\}$, $lower = \{5000, 6000, 7000, 8000, 9000\}$ and $upper = lower + x$, $x \in \{1000, 2000, 3000, 4000\}$. Figure 5.7 shows the average run time with fixed *span* and *lower* parameters and a variable value for *upper*. It can be seen, that the run times are nearly constant, which matches with the analysis of the implementation.
Figure 5.5 and 5.6 show the average run time of the *RepeatConstraint* monitor with the parameters $lower = 5000(8000)$, $upper = 7000(90000)$. By the analysis of the implementation, a runtime that is linear dependent on *span* was expected, which can be slightly seen in the results. The increase is not much larger than the deviations between individual measures, but can be seen by nearly all runs with different values for *lower* and *upper*.

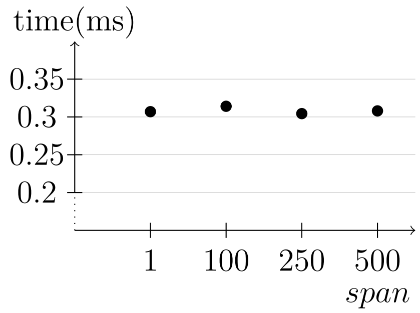

Figure 5.8: Average run times of the *RepetitionConstraint* with the parameters $lower = 500, upper = 900$

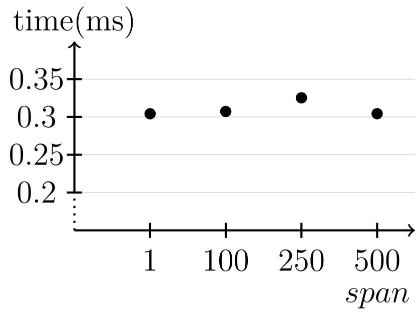

Figure 5.9: Average run times of the *RepetitionConstraint* with the parameters $lower = 700, upper = 1100$

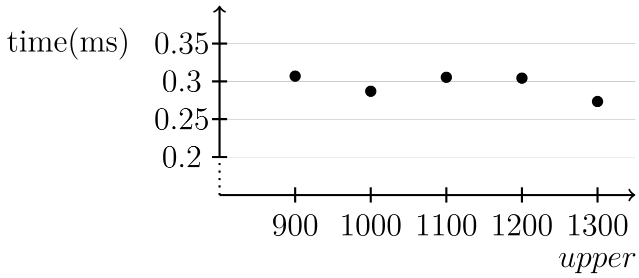

Figure 5.10: Average run times of the *RepetitionConstraint* with the parameters $span = 1, lower = 500$

RepetitionConstraint

The traces for this constraint were created with the parameters $span \in \{1, 100, 250, 500\}$, $lower = \{500, 600, 700, 800, 900\}$ $upper = lower + x$, $x \in 400, 500, 600, 700, 800$ and $jitter = \frac{lower}{2}$.
Figure 5.8 and 5.9 are showing the average run times of the monitor with the parameters $lower = 500(700)$ and $upper = 900(1100)$ with different values of the *span* parameter. Figure 5.10 shows the average run time in dependency of the *upper* parameter. Like expected in the analysis, the parameters did not an influence to the run times and the run times are nearly constant.

SynchronizationConstraint

Figure 5.11 shows the average run times of the *SynchronizationConstraint* monitor, which was checking traces with three event streams with two events per synchronization cluster in each stream. The synchronization clusters were 200 timestamps apart, so they did not overlap. The run times were nearly constant, which was

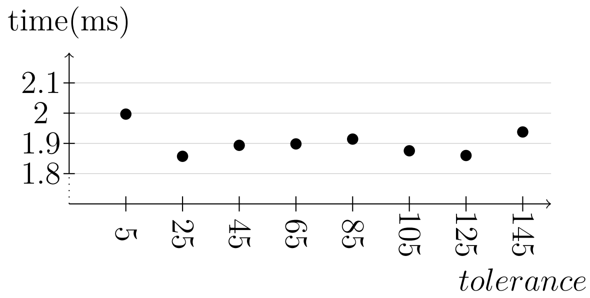

Figure 5.11: Average run times of the *SynchronizationConstraint* with three event streams and two events per cluster and stream and a cluster distance of 200

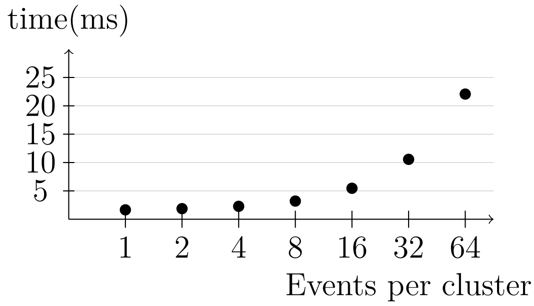

Figure 5.12: Average run times of the *SynchronizationConstraint* with three event streams, $tolerance = 91$ and a cluster distance of 182

expected for these parameters, because at most 3 (the number of input streams) events were stored and considered in each input time stamp.
Figure 5.12 shows the average run times of the monitor with traces of three streams, a *tolerance* value of 91 and a distance between clusters of 182. So again, the clusters were not overlapping. It can be seen, that the run times grow linear when increasing the number of events in each cluster. This matches with the expectation, because the more events occur in an interval of the length *tolerance*, the more events need to be stored and considered in each input timestamp.

StrongSynchronizationConstraint

Figure 5.13 shows the average run time of the monitor with the parameters $tolerance = 37$ and a cluster distance of 2, so that 19 clusters are overlapping. It can be seen, that the run time increases, when more input streams are used. In Figure 5.14, a fixed number of input streams was used and the cluster distance was 2, like before. Again, an increase in the run times can be seen. This matches with the expectations of the analysis, in which the run time was said to be in $\mathcal{O}(|event| * tolerance)$.

ExecutionTimeConstraint

The run time evaluation of the *ExecutionTimeConstraint* monitor was done by traces, which fulfill the constraint the parameters $lower \in \{100, 300, 500, 700, 900\}$ and $upper = lower + x$, $x \in \{100, 600, 1100, 1600, 2100\}$. For each of combination of these parameters, one trace with 1, 11, 21 and 31 preemptions between the *start* and *end* event were created. In figure 5.15 the average run time with fixed *lower*

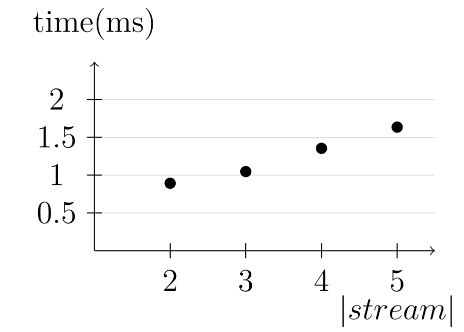

Figure 5.13: Average run times of the *StrongSynchronizationConstraint* with $tolerance = 37$ and a cluster distance of 2

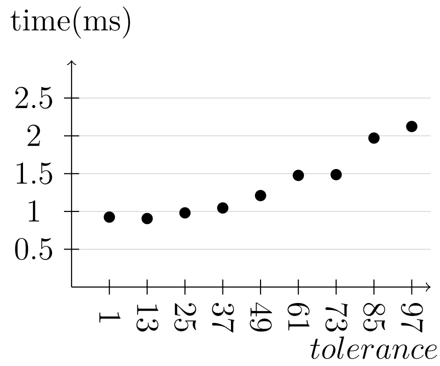

Figure 5.14: Average run times of the *StrongSynchronizationConstraint* with three event streams and a cluster distance of 2

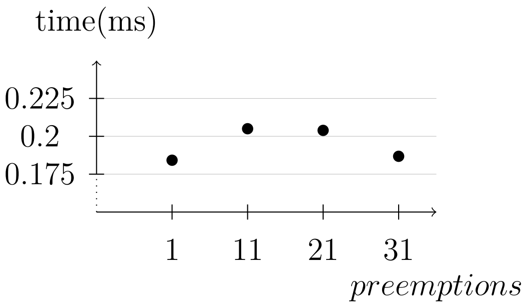

Figure 5.15: Average run times of the *ExecutionTimeConstraint* with the parameters $lower = 100, upper = 200$

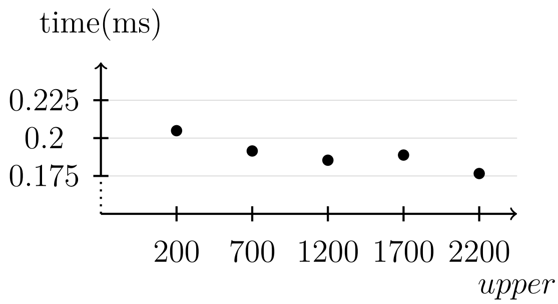

Figure 5.16: Average run times of the *ExecutionTimeConstraint* with the parameters $lower = 100, preemptions = 11$

and *upper* can be seen. In figure 5.16, *lower* and the number of preemptions is fixed. A correlation between the input parameters and the run times can not be observed, which was expected, because the run time is independent from the parameters or the placement of events, like stated in chapter 5.

OrderConstraint

The *OrderConstraint* monitor was evaluated on traces with distances between subsequent *source* events between 1 and 91 in steps of 10 and maximal distances between the i^{th} *source* and *target* event between 0 and 45 in steps of 5. In traces, where the distance between the *source* events and their associated *target* events were 0 or the distance between subsequent *source* events were 1, the run time was circa double as large as in the other traces. The reason for this is that the smaller the

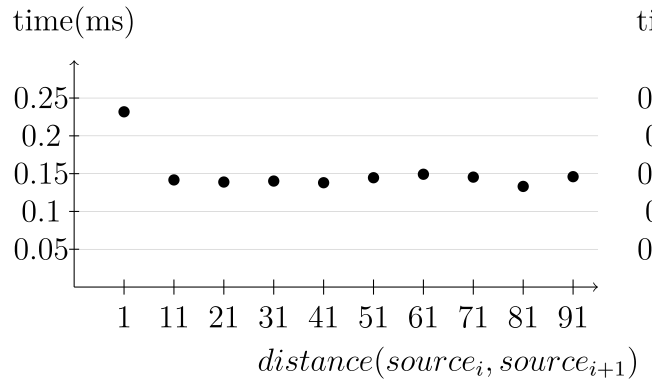

Figure 5.17: Average run times of the *Order-Constraint* with a distance between *source* events and their associated *target* events of 5 in dependency of the distance between subsequent *source* events

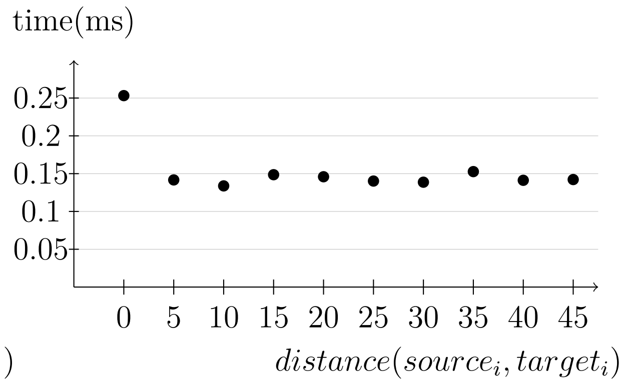

Figure 5.18: Average run times of the *Order-Constraint* with a distance between subsequent *source* events of 11 in dependency of the distance *source* events and their associated *target* event

distance between the *source* and *target* events are, the more often two events occur in the same time stamp, which means, that two events must be processed in one timestamp, instead of one, which requires more time.

SporadicConstraint

The traces, that were used for the evaluation fulfill the constraint with the parameters $jitter \in \{1, 11, 21, 31\}$, $lower \in \{500, 600, ..., 900\}$ and $upper = lower + x$, $x \in \{100, 200, ..., 500\}$. The average run time per timestamps of the monitor with the parameters $lower = 500$ and $upper = 600$ with different values for the *jitter* parameter can be seen in Figure 5.19. Similar to the run times with varying *upper* values (figure 5.20), the run times are nearly constant. Like expected by the analysis of the implementation in the previous section, the parameters had no influence on the run time.

PeriodicConstraint

The run time evaluation was done on traces, which fulfill the *PeriodicConstraint* with the parameters $period \in \{10, 20, 30, .., 100\}$ and $jitter \in \{0, 1, .., 9\}$. In figure 5.21 the average run times of the monitor with a constant *period* and a variable *jitter* can be seen, in figure 5.22, *jitter* is fixed and *period* is variable. Despite some

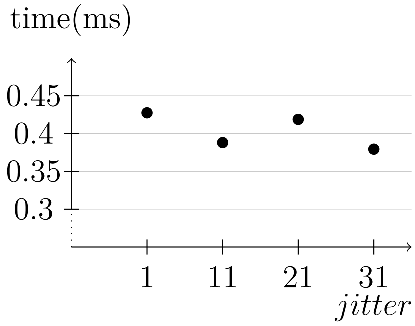

Figure 5.19: Average run times of the *SporadicConstraint* with the parameters $lower = 500, upper = 900$

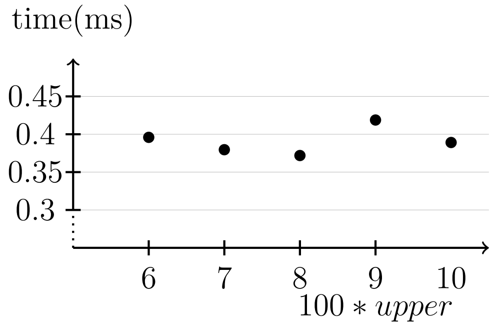

Figure 5.20: Average run times of the *SporadicConstraint* with the parameters $lower = 500, jitter = 21$

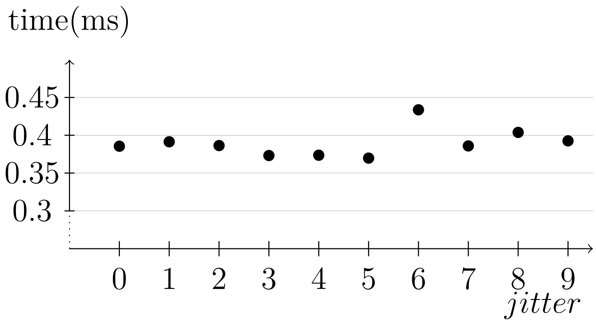

Figure 5.21: Average run times of the *Periodic-Constraint* with a *period* of 10 and variable *jitter*

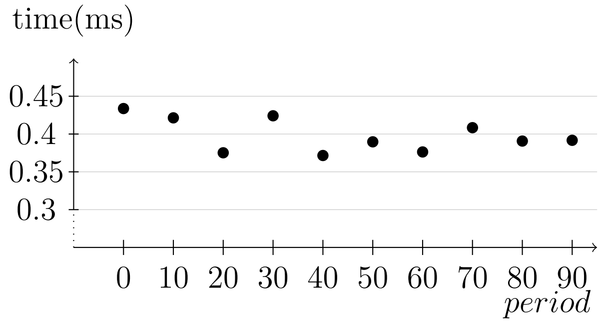

Figure 5.22: Average run times of the *Periodic-Constraint* with a *period* of 6 and variable *period*

fluctuations, the run time is constant and independent of the input parameters. This behaviour was expected by the complexity analysis.

PatternConstraint

The monitor of the *PatternConstraint* was first evaluated on traces with lengths of the $|offset|$ parameter of 1, 2 and 3 and varying values for the parameters *period* and *jitter*. Figure 5.23 and 5.24 are showing some of these results, which where nearly constant at around 0.65ms per input timestamp. After these run time measurements, the run time was measured on traces with the parameters $jitter = 0$ and $period = 200$. The *offset* parameter had an increasing length from 1 to 100 and was filled with $offset = [0, 1, 2, 3, ...]$ and the *period* parameter was set to 200. The results of this measurement can be seen in figure 5.25 .It can be seen, that the average run

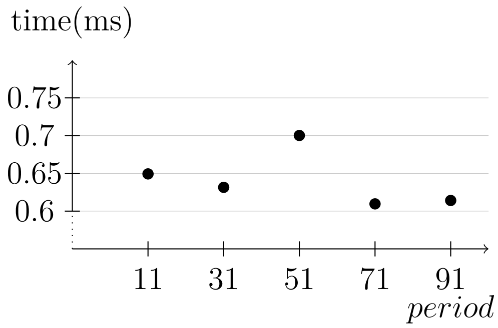

Figure 5.23: Average run times of the *PatternConstraint* with the parameters *offset*$= [0, 1]$ and $jitter = 0$

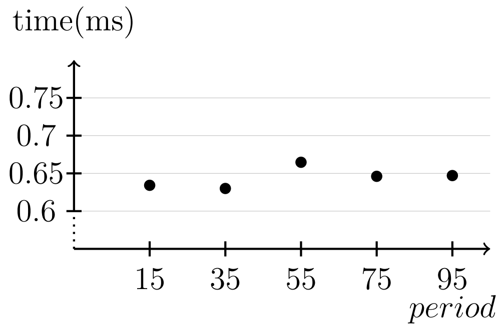

Figure 5.24: Average run times of the *PatternConstraint* with the parameters $offset = [1, 3, 5]$ and $jitter = 1$

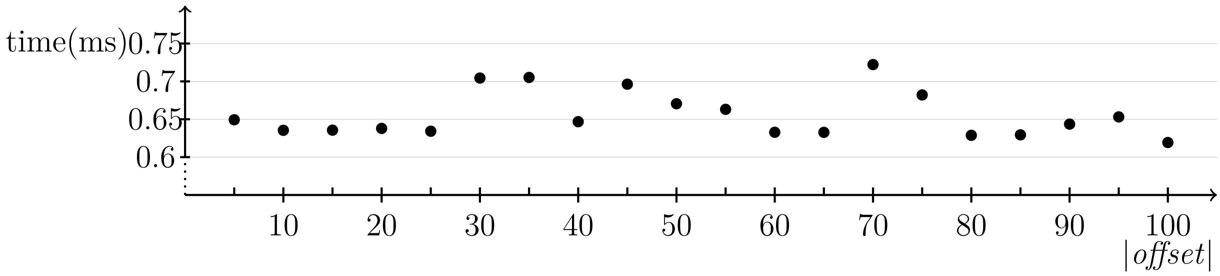

Figure 5.25: Average run times of the *PatternConstraint* with the parameters $period = 200$ and $jitter = 0$

times was nearly constant, beside some measurement deviations. This behaviour was expected by the analysis in the previous section.

ArbitraryConstraint

Similar to the previous constraint, multiple runs were done for the run time measurement. First with small lengths of the *minimum* and *maximum* parameter and changing values for the values inside of these parameters, and then with a length of the *minimum* and *maximum* parameter of 1 to 100. Figure 5.26 and 5.27 are showing some of the results with short *minimum* and *maximum* parameters. It can be seen, that the results with the same length of these parameters are nearly constant, but the traces with a *minimum* length of 3 took slightly more time. Figure 5.28 shows the average run times in dependency of the length of the *minimum* parameter. The graphic shows a nearly linear growth of the run time, but based on the analysis a growth by $|maximum|^2$ was expected.

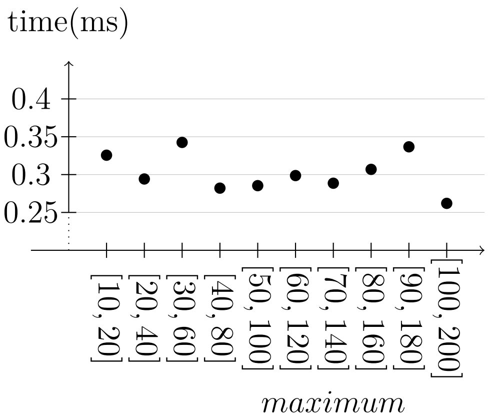

Figure 5.26: Average run times of the *ArbitraryConstraint* with the parameter *minimum* $= [10, 20]$

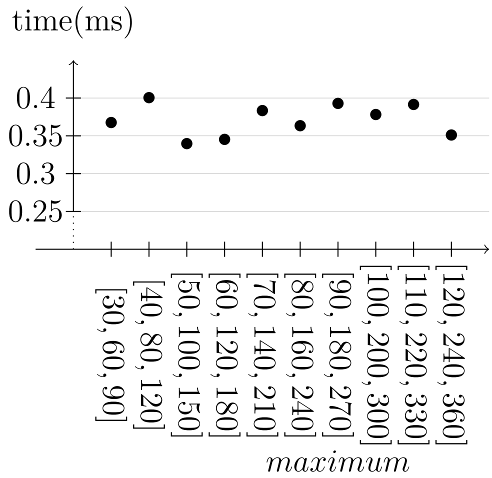

Figure 5.27: Average run times of the *ArbitraryConstraint* with the parameter *minimum* $= [30, 60, 90]$

The *ArbitraryConstraint* is defined as

$$ArbitraryConstraint(events, minimum, maximum) \Leftrightarrow$$
$$\forall i : RepeatConstraint(events, lower = minimum_i, upper = maximum_i, span = i)$$

The runtime of the *RepeatConstraint* is linear dependent on the *span* parameter, so the run time of the *RepeatConstraint* is expected to grow by $\sum_{i=1}^{|maximum|} i = \frac{|maximum|^2 + |maximum|}{2}$.
Earlier in this section, we've shown that the run time of the *RepeatConstraint* is continuously growing, but fairly slow. This explains the nearly linear run time, because the rise in the runtime of the *RepeatConstraint* is so small that the linear part of $\frac{|maximum|^2 + |maximum|}{2}$ is predominating.

BurstConstraint

Figure 5.29 shows the average run time per input timestamp with increasing a number of occurrences per burst. Similar to the *RepeatConstraint*, over which the *BurstConstraint* is defined and implemented, are small increase in the run times can be seen by increasing the number of occurrences. This increase is smaller than the fluctuations between the individual measurements, but the trend can be seen in all of the results. The linear growth was expected by the analysis in the previous section.

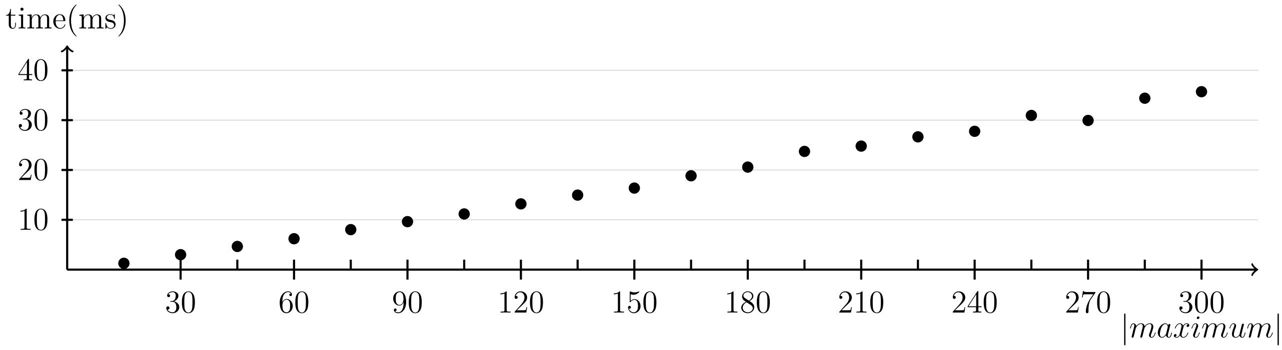

Figure 5.28: Average run times of the *ArbitraryConstraint* with $|minimum| = 1..100$

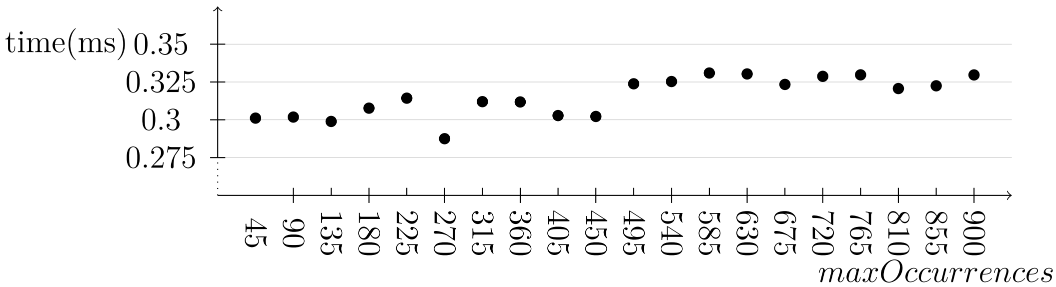

Figure 5.29: Average run times of the *BurstConstraint* with increasing *occurrences* per burst and a length of 2000

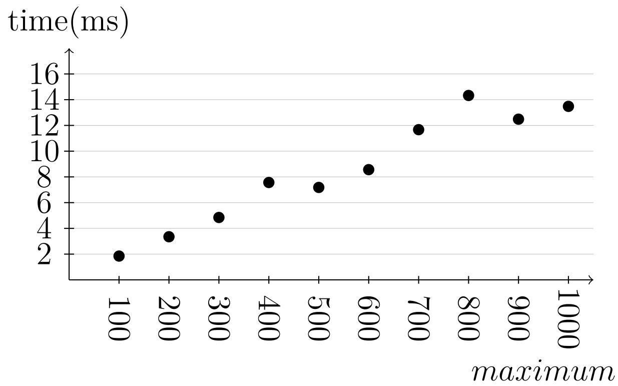

Figure 5.30: Average run times of the *Reaction-Constraint* with a distance between subsequent *stimulus* events of 1 (worst case) and variable *maximum*

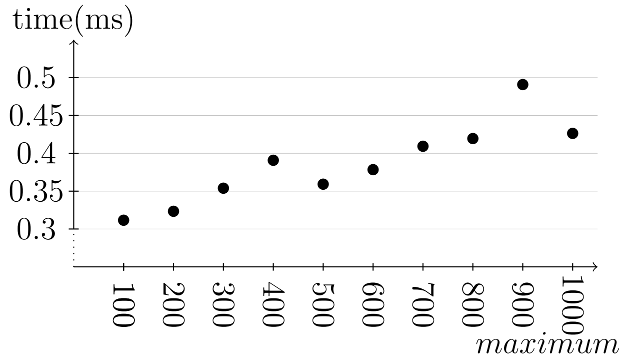

Figure 5.31: Average run times of the *Reaction-Constraint* with a distance between subsequent *stimulus* events of 128 and variable *maximum*

ReactionConstraint

The runtime evaluation of the *ReactionConstraint* was done on traces with the parameters $minimum \in \{100, 200, ..., 1000\}$ and $maximum = minimum$, while the distances between subsequent *stimulus* event were in $\{1, 2, 4, 8, ..., 1024\}$, so that $minimum$, $\lceil \frac{minimum}{2} \rceil$, $\lceil \frac{minimum}{4} \rceil$, ..., $\lceil \frac{minimum}{1024} \rceil$ events must be stored and considered at every event in the monitor.

Figure 5.30 and 5.31 are showing the average run times of the monitor with increasing *minimum* and *maximum* parameters, but fixed distance between subsequent *stimulus* events. The first figure shows the run times with *stimulus* distances of 1, which is the worst case, because *maximum* events must be stored and considered for the correctness decision of the monitor. Like expected by the analysis, the run time is increasing linear with larger *maximum* values. This behaviour can also be seen in the second figure, where the distance between the events is 128, but the run times are much smaller here. This is, because between 1 ($\lceil \frac{100}{128} \rceil$) and 8 ($\lceil \frac{1000}{128} \rceil$) were considered in each timestamp with events, not between 100 and 1000 in the previous case.

AgeConstraint

The run time of the *AgeConstraint* monitor were measured on traces with the same parameters as the previous constraint. Figure 5.32 shows the run times with event distances of 1, which is the worst case in terms of monitoring, in dependency of the *maximum* parameter. With increasing *maximum* values, the average run time

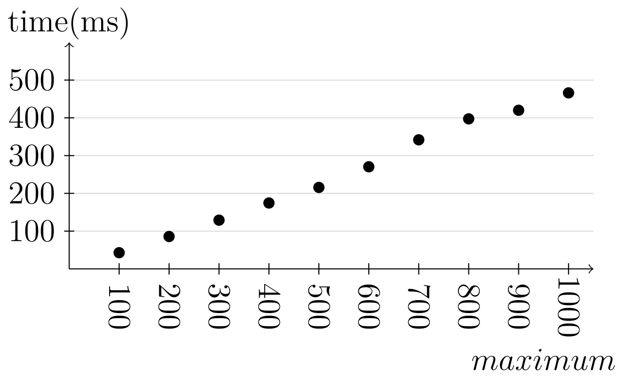

Figure 5.32: Average run times of the *AgeConstraint* with a distance between subsequent *stimulus* events of 1 (worst case) and variable *maximum*

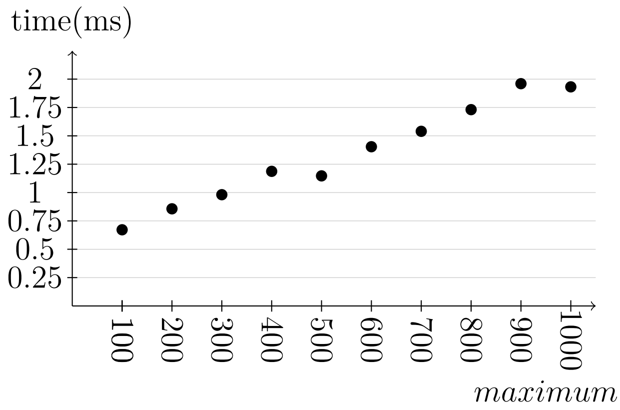

Figure 5.33: Average run times of the *AgeConstraint* with a distance between subsequent *stimulus* events of 128 and variable *maximum*

grew linear, like expected in the analysis. The average run times with the same *maximum* values and a distance between subsequent *stimulus* events is shown in figure 5.33. The run time is growing nearly linear again. Deviations can be seen at $maximum = 500$ and $maximum = 1000$. Because $\lceil \frac{400}{128} \rceil = \lceil \frac{500}{128} \rceil$ and $\lceil \frac{900}{128} \rceil = \lceil \frac{1000}{128} \rceil$, therefore the number of events, which must be stored, and considered in each input timestamp are equal in these traces. Therefore, the run time is not increasing at these parameter values.

OutputSynchronizationConstraint

The traces for the evaluation of the *OutputSynchronizationConstraint* were generated with 2,3, 4 and 5 *stimulus* streams, *tolerance* values of 10 to 25 in steps of 3 and a distance between synchronization clusters of 2, 4, 8, 16 or 32. In a second run, the run times for traces with 2, 22, 42, ..., 202 *response* streams were measured. Figure 5.34 shows the run time with a cluster distance of 2 and 4 *response* streams. Like expected, the growth of the run time is linear with larger values for the *tolerance* parameter. Figure 5.35 the average run times with a fixed cluster distance of 2 and $tolerance = 2$. As expected, the run times are growing by the square of $|response|$.

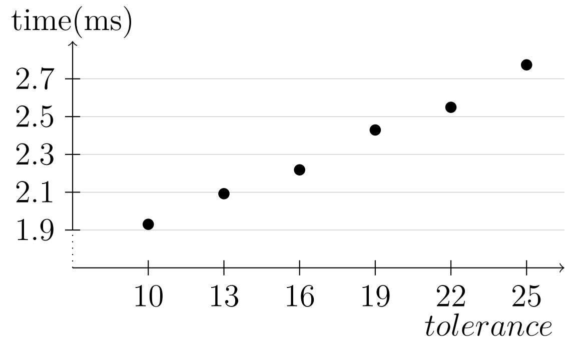

Figure 5.34: Average run times of the *OutputSynchronizationConstraint* with 4 *response* streams and a cluster distance of 2

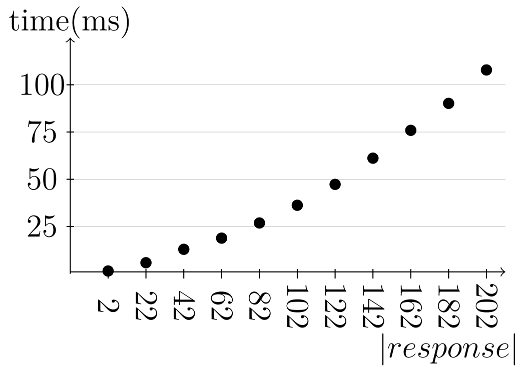

Figure 5.35: Average run times of the *OutputSynchronizationConstraint* with a cluster distance of 2 and *tolerance* = 10

InputSynchronizationConstraint

The traces for the evaluation of the *InputSynchronizationConstraint* were generated with 2,3, 4 and 5 *stimulus* streams, *tolerance* values of 10 to 25 in steps of 3 and a distance between synchronization clusters of 2, 4, 8, 16 or 32. Similar to the previous constraint, traces with up to 202 *stimulus* streams were tested in a second run. Figure 5.36 shows the run time of the monitor with the traces with three *stimulus* streams and a fixed cluster distance of 2. The run times are nearly constant, which was expected by the analysis of the source code. Figure 5.37 shows the average run time with a fixed cluster distance and *tolerance* and an increasing number of *stimulus* streams. Like expected, the run times increases by the square of the number of *stimulus* streams, but the increase is much larger than in the previous constraint.

EventChain

The run time of monitor for the correctness of event chains were also measured. The traces were generated with the same parameters as for the *ReactionConstraint*. Figure 5.38 and 5.39 are showing the results of this measurement, with fixed distances between subsequent *stimulus* events of 1 and 128 timestamps and distances between *stimulus* events and their associated *response* event of 100, 200, ..., 1000. The run times in both cases are nearly constant, but the run times are slightly larger in figure 5.38. The reason for this is, that the *stimulus* and *response* events occur in the same timestamps here, but not in figure 5.39.

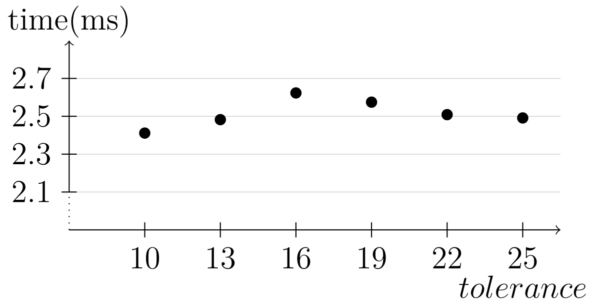

Figure 5.36: Average run times of the *InputSynchronizationConstraint* with 3 stimulus streams and a cluster distance of 2

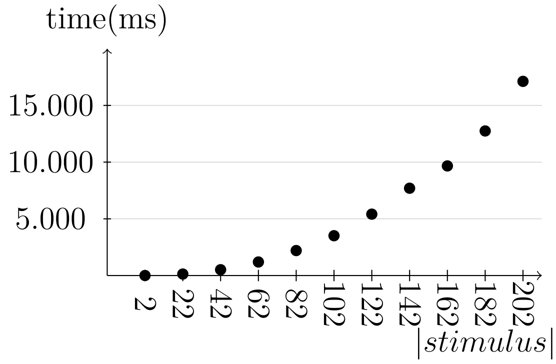

Figure 5.37: Average run times of the *InputSynchronizationConstraint* with a cluster distance of 2 and *tolerance* = 10

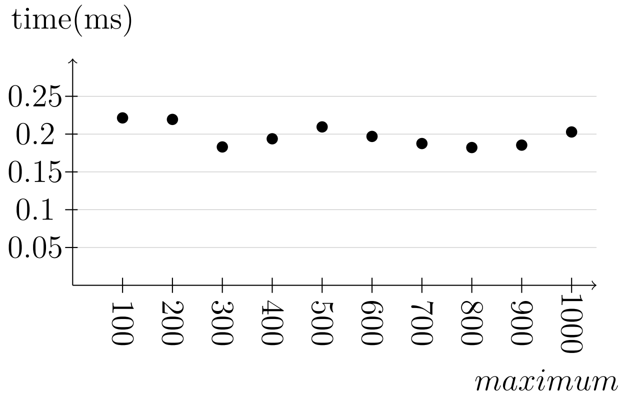

Figure 5.38: Average run times of the *EventChain* check with a distance between subsequent *stimulus* events of 1 and variable *maximum*(*ReactionConstraint* parameter)

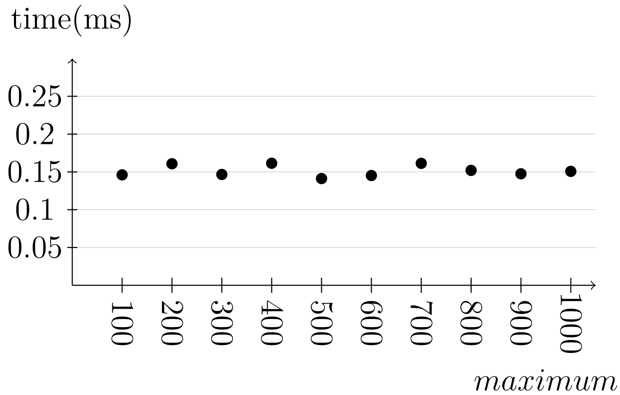

Figure 5.39: Average run times of the *EventChain* check with a distance between subsequent *stimulus* events of 128 and variable *maximum*(*ReactionConstraint* parameter)

6 Summary and Outlook

In this chapter, a summary of the presented work is given. Additionally, some ideas of future work will be presented.

6.1 Summary

In this thesis it has been shown that implementing a monitor for the AUTOSAR Timing Constraint is problematic due to informal definitions. Because of this, the timing constraints defined in the Timing Augmented Description Language v2 (TADL2) were considered for the implementation of a monitoring tool.
After the relations between the AUTOSAR Timing Extensions and the timing constraints defined in TADL2 were explained, the term *simple monitorable*, which ensures that a property on an possibly infinite trace can be monitored with finite resourcesm were introduced and extended by the possibility of inserting new timestamps. This term was applied to the TADL2 timing constraints, with the result, that eight of the constraints are simple monitorable with or without delay, five constraints require infinite memory resources on infinite traces in worst case scenarios and 4 constraints require infinite memory resources on nearly all infinite traces. On one constraint the term *simple monitorable* is not applicable, because it is not defined on event streams.
After the theoretical part, an implementation for all of the TADL2 timing constraints in TeSSLa were given, except for the *ComparionConstraints*, which functionality already was implemented in TeSSLa. The worst case runtime per timestamp with input events and the memory usage were analyzed. At the end, the run time of the implementations were measured on large generated traces.

6.2 Future Work

For a real world use of the monitors, more work on this topic is required. It is possible to map TeSSLa-specifications reconfigurable hardware like FPGAs [DDG+18]. Because of memory and recursion restrictions, this is not possible for all specifications . The possibility to map the implementations on reconfigurable hardware

would increase the performance and opens the gate for real world usage in embedded systems in the automotive industry.
Some constraints were classified as *not simple monitorable*, but could be restricted, so that they are *simple monitorable*. For example the *Reaction*- and *AgeConstraint* are classified as *always not simple monitorable*, because they need to store every occurring color and therefore have a continuously growing memory usage. If all events have a minimal distance and the color attribute is defined as integers, which occur strictly ordered, a monitor with a fixed upper limit in memory resources could be build. Restrictions of this kind are possible to many of the timing constraints which are classified as *not simple monitorable*, but it must be ensured that the monitored system also fulfills the restrictions.

List of Figures

List of Tables

Quelltextverzeichnis

Abbreviations

DFST	Deterministic Finite State Transducer
TDFST	Timed Deterministic Finite State Transducer

Bibliography

[ABLS05] ARAFAT, Oliver ; BAUER, Andreas ; LEUCKER, Martin ; SCHALLHART, Christian: Runtime verification revisited. (2005), 10/2005

[AD94] ALUR, Rajeev ; DILL, David L.: A theory of timed automata. In: *Theoretical Computer Science* 126 (1994), Nr. 2, 183 - 235. `http://dx.doi.org/https://doi.org/10.1016/0304-3975(94)90010-8`. – DOI https://doi.org/10.1016/0304–3975(94)90010–8. – ISSN 0304–3975

[AUT] *Current Partners - AUTOSAR*. `https://www.autosar.org/about/current-partners/`, . – Accessed: 2020-11-13

[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

[BLS07] BAUER, Andreas ; LEUCKER, Martin ; SCHALLHART, Christian: The Good, the Bad, and the Ugly, But How Ugly Is Ugly? In: SOKOLSKY, Oleg (Hrsg.) ; TAŞIRAN, Serdar (Hrsg.): *Runtime Verification*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007. – ISBN 978–3–540–77395–5, S. 126–138

[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

[DDG+18] Decker, N. ; Dreyer, B. ; Gottschling, P. ; Hochberger, C. ; Lange, A. ; Leucker, M. ; Scheffel, T. ; Wegener, S. ; Weiss, A.: Online analysis of debug trace data for embedded systems. In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, S. 851–856

[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)

[LPZ85] Lichtenstein, Orna ; Pnueli, Amir ; Zuck, Lenore: The glory of the past. In: Parikh, Rohit (Hrsg.): *Logics of Programs*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1985. – ISBN 978–3–540–39527–0, S. 196–218

[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