Model-checking in the Foundations of Algorithmic Law and the Case of Regulation 561

Moritz Müller[1] and Joost J. Joosten[2]

[1]Passau University
[2]University of Barcelona

September 6, 2023

Abstract

We discuss model-checking problems as formal models of algorithmic law. Specifically, we ask for an algorithmically tractable general purpose model-checking problem that naturally models the European transport Regulation 561 ([50]), and discuss the reaches and limits of a version of discrete time stopwatch automata.

1 Model-checking and algorithmic law

Should legal practice be enforced by software? This is a subtle and difficult question that we shall not address in this paper. The matter of fact is, our society has evolved in such a way that legal practice *is* currently enforced by software in various situations.

In this paper we will study how model checking can help in such situations. Our leading case study will be the European Traffic Regulation 561 [50]. Notwithstanding, our general approach aims at a more general panorama.

So, we do not enter the discussion whether software should be used to enforce the law. Rather we make the point that if it is so used, it better be done in a right fashion as to increase transparency, fairness and as to eradicate possible errors. We hope that the current paper contributes to these aims.

1.1 Computational problems in algorithmic law

The European transport Regulation 561 [50] concerns activities of truck drivers as recorded by tachographs. A tachograph recording determines for each time unit the activity of the driver which can be *driving*, *resting* or *doing other work*. Regulation 561 is a complex set of articles that limits driving and work time by prescribing various types of rest periods. The regulation prescribes that the time units are minutes, so a tachograph recording of 2 months determines a sequence of activities of length 87840.

It is clear that the legality of such a recording can only be judged with the help of an algorithm.

We think that algorithmic law-enforcement can only make sense in case there are no discretionary powers to be implemented. But our meaning is irrelevant here since legal practice *uses* automated law enforcement and we will consider such an example. In this paper we shall see that formalising a law typically leads to choices to be made to disambiguate or even repair the written law. A programmer is not entitled nor knowledgeable to do so, hence we observe that such a disambiguation stage should involve both legal and computer science scholars.

By the application of a law to a case we mean the decision whether the case is legal according to that law or not. By an *algorithmic law* we mean a law whose application to a case is (necessarily or intended/suggested to be) executed by an algorithm. Instead of designing one algorithm per law we are interested in *general purpose* algorithms: these take as input both a case from a set of cases of interest, and a law from a set of laws of interest, and decide whether the given case is legal according to the given law or not. In order to present cases and laws of interest as inputs to an algorithm, both have to be suitably *formalized*.

For Regulation 561, we make the choice that a case is naturally formalized as a word over the alphabet $\Sigma := \{d, r, w\}$: e.g., the word $dddwrr \in \Sigma^6$ is the activity sequence consisting of 3 minutes driving, followed by 1 minute other work, followed by 2 minutes resting.[1] With this choice, cases whose legality have to be decided by the law will be words of letters like the string $dddwrr$ above. In mathematical logic it is a straightforward and widely applied technique to conceive a word as a mathematical structure K (see e.g. [36, Example 4.11]). So, generally, we formalize a set of cases by a class of finite structures $\mathcal{K}$.

In this setting, a generic formalization of a law is given by translating the law to a sentence φ of a formal language, in the context of a logic L. Thus, the sentence expressing the law will impose requirements on the cases and that a particular case K is legal according to the law φ then formally means that $K \models \varphi$, i.e., K satisfies φ, where the structure K formalises our case from the allowed class of structures $\mathcal{K}$. We arrive at what is the central computational problem of algorithmic law:

Model-checking The *model-checking problem (for L over $\mathcal{K}$)* is a formal model for a family of algorithmic laws where laws are formalized by sentences of L and cases are formalized by structures in $\mathcal{K}$.

$\mathrm{MC}(\mathcal{K}, L)$	
Input:	$K \in \mathcal{K}$ and $\varphi \in L$.
Problem:	$K \models \varphi$?

A *model-checker (for L over $\mathcal{K}$)* is an algorithm deciding $\mathrm{MC}(\mathcal{K}, L)$. This is a general purpose algorithm as asked for above.

[1] Some tachograph readers will work with other formats like *activity-change lists* which is a list of moments in time where the driver's activity changes. This could beg for another formalisation and we leave this discussion outside the scope of this paper.

We consider two more computational problems associated to algorithmic law.

Consistency-checking A minimal requirement for law design is that it should be possible to comply with the law. This sounds like a void academic requirement but inconsistent regulations are abundant (see e.g. [33], [24], [31] or Remark 14).

For laws governing activity sequences consisteny means that there should be at least one such sequence (or equivalently, structure K) that is legal according to the law. In case that we can indeed find a particular case K that satisfies the law, we say that the law is *consistent*.

A related question of interest is whether a certain type of behaviour can be legal. This is tantamount to ask whether the artificial law augmented by demanding the type of behaviour is consistent.

This is formally modeled by the *consistency problem (for L over $\mathcal{K}$)*:

$\textsc{Con}(\mathcal{K}, L)$	
Input:	$\varphi \in L$.
Problem:	does there exist some $K \in \mathcal{K}$ such that $K \vDash \varphi$?

Scheduling Assume a truck driver has to schedule next week's driving, working and resting and is interested to drive as long as possible. A week has 10080 minutes, so the driver faces the computational optimization problem to compute a length 10080 extension of the word given by the current tachograph recording that is legal according to Regulation 561 and that maximizes driving time.

Consider laws governing activity sequences, that is, $\mathcal{K}$ is the (set of structures corresponding to the) set of finite words Σ^* over some alphabet Σ. For a word $w = a_0 \cdots a_{n-1} \in \Sigma^n$ (the a_i are letters that represent the corresponding activities) and a letter $a \in \Sigma$, let $\#_a(w)$ denote the number of times the letter a appears in w, i.e.,

$$\#_a(w) := |\{i < n \mid a_i = a\}|.$$

The *scheduling problem (for L over $\mathcal{K} = \Sigma^*$)* is:

$\textsc{Scheduling}(\mathcal{K}, L)$	
Input:	$\varphi \in L$, $w \in \Sigma^*$, $a \in \Sigma$ and $n \in \mathbb{N}$.
Problem:	if there is no $v \in \Sigma^n$ such that $wv \vDash \varphi$, then output "illegal"; otherwise output some $\overline{v} \in \Sigma^n$ such that $\#_a(w\overline{v}) = \max\{\#_a(wv) \mid v \in \Sigma^n, wv \vDash \varphi\}$.

1.2 Model-checking as a formal model

There is a vast amount of research concerning model-checking problems $\mathrm{MC}(\mathcal{K}, L)$. The two main interpretational perspectives stem from *database theory* and from *system verification*. In database theory [47], $\mathcal{K}$ is viewed as a set of databases, and L a set of Boolean queries. In system verification [7], $\mathcal{K}$ is as a set of transition systems or certain automata that formalize concurrent systems or parallel programs, and L formalizes correctness specifications of the system, that is, properties all executions

of the system should have. We add a third interpretational perspective on model-checking problems as formal models for families of algorithmic laws. We highlight three conflicting requirements on such a formal model.

Tractability requirement The first and foremost constraint for a model $\mathrm{MC}(\mathcal{K}, L)$ of a family of algorithmic laws is its computational complexity. For the existence of a practically useful general purpose model-checker the problem $\mathrm{MC}(\mathcal{K}, L)$ should be *tractable*. We argue that the notion of tractability here cannot just mean PTIME, a more fine-grained complexity analysis of $\mathrm{MC}(\mathcal{K}, L)$ is required.

Classical computational complexity theory tells us that already extremely simple pairs $(\mathcal{K}, L)$ have intractable model-checking problems. An important example from database theory is that $\mathrm{MC}(\mathcal{K}, L)$ is NP-complete for L the set of conjunctive queries and $\mathcal{K}$ the set of graphs (or the single binary word 01) [18]. An important example [52] from system verification is that $\mathrm{MC}(\mathcal{K}, L)$ is PSPACE-complete for L equal to linear time temporal logic LTL and $\mathcal{K}$ the class of finite automata [54].

However, the mentioned PSPACE-completeness result is largely irrelevant because this model-checking problem is *fixed-parameter tractable (fpt)*, that is, it is decidable in time $f(k) \cdot n^{O(1)}$ for some computable function $f : \mathbb{N} \to \mathbb{N}$ where n is the total input size and $k := \|\varphi\|$ the size of (a reasonable binary encoding of) the input LTL formula φ. In fact, we have *parameter dependence* $f(k) \leqslant 2^{O(k)}$. Informally speaking, we are mainly interested in inputs with $k \ll n$, so this can be considered tractable. In other words, the computational hardness relies on uninteresting inputs with relatively large k. In contrast, model-checking conjunctive queries over graphs is likely not fixed-parameter tractable: this is equivalent to $\mathsf{FPT} \neq \mathsf{W}[1]$, the central hardness hypothesis of parameterized complexity theory .[2] Most people indeed somehow believe that $\mathsf{FPT} \neq \mathsf{W}[1]$ in a sense similar to that most people believe that $\mathsf{PTIME} \neq \mathsf{NP}$.

The focus on inputs with $k \ll n$ is common in model-checking and it is an often repeated point that a reasonable complexity analysis must take this asymmetry of the input into account; [49] is an early reference addressing both perspectives from database theory and system verification. The theoretical framework for such a fine-grained complexity analysis is parameterized complexity theory [36, 27, 28] whose central tractability notion is fixed-parameter tractability.[3]

To sum up, judging he tractability of $\mathrm{MC}(\mathcal{K}, L)$ should be based on a fine-grained complexity analysis that measures the computational complexity with respect to various input *aspects* $n, k, \ldots$.[4] The quality of the model $\mathrm{MC}(\mathcal{K}, L)$ depends on the "right" identification of relevant aspects in its complexity analysis.

Expressivity requirement Recall thate we ask for a *general purpose* model-checker that solves a model-checking problem $\mathrm{MC}(\mathcal{K}, L)$ modeling a family $\mathcal{L}$ of algo-

[2] Grohe [39] (refined in [21, 22]) gives a quite complete understanding of which conjunctive queries are tractable.

[3] This paper does not require familiarity with parameterized complexity theory. Only Section 9.3 requires some results of this theory and will recall what is needed.

[4] Formally, an *aspect* could be defined as a *parameterization*, possibly viewed as a *size measure* as in [36, p.418f]. However, we don't need a definition and use the term informally.

rithmic laws instead of single-purpose model-checkers deciding $\mathrm{MC}(\mathcal{K}, \{\varphi\})$, one per algorithmic law φ. From a theoretical perspective we expect insight on which laws can possibly be algorithmic.

From a practical perspective, this avoids the costly production of many algorithms, their updates following law reforms and their validation for legal use. It is thus desirable to find tractable $\mathrm{MC}(\mathcal{K}, L)$ for as rich as possible classes $\mathcal{K}$ and L. In particular, for laws governing sequences of activities (i.e., $\mathcal{K} = \Sigma^*$) we ask for an as expressive as possible logic L. Of course, this is in tension with the tractability requirement.

Naturality requirement From an algorithmic perspective it is not only the expressivity of L that matters, but also its *succinctness*. Typically, model-checking complexity grows fast with the size of the sentence φ formalizing the law, so logics allowing for shorter formalizations are preferable. E.g., it is well-known that the expressive power of LTL is not increased when adding past modalities but their use can lead to exponentially shorter sentences.

Crucially, the complexity of model-checking (over finite automata) is not substantially increased. Moving to a more succinct logic is not necessarily an improvement. E.g. further adding a now-modality again increases succinctness exponentially but apparently also the model-checking complexity [44].

Furthermore, it is one thing to model a law application by a model-checking instance (K, φ) any old how and another to do so by somehow typical members of $\mathcal{K}$ and L. E.g., in case the formalization of actual laws uses only special artificial members of $\mathcal{K}$ (*semantic overkill*) or L (*syntactic overkill*), one would want to trade the richness of $\mathcal{K}$ and L for a faster model-checker.

Very long or contrived formalizations of laws are also prohibitive for legal practice which requires the law to be readable and understandable by humans. This is vital also for the validation of their formalization, i.e., their translation from the typically ambiguous natural language into a formal language able to be algorithmically processed. Without attempting a definition of this vague term, we thus informally require that the (formalization given by the) model $\mathrm{MC}(\mathcal{K}, L)$ must be *natural*.

Other requirements We focus on the above three requirements but, of course, there are more whose discussion we omit. An important one is trust in the output of model-checkers. For example, even if the mathematical model is correct, its translation to an executable computer program may be flawed. This issue could call for formally verified implementations of the software and we refer to [1] for an example.

Additionally, algorithmic outputs should be transparent and explainable to be used in legal practice and it is unclear what this exactly means. Further requirements on the model might come from ethical or political considerations - e.g., the required transparency can be in conflict with intellectual property rights and there can be more general issues concerning the involvement of the private sector in law execution.

Often, automated law enforcement has been considered to violate the legal *principle of contestability*: citizens can hardly contest to a legal judgement if it is a computer program that made that judgement. A human expert and a computer program may be equally unintelligible for a citizen. However, there is one important fundamental

difference and this is that the citizen can enter in a *dialogue* with the human officer where such a dialogue is generally simply precluded from computerised interactions. The model-checking approach does however naturally allow for such a dialogue. Recall that a case $\mathcal{M}$ satisfying a law λ would be casted in the model-checking framework as $\mathcal{M} \vDash \lambda$. Let π be a property of a law λ. For example that the law acts equally on all citizens disregarding if they hold one or two passports[5]. Asking if the law λ satisfies the property π can be casted as $\vDash \lambda \to \pi$ in the model checking framework.

We focus on laws governing temporal sequences of activities, that is, laws concerning cases that can readily be formalized by words over some finite alphabet Σ, i.e., $\mathcal{K} = \Sigma^*$. This paper is about the quest for a logic/language L such that $\mathrm{MC}(\mathcal{K}, L)$ is a good model for such laws. To judge expressivity and naturality we use European Regulation 561 [50] as a test case, that is, we want L to naturally formalize Regulation 561. Given the complexity of this regulation, this is an ambitious goal and we expect success to result in a model that encompasses a broad family of laws concerning sequences of activities.

1.3 Contributions and outline of this paper

In our paper we suggest (a version of) discrete time *stopwatch automata* SWA as an answer to our central question, that is, we propose $\mathrm{MC}(\Sigma^*, \mathsf{SWA})$ as a model for algorithmic laws concerning sequences of activities.[6]

In the next section, Section 2 we first revisit basic notions for classical theory for model checking. We also mention the main ingredients that lead to our particular choice of SWAs in this paper. A fuller theoretical motivational discussion of our choice is postponed to Section 10 after we have seen SWAs in action and have proven various of its computational properties. In Section 3 we give our basic definition on Stopwatch automata that we shall work with in this paper. Naturally, we also define the corresponding notion of computations of SWAs. Since we will have to quantify the computational and runtime behavior of our SWAs, we also say some words on concrete formalised SWAs and their sizes.

After having defined our notion of SWA we test our informal criterion of naturality for this notion. We do so by showing in Sections 4 and 5 how the complex European trafic regulation 561 can naturally be formalised using our notion of SWAs.

From Section 6 on, we start exploring the theoretical and computational properties of SWAs. In particular, in Section 6 we observe that SWA has the same expressive power as MSO over finite words. Furthermore, we showcase the robustness of our definition by exhibiting a definitorial equivalent variation.

In Section 7 we show that the model-checking-complexity for the notion of SWAs that we defined is rather tame and likewise for consitency checking. This relative tameness is expressed by the main theorem from Section 7 where we prove the following

[5] Asking this to the Dutch Syri Risk Indication System could have revealed its bias before its devastating impact on various often vulnerable families in The Netherlands.

[6] In the notation of Section 1.1 we define $w \vDash \mathbb{A}$ for a finite word w and a stopwatch automaton $\mathbb{A}$ to mean that $\mathbb{A}$ accepts w.

upper bound on the complexity of $\mathrm{MC}(\Sigma^*, \mathsf{SWA})$:

Theorem 1. *There is an algorithm that given a stopwatch automaton $\mathbb{A}$ with size $\|\mathbb{A}\|$ and a word w with length $|w|$ decides whether $\mathbb{A}$ accepts w in time*

$$O\big(\|\mathbb{A}\|^2 \cdot t_{\mathbb{A}}^{c_{\mathbb{A}}} \cdot |w|\big).$$

Let us briefly comment on what the other symbols from this theorem denote. Our stopwatches have bounds on the possible largest value that they can register, and their bounds correspond to time constants mentioned in laws. We let $c_{\mathbb{A}}$ denote the number of stopwatches and $t_{\mathbb{A}}$ the largest stopwatch bound of a stopwatch automaton $\mathbb{A}$.

We stress that the aspect $t_{\mathbb{A}}$ does not appear in the exponent, so this overcomes a bottle-neck of various model-checkers designed in system verification which will be discussed in Section 10.2.

In Section 8 we prove in Theorem 31 that, surprisingly, scheduling is not much harder than model checking in our setting.

Section 9 discusses our model $\mathrm{MC}(\Sigma^*, \mathsf{SWA})$ following the criteria of Section 1.2, and gives a critical examination of the factor $t_{\mathbb{A}}^{c_{\mathbb{A}}}$ in the runtime of our model-checker. Intuitively, typical inputs have small $c_{\mathbb{A}}$ and large $t_{\mathbb{A}}$, and it would be desirable to replace this factor by, e.g., $2^{O(c_{\mathbb{A}})} \cdot t_{\mathbb{A}}^{O(1)}$. We show this is unlikely to be possible by relating a hypothetical such algorithm to an unproven assumption from computational complexity that most computer scientists believe to hold true: $\mathsf{FPT} \neq \mathsf{W}[1]$. Using this Complexity Theoretical assumption, our Theorem 38 implies:

Theorem 2. *Assume $\mathsf{FPT} \neq \mathsf{W}[1]$ and let $f : \mathbb{N} \to \mathbb{N}$ be a computable function. Then there does not exist an algorithm that given a stopwatch automaton $\mathbb{A}$ and a word w decides whether $\mathbb{A}$ accepts w in time*

$$\big(\|\mathbb{A}\| \cdot f(c_{\mathbb{A}}) \cdot t_{\mathbb{A}} \cdot |w|\big)^{O(1)}.$$

We present a proof of this theorem that does not require any particular knowledge on the mentioned classes and just work with a particular representative from them.

Finally, in Section 10 we motivate the choices we made to work with our definition of bounded stopwatch automata. This motivation should be placed in the context of the larger landscape of literature on model checking and temporal logics.

2 Finite Automata

As mentioned before, the imperative constraint to find the right notion of formalising laws that should be enforced in an automated fasion is the tractability of $\mathrm{MC}(\mathcal{K}, L)$. In secion 10 we survey[7] the relevant literature on model-checking and discuss shortcomings of known model-checkers. Thereby we motivate what the right input aspects are, i.e., those relevant to calibrate the computational complexity of $\mathrm{MC}(\mathcal{K}, L)$ and to judge its tractability.

[7] This survey mentions many logics and automata and we shall not assume familiarity with these concepts later on.

In this section we merely mention what our main model choices have been and indicate the considerations that played a role arriving to this definition. Before doing so, we first recall the basic definitions and properties of our starting point: non-deterministic finite automata.

2.1 Regulation 561 and Büchi's theorem

We recall Büchi's theorem and, to fix some notation, the definitions of regular languages and finite automata. To start, we use the standard notation to denote strings over a finite alphabet.

An *alphabet* Σ is a non-empty finite set of *letters*, $\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$ denotes the set of (finite) *words*. A word $w = a_0 \cdots a_{n-1} \in \Sigma^n$ (the a_i are letters) has *length* $|w| := n$.

Definition 3. A *(non-deterministic) finite automaton* $\mathbb{B}$ is given by a finite set of *states* Q, an alphabet Σ, sets of *initial* and *final* states $I, F \subseteq Q$, and a set $\Delta \subseteq Q \times \Sigma \times Q$ of *transitions*.

Computation of an automaton $\mathbb{B}$ are defined in terms of sequences of transitions as usual.

Definition 4. A *computation of* $\mathbb{B}$ *on* $w = a_0 \cdots a_{n-1} \in \Sigma^n$ is a sequence $q_0 \cdots q_n$ of states such that $(q_i, a_i, q_{i+1}) \in \Delta$ for every $i < n$. The computation is *initial* if $q_0 \in I$ and *accepting* if $q_n \in F$. The *language* $L(\mathbb{B})$ *of* $\mathbb{B}$ is the set of words $w \in \Sigma^*$ such that $\mathbb{B}$ *accepts* w, i.e., there exists an initial accepting computation of $\mathbb{B}$ on w.

A language (i.e., subset of words over Σ) is *regular* if it equals $L(\mathbb{B})$ for some finite automaton $\mathbb{B}$. We refer to [55] for a definition of Monadic second order definable (MSO-definable) languages and a proof of Theorem 5. This theorem basically says that regular languages are quite expressive.

Theorem 5 (Büchi)**.** *A language is* MSO*-definable on words if and only if it is regular.*

Although this theorem shows the strong expressive power of regular languages, the so-called Pumping Lemma can be seen as showing a limit to the expressive power: finite automata on very long computations will either terminate or enter a loop at some time and this loop is reflected in repetitions of patters inside words that are in the language.

Theorem 6 (Pumping Lemma)**.** *For any regular language* L *there is a constant* $c \in \mathbb{N}$ *such that every word* $w \in L$ *of length* $|w| \geqslant c$ *can be written as* xyz *for words* x, y, z *with* $|xy| \leqslant c$ *and* $|y| > 0$ *such that* $xy^n z \in L$ *for every* $n \in \mathbb{N}$.

Moreover, this constant c *can be taken to be the number of states of an automaton accepting* L.

Büchi's theorem can be extended to infinite words and trees using various types of automata – we refer to [30] for a monograph on the subject. The proof of Büchi's theorem is effective in that there is a computable function that computes for every MSO-sentence φ and automaton $\mathbb{B}_\varphi$ that accepts a word w if and only if $w \vDash \varphi$. It follows

that $\mathrm{MC}(\Sigma^*, \mathsf{MSO})$ is fixed-parameter tractable: given an input (w, φ), compute $\mathbb{B}_\varphi$ and check if $\mathbb{B}_\varphi$ accepts w. This takes time[8] $f(|\varphi|) \cdot |w|$ for some computable function $f : \mathbb{N} \to \mathbb{N}$. It also follows that $\textsc{Con}(\Sigma^*, \mathsf{MSO})$ is decidable because finite automata have *decidable emptiness*: there is an (even linear time) algorithm that, given a finite automaton $\mathbb{A}$, decides whether $L(\mathbb{A}) = \varnothing$.

MSO is a very expressive logic. In [24] it is argued that Regulation 561 can be formalized in MSO, and naturally so. Thus, in a sense $\mathrm{MC}(\Sigma^*, \mathsf{MSO})$ is tractable, expressive and natural, so a good answer to our central question. The starting point of this work was the question for a better model, namely improving its tractability. The problem with the running time $f(|\varphi|) \cdot |w|$ of Büchi's model-checker is that the parameter dependence $f(k)$ grows extremely fast: it is non-elementary in the sense that it cannot be bounded by $2^{2^{\cdot^{\cdot^{2^k}}}}$ for any fixed height tower of 2's.

2.2 Model checking tailored for law

The non-elementary parameter dependence in Büchi's Theorem is bad in terms of feasible model checkers. There are other logics that have better parameter dependences and *Linear Temporal Logic* LTL is one of them. However, it is known that the price to pay is that various properties that we do want to express in law become too long to write down and we gave precise examples and references later on.

To conclude, MSO gives the wrong model because it does not allow sufficiently fast model-checkers, and LTL is the wrong model because it is not sufficiently (expressive nor) succinct, hence not natural. It can be expected that, like Regulation 561, many algorithmic laws concerning sequences of activities state lower and upper bounds on the duration of certain activities or types of activities. The constants used to state these bounds are not necessarily small, and this is an important aspect to take into account when analyzing the model-checking complexity.

Various model checking for timed automata have the explicit time bounds t in the exponent. From the perspective of algorithmic laws, t is not typically small and runtimes exponential in $t = 56h = 3360\ min$ are thus prohibitive. Tamer runtimes with t moved out of the exponent have been found for a certain natural MITL-fragment $\mathsf{MITL}_{0,\infty}$ both over discrete and continuous time – see [41, 5].

Automata for temporal law should find a balanced and somewhat feasible way to speak about durations which seems to be a central notion in Regulation 561 and alike. Stopwatches are a right tool to speak of durations but they lead to undecidability. We choose to work therefore with *bounded* stopwatch automata which blocks the venom of undecidability and in our case arguably scales reasonably well with the parameters.

3 Stopwatch automata

Before giving our definition we informally describe the working of a *stopwatch automaton* SWA. A stopwatch automaton is an extension of a finite automaton whose

[8] This is not true for the empty word w. We trust the reader's common sense to interpret this and similar statements reasonably.

computations happen in discrete time: the automaton can stay for some amount of time in some state and then take an instantaneous transition to another state.

There are constraints on which transitions can be taken at a given point of time as follows. Time is recorded by a set of *stopwatches* X, every stopwatch $x \in X$ has a *bound* $\beta(x)$, a maximal time it can record. Every stopwatch is *active* or not in a given state.

During a run that stays in a given state for a certain amount of time, the value of the active stopwatches increases by this amount of time (up to their bounds) while the inactive stopwatches do not change their value. Transitions between states are labeled with a *guard* and an *action*. The guard is a condition on the values of the stopwatches that has to be satisfied for the transition to be taken, usually requiring upper or lower bounds on certain stopwatch values. The action modifies stopwatch values, for example, resets some of the stopwatches to value 0.

In classical automata like the one we presented in Subsection 2.1, typically it are the transitions between states that are labeled by *letters of the alphabet*. In our stopwatch automata instead it will be the states that are labeled by letters. A stopwatch automaton *accepts* a given word over an alphabet Σ if there exists a computation that *reads* the word, that is, it starts in a special state *start* and ends in a special state *accept*. During the computation, for example, staying in a state for 5 time units means reading 5 copies of the letter labelling the state.

3.1 Abstract stopwatch automata

We now give the definitions that have been anticipated by the informal description above.

Definition 7. An *abstract stopwatch automaton* is a tuple $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ where

- Q is a finite set of *states* containing the states *start* and *accept*;
- Σ is a finite *alphabet*;
- X is a finite set of *stopwatches*;
- $\lambda : Q \to \Sigma$;
- $\beta : X \to \mathbb{N}$ maps every stopwatch $x \in X$ to its *bound* $\beta(x) \in \mathbb{N}$;
- $\zeta : X \to P(Q)$ maps every stopwatch $x \in X$ to the set $\zeta(x) \subseteq Q$ of states where the stopwatch x is *active in*;
- $\Delta \subseteq Q \times \mathcal{G} \times \mathcal{A} \times Q$ is a set of *transitions*.

Here, $\mathcal{G}$ is the set of *abstract guards* (for $\mathbb{A}$), namely sets of assignments, and $\mathcal{A}$ is the set of *abstract actions* (for $\mathbb{A}$), namely functions from assignments to assignments. An *assignment* (for $\mathbb{A}$) is a function $\xi : X \to \mathbb{N}$ such that $\xi(x) \leqslant \beta(x)$ for all $x \in X$. To be precise, we should speak of a β-assignment but the β will always be clear from the context. We define the *bound of* $\mathbb{A}$ to be

$$B_{\mathbb{A}} := \prod_{x \in X} (\beta(x) + 1)$$

understanding that the empty product is 1 so that $\prod_{x \in \varnothing} (\beta(x) + 1) := 1$.

It is easy to see that the bound $B_{\mathbb{A}}$ of $\mathbb{A}$ is the cardinality of the set of assignments (for $\mathbb{A}$) and we can identify an assignment with a point in the product space $\prod_{x\in X}(\beta(x)+1)$. We say that a transition $(q,g,\alpha,q')\in\Delta$ is *from* q, and *to* q', and *has* abstract guard g and abstract action α. Computations of stopwatch automata are defined in terms of their corresponding *transition systems*.

Definition 8. Let $\mathbb{A}=(Q,\Sigma,X,\lambda,\beta,\zeta,\Delta)$ be an abstract stopwatch automaton. The *transition system* $TS(\mathbb{A})$ of $\mathbb{A}$ is given by a set of nodes and labeled edges: a *node (of* $TS(\mathbb{A})$*)* is a pair (q,ξ) of a state $q\in Q$ and an assignment ξ; a *labeled edge (of* $TS(\mathbb{A})$*)* is a triple $((q,\xi),t,(q',\xi'))$ for nodes $(q,\xi),(q',\xi')$ and $t\in\mathbb{N}$ such that **either**:

- $t=0$ and $(q,g,\alpha,q')\in\Delta$ for an abstract guard g and an abstract action α such that $\xi\in g$ and $\alpha(\xi)=\xi'$,

  or,

- $t>0$ and $q=q'$ and ξ' is the assignment given by

$$\xi'(x)=\begin{cases}\min\{\xi(x)+t,\beta(x)\} & \text{if } q\in\zeta(x),\\ \xi(x) & \text{else.}\end{cases}$$

We will speak of *instantaneous transitions* and *duration transitions* respectively. For $t\in\mathbb{N}$ we let $\xrightarrow{t}$ be the binary relation that contains those pairs $((q,\xi),(q',\xi'))$ of nodes such that $((q,\xi),t,(q',\xi'))$ is a labeled edge. As announced before, computations of stopwatch automata are defined through the corresponding transition systems.

Definition 9. A (finite) *computation of* $\mathbb{A}$ is a finite walk in $TS(\mathbb{A})$, i.e., for some $\ell\in\mathbb{N}$ a sequence

$$\Big(\big((q_i,\xi_i),t_i,(q_{i+1},\xi_{i+1})\big)\Big)_{i<\ell}$$

of directed edges of $TS(\mathbb{A})$ such that $q_i\neq \mathit{accept}$ for all $i<\ell$; we write this as

$$(q_0,\xi_0)\xrightarrow{t_0}(q_1,\xi_1)\xrightarrow{t_1}(q_2,\xi_2)\xrightarrow{t_2}\cdots\xrightarrow{t_{\ell-1}}(q_\ell,\xi_\ell).$$

In this case, we say that the computation is *from* (q_0,ξ_0) and *to* (q_ℓ,ξ_ℓ); it is *initial* if ξ_0 is constantly 0 and $q_0=\mathit{start}$; it is *accepting* if $q_\ell=\mathit{accept}$. The computation *reads* the word

$$\lambda(q_0)^{t_0}\lambda(q_1)^{t_1}\cdots\lambda(q_{\ell-1})^{t_{\ell-1}}.$$

We understand that σ^0 denotes the empty string for every letter σ in the alphabet Σ and juxtaposition of strings corresponds to concatenation. Through computations, we define strings and languages accepted by a Stopwatch automaton.

Definition 10. The automaton $\mathbb{A}$ *accepts* $w\in\Sigma^*$ if there is an initial accepting computation of $\mathbb{A}$ that reads w. The set of these words is the *language* $L(\mathbb{A})$ *of* $\mathbb{A}$.

The requirement $q_i \neq \mathit{accept}$ for all $i < \ell$ in the definition of computations[9] means that we interpret *accept* as a halting state; it implies that the label $\lambda(\mathit{accept})$ as well as transitions from *accept* are irrelevant. In our applications, the bound of a stopwatch x typically exceeds the maximal guard value occurring for x by one. As such, we can distinguish between all clock values that went over this maximal guard and the ones not exceeding it. Furthermore, we observe that a duration transition can typically be split into smaller consecutive duration transitions without altering the word that is being accepted.

Remark 11. Stopwatch automata are straightforwardly explained for continuous time $\mathbb{R}_{\geqslant 0}$ where they read timed words, and bounds $\beta(x) = \infty$. Stopwatch automata according to [26, 42] are such automata where guards are Boolean combinations of $x \geqslant c$ (for $x \in X$ and $c \in \mathbb{N}$), and actions are resets (to 0 of some stopwatches). The emptiness problem for those automata is undecidable [42].

So-called *timed automata* additionally require stopwatches to be active in all states, and have decidable emptiness [4]. The model allowing $x \geqslant c + y$ (for $x, y \in X$ and $c \in \mathbb{N}$) in guards still has decidable emptiness and is exponentially more succinct than guards with just Boolean combinations of $x \geqslant c$ ([14]). Allowing more actions is subtle, e.g., emptiness becomes undecidable when $x := x \dot{-} 1$ or when $x := 2x$ is allowed; see [16] for a detailed study.

3.2 Specific stopwatch automata

To consider an abstract stopwatch automata as an input to an algorithm, we must agree on how to specify the guards and actions, i.e., properties and functions on assignments. This is a somewhat annoying issue because on the one hand our upper bounds on the model-checking complexity turn out to be robust with respect to the choice of this specification in the sense that they scale well with the complexity of computing guards and actions, so a very general definition is affordable. On the other hand, for natural stopwatch automata including the one we are going to present for the European Traffic Regulation 561, we expect guards and actions to be simple properties and functions.

As mentioned, typically guards mainly compare certain stopwatch values with constants or other values, and actions do simple re-assignments of values like setting some values to 0.

Hence our choice on how to specify guards and actions is somewhat arbitrary. To stress the robustness part, we use a general model of computation: Boolean circuits . In natural automata, we expect these circuits to be small.

An assignment determines for each stopwatch $x \in X$ its bounded value and as such can be specified by

$$b_{\mathbb{A}} := \sum_{x \in X} \lceil \log(\beta(x) + 1) \rceil$$

many bits. We think of the collection of $b_{\mathbb{A}}$ bits as being composed of blocks, with a block of $\lceil \log(\beta(x) + 1) \rceil$ bits corresponding to the binary representation of the value

[9] As additional motivation for this choice we mention that allowing computations to pass more often through *accept* will have undesired side-effects: any accepted word wa ending on some letter a would lead to any word $w(a)^n$ to be also accepted for any $n - 1 \in \mathbb{N}$.

of stopwatch $x \in X$ under the assignment. Below, in our definitions of specific guards and actions we should be careful that the amount of $b_{\mathbb{A}}$ bits to represent assignments may yield in values exceeding the specified clock bounds $\beta(x)$ for clock x.

A *specific guard* is a Boolean circuit with one output gate and $b_{\mathbb{A}}$ many input gates. The output gate flags whether or not a particular assignment satisfies the guard. The fact that the Boolean circuit may also be defined on values exceeding bounds is irrelevant as long as it behaves properly on the bounded values. Clearly, a specific guard determines an abstract guard in the obvious way.

A *specific action* is a Boolean circuit with $b_{\mathbb{A}}$ many output gates and $b_{\mathbb{A}}$ many input gates. On input an assignment, for each clock $x \in X$, it computes the binary representation of a value $v_x \in \mathbb{N}$ in the block of $\lceil \log(\beta(x)+1) \rceil$ output gates corresponding to x. Again, the fact that the Boolean circuit may also be defined on values exceeding bounds is irrelevant as long as it behaves properly on the bounded values. Furthermore, we agree that the assignment computed by the circuit maps x to $\min\{v_x, \beta(x)\}$ thereby mapping assignments to assignments. A specific action determines an abstract one in the obvious way.

A *specific stopwatch automaton* is defined like an abstract one but with specific guards and actions replacing abstract ones. A specific stopwatch automaton determines an abstract one taking the abstract guards and actions as those determined by the specific ones. Computations of specific stopwatch automata and the language they accept are defined as those of the corresponding abstract one.

We shall only be concerned with specific stopwatch automata and shall mostly omit the qualification 'specific'. But once we have specific stopwatch automata, we can speak about their size and the size of their representation/coding. The size of the representation/coding of an automaton $\mathbb{A}$ shall be denoted by $\|\mathbb{A}\|$.

It is with specific automata that we can compare size of automata and estimate time durations of constructions on automata. For example, we can compare automata and their transition systems in a quantitative fashion.

Lemma 12. *Give a SWA $\mathbb{A}$, we can obtain $TS(\mathbb{A})$ of size and in time $O(\|\mathbb{A}\|^2 \cdot B_{\mathbb{A}}^2)$.*

The easy proof follows from direct inspection of Definition 8. Since the value of $B_{\mathbb{A}}$ is typically exponential in terms of $\|\mathbb{A}\|$, we observe that likewise the size of $TS(\mathbb{A})$ is exponential in terms of $\|\mathbb{A}\|$. We shall later see in Proposition 24 that this exponential factor is essential.

4 A stopwatch automaton for Regulation 561

Aside expressivity and tractability, we stressed naturality as a criterion of models for algorithmic law. In this section and the next section we make the point for stopwatch automata by implementing Regulation 561.

4.1 General considerations

As already mentioned, Regulation 561 is a complex set of articles concerning sequences of activities of truck drivers. Possible activities are *driving*, *resting* or *other work*. The

activities over time are recorded by tachographs and formally understood as words over the alphabet $\Sigma := \{d, r, w\}$. In the real world time units are minutes. Regulation 561 limits driving and work times by demanding breaks, daily rest periods and weekly rest periods, both of which can be regular or reduced under various conditions.

Remark 13. Regulation 561 contains a few laws concerning multi-manning that gives rise to an additional activity *available* and a distinction between breaks and rests. This is omitted in our treatment.

Furthermore, we will assume that the data we work with has the right format, is consistent and is free of errors. From [31] it is clear that this is a far cry from being close to reality. Moreover, Regulation 561 prescribes working in UTC and it is known that no tachograph actually records in UTC. In [24] it is shown that the change from non-UTC to UTC data format actually can lead to erroneous interpretations. In theory it is possible to drive from Barcelona to Passau straight without any rests so that if you record the activities using UTC the driver gets no fine but if you record using UTC the driver will be imprisoned. All these complications will not be considered in this paper.

We construct a stopwatch automaton that accepts precisely the words over Σ that represent activity sequences that are legal according to Regulation 561. The states Q of the automaton are:

drive, break, other work,
reduced daily, regular daily, reduced weekly, regular weekly,
compensate1, compensate2, week, start, accept.

The states in the first row have the obvious meaning. The second row collects states representing different kinds of rest periods. The function λ labels *other work* by w, *drive* by d and all other states by r. The states *compensate1* and *compensate2* are used for the most complicated part of Regulation 561 that demands certain compensating rest periods whenever a weekly rest period is reduced. The state *week* is auxiliary, and accepting computations spend 0 time in it. The same is true for *start*. So, the λ-labels of *start* and *week* do not matter.

We construct the automaton stepwise implementing one article after the next, introducing stopwatches along the way. For each stopwatch x we state its bound $\beta(x)$ and the states in which it is active, whence specifying the ζ-label of the stopwatch. We shall refer to stopwatches that are nowhere active as *counters* or *registers*, depending on their informal usage; a *bit* is a counter with bound 1.

We describe a transition (q, g, α, q') saying that there is a transition from q to q' with guard g and action α. We specify guards by a list of expressions of the form $z \leqslant r$ or $z + z' > r$ or the like for $r \in \mathbb{N}$; this is shorthand for a circuit that checks the conjunction of these conditions. We specify actions by lists of expressions of the form $z := r$ or $z := z' + r$ or the like for $z, z' \in X$ and $r \in \mathbb{N}$; this is shorthand for the action that carries out the stated re-assignments of values in the order given by the list.

These lists are also described stepwise treating one article after the next. As a mode of speech, when treating a particular law, we shall say that a given transition has this or that action or guard: what we mean is that the actions or guards of the transition

of the final automaton is given by the lists of these statements in order of appearance (mostly the order won't matter).

We illustrate this mode of speech by describing the automaton around *start*: let x_{start} be a stopwatch with bound 1 and active at *start*; there are no transitions to *start* and transitions from *start* to all other states except *week*; these transitions have guard $x_{start} = 0$.

Later these transitions shall get more guards and also some actions. These stipulations mean more precisely the following: the bound β satisfies $\beta(x_{start}) = 1$; the set Δ contains for any state $q \notin \{week, start\}$ the transition $(start, g, \alpha, q)$ where the guard g checks the conjunction of $x_{start} = 0$ and the other guards introduced later, and the action α carries out the assignments and re-assignments as specified later; the function ζ satisfies that $x_{start} \in \zeta(q)$ if and only if $q = start$.

4.2 The result of a lengthy description

We concluded the last subsection by introducing some stopwatches and describing corresponding bounds and actions. In the next section we will follow article by article and translate this to our automaton. Thus, from each piece of text, possibly, new stopwatches, new states, actions, constraints and bounds are defined.

The final result will be a complex automaton with 12 states and numerous transitions. We include here in Figure 1 an image of a high level representation of the resulting automaton.

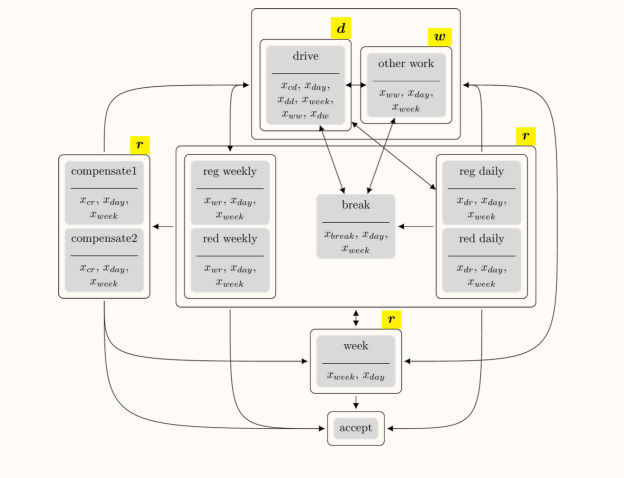

Figure 1: Schematic bird-eye view of our Regulation 561 automaton

The figure includes all states except start and accept and per state it indicates the main stopwatches active in it (without the corresponding bounds). Also, the transitions are depicted but without the corresponding guards and actions. For an earlier version of this paper we collected all the describes attributes of the transitions and a long table. To get an idea of the implicitly defined transition table, we refer the reader to a poster that includes this table at https://www.ub.edu/prooftheory/event/lawdesign/.

5 A stopwatch automaton for Regulation 561: details

We loosely divide Regulation 561 into daily and weekly demands. We first describe how to implement the daily demands using the first 5 states and *daily driving* and *accept*. The other states will be used to implement the weekly demands.

During the construction we shall explicitly collect the constants appearing in the articles and denote them by $t_0, \ldots, t_{16}$. Our construction is such that these constants determine all guards, actions and bounds in an obvious way. Knowing this will be useful for the discussion in later sections.

5.1 Daily demands

We use the first 3 states to implement the the law about *continuous driving*:

> Article 7 (1st part): After a driving period of four and a half hours a driver shall take an uninterrupted break of not less than 45 minutes, unless he takes a rest period.

First we shall address the scenario of a driving period without not a single minute of interruption in between. Next we shall consider the scenario where intertwined rest-minutes are allowed.

We use a stopwatch x_{cd} with bound $4.5h + 1 = 271$ that is active in *drive*. Further, we use a stopwatch x_{break} with bound $9h$ that is active in *break*. For the law under consideration we could use the bound of $4.5h + 1$, the reason we use $9h$ will become clear later when implementing Article 8.7.

There are transitions back and forth between any two of the states *break*, *drive* and *other work*. The transitions from *break* to both *drive* and *other work* have guard $x_{break} \geqslant 45$ and action $x_{break} := 0;\ x_{cd} := 0$. All transitions leaving *drive* have guard $x_{cd} \leqslant 4.5h$. This ensures that a computation staying in *drive* for more than 4.5h will not be able to leave this state, so cannot be accepting.

Transitions to *regular daily* and *reduced daily* have action $x_{cd} := 0$: this ensures the "unless..." statement in Article 7 (transitions to weekly rest periods described below will also have this action). The first part of this Article 7 uses constants $t_0 := 4.5h = 270;\ t_1 := 45$ (the constant $9h$ is denominated later by t_{16}).

Article 7 of Regulation 561 allows to divide the demanded break into two shorter ones:

> Article 7 (2nd part): This break may be replaced by a break of at least 15 minutes followed by a break of at least 30 minutes each distributed over the period in such a way as to comply with the provisions of the first paragraph.

To implement this possibility, we use a bit b_{rb} that, intuitively, indicates a reduced break. We add transitions from *break* to *other work* and *drive* with guard $15 \leqslant x_{break} < 45$ and action $b_{rb} := 1;\ x_{break} := 0$. We note that these transitions do not have action $x_{cd} := 0$. We also observe that this transition is also allowed when $b_{rb} = 1$ in which case setting b_{rb} to one will have simply no effect.

We add transitions[10] from *break* to *other work* and *drive* with guards $b_{rb} = 1$ and $30 \leqslant x_{break}$ and action $b_{rb} := 0$; $x_{cd} := 0$; $x_{break} := 0$. Transitions to states representing daily or weekly rests introduced below all get action $b_{rb} := 0$. The second part of Article 7 uses the constant $t_2 := 15$; we do not introduce a name for 30 but view this constant as equal to $t_1 - t_2 = 45 - 15$.

Remark 14. We observe that Article 7.2 strictly speaking is inconsistent in the following sense. The second part of Art. 7.2 describes a situation which is in conflict with the first paragraph but allowed by way of exception. So far so good, but then it says "in such a way as to comply with the provisions of the first paragraph" which we observed is impossible. This is an innocuous inconsistency because everyone will simply tacitly understand that this last phrase should simply be ignored. However, it is a decision that needs to be made to consistently interpret the law and in a sense, it is a free choice up to the programmer or modeller. More subtle examples of the modeller taking essential interpretational decisions are dealt with in [24, 31].

So far, the automaton does not allow to rest for e.g. 5 minutes. To allow this we add transitions back and forth between *break* and *other work*, *drive* where the transitions from *break* have guard $x_{break} < 15$ and action $x_{break} := 0$.

Remark 15. Actually, the concept of *continuous driving* is underspecified in the regulation. The interpretation that we have chosen here seems a natural one. However, according to our interpretation, as far as Artricle 7 is concerned, it is legal for a driver to spend 9 hours straight spending two minutes driving followed by two minutes of rest to generate the word $(ddrr)^{135}$. It seems doubtful that this is in line with the *spirit of the law*. As a matter of fact, there is another European regulation ((EU) 2016/799) that implies that *drd* cannot happen and any minute of rest between two minutes of driving will be considered as driving. However, alternating periods of two minutes is not considered by this regulation.

[10] We observe that the transitions that we define here involving the bit b_{rb} actually subsume the previously defined transitions in the following sense. A driving period of 4.5h straight followed by a 45m break can lead to an accepting computation also if the automaton leaves the driving state after 4.5h to move to the break state to stay there 15m, then move to driving setting $b_{rb} := 1$ to immediately move back to the break state and stay there for another 30m. However, the purpose of this section is not to provide a minimal or optimal automaton. Rather are we interested in conveying the expressive power of the SWA formalism which allows for an implementation of the law, article by article.

Remark 16. An important problem with the formal ontology of *continuous driving* is that it is not a physical observable like speed. Neither does it seem to be defined in an unambiguous way in terms of physical observables. Consequently we run into troubles as, for example, the one mentioned in Remark 5.1

The part of the automaton defined so far that only involves the states *drive*, *break* and *other work* is depicted in Figure 2. We shall no longer graphically represent the defined transitions with corresponding guards and actions but encourage the reader to do so while reading the remainder of this section.

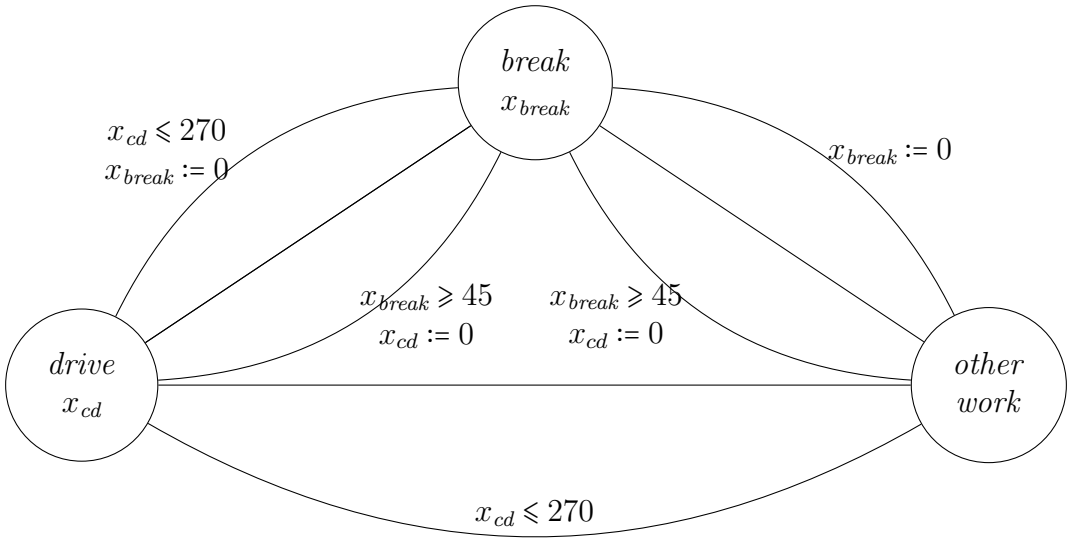

Figure 2: Illustration of Article 7 (first part); stopwatches x_{cd}, x_{break} are shown at the states where they are active.

Article 4.(k) defines 'daily driving time' as the accumulated driving time between two daily rest periods.

Remark 17. There is a degenerate boundary case that is problematic to this Definition 4.(k). Namely when a driver is new to the office. According to just this regulation, his corresponding driver card will not have any (daily) rest period yet so there cannot be any daily driving time either. Of course, there is an easy and natural way to deal with this academic anomaly. But again, this is an example of a (straightforward in this case) decision left to the programmer/modeler.

According to Article 4.(g) daily rest periods can be regular or reduced, the former

meaning at least $11h$ of rest, the latter means less than $11h$ but at least $9h$ of rest. These are represented by the states *regular daily* and *reduced daily*.

> Article 8.1: A driver shall take daily and weekly rest periods.
>
> Article 8.2: Within each period of 24 hours after the end of the previous daily rest period or weekly rest period a driver shall have taken a new daily rest period. If the portion of the daily rest period which falls within that 24 hour period is at least nine hours but less than 11 hours, then the daily rest period in question shall be regarded as a reduced daily rest period.

Weekly rest periods are treated in the next subsection. We use a stopwatch x_{day} with bound $24h+1$ which is active in all states except *accept* and *start*, and a stopwatch x_{dr} with bound $11h$ active in *reduced daily* and *regular daily*. We have transitions back and forth between the states *break*, *drive*, *other work* and the states *regular daily*, *reduced daily*. The transitions to *regular daily* are guarded by $x_{day} \leqslant 24h - 11h = 780;\ b_{rb} = 0$; transitions to *reduced daily* are guarded by $x_{day} \leqslant 24h - 9h = 900;\ b_{rb} = 0$. The transitions from *regular daily* are guarded by $x_{dr} \geqslant 11h$, and the transitions from *reduced daily* are guarded by $11h > x_{dr} \geqslant 9h$ – later we shall refer to these guards as *definitorial* for the states *regular daily* and *reduced daily*. Transitions from *regular daily*, *reduced daily* have action $x_{dr} := 0, x_{day} := 0$.

All transitions to *accept* get guard $x_{day} \leqslant 24h$. Note that an accepting computation cannot involve an assignment satisfying $x_{day} > 24h$, so eventually has to visit and leave *regular daily*, *reduced daily* (or their weekly counterparts, see below). This ensures Article 8.1 for daily rest periods. These laws use constants $t_3 := 24h = 1440, t_4 := 11h = 660, t_5 := 9h = 540$.

Actually the definition of regular daily rest periods in Article 4.(g) is more complicated:

> 'regular daily rest period' means any period of rest of at least 11 hours. Alternatively, this regular daily rest period may be taken in two periods, the first of which must be an uninterrupted period of at least 3 hours and the second an uninterrupted period of at least nine hours,

To implement this we use a bit b_{dr} indicating that a $3h$ part of a regular daily rest period has been taken. We duplicate the transitions from *regular daily* but replace the guard $x_{dr} \geqslant 11h$ by $x_{dr} \geqslant 9h, b_{dr} = 1$. To add the possibility of taking a partial regular daily rest period of at least $3h$ we add transitions from *regular daily* to *drive* and *other work* with guards $b_{dr} = 0, 3h \leqslant x_{dr} < 11h$ and action $b_{dr} := 1$; note these transitions do not have action $x_{day} := 0$. All transitions with action $x_{day} := 0$ also get action $b_{dr} := 0$, including those modeling weekly rest periods described below. This uses the constants $t_6 = 3h = 180, t_7 := 9h = 540$.

> Article 6.1: The daily driving time shall not exceed nine hours. However, the daily driving time may be extended to at most 10 hours not more than twice during the week.

Remark 18. We recall that daily driving times are periods that are delimited by daily rest periods and as such a single daily driving time can very well be spread over two different calendar days. The week however is defined as calendar week starting at Monday 00:00. Now, what happens if a driver has a daily driving period of 10 hours starting on a Sunday and ending on a Monday? This is an extended daily driving time. Should it be counted to the week staring on that Monday or to the week ending on that Sunday? The law seems underspecified here. We shall see that our model will disambiguate by assigning it to the week that starts on Monday. Various tachograph readers make differenct choices and, for example, the sofware *Police Controller* has an option to fix your choices or to choose the distribution as to minimize the fine.

To implement Article 6.1 we use a stopwatch x_{dd} active at *drive* with bound $10h+1$ to measure the daily driving time. Additionally, we use a counter c_{dd} with bound 3. As described later, this counter will be reset to 0 when the week changes. Duplicate the transitions to *regular daily* and *reduced daily*: one gets guard $x_{dd} \leqslant 9h$, the other guard $9h < x_{dd} \leqslant 10h$ and action $c_{dd} := c_{dd} + 1$. Transitions from *regular daily* and *reduced daily* get guard $c_{dd} \leqslant 2$.

5.2 Weekly demands

While Regulation 561 has a concept of a day determined by daily rest periods, its concept of week is that of the calendar (Article 4.(i)).

Remark 19. Actually, the *noun* week is technically defined as a calendar week, starting on a Monday at 00:00 and ending at Sunday at 24:00. The regulation does not explicitly mention what should be done a leap second is added at Sunday so that the moment 24:00:01 exists.
In addition, it is clear from the regulation that the adjective *weekly* does not refer to the technical definition of calendar week. For example, Article 8.9 says

> A weekly rest period that falls in two weeks may be counted in either week, but not in both.

This is an example of misleading nomenclature.

Our formalization of real tachograph recordings by timed words replaces the (allegedly UTC) time-points of tachograph recordings by numbers starting from 0. Hence time is shifted and the information of the beginning of weeks is lost. A possibility to remedy this is to use timed words where the beginnings of weeks are marked, or at least the first of them. For simplicity, we restrict attention to tachograph recordings

starting at the beginning of a week, that is, we pretend that time-point 0 starts a week. We then leave it to the automaton to determine the time-points when weeks change.

To this end[11], we use the auxiliary state *week* and a stopwatch x_{week} with bound $7 \cdot 24h + 1 = 168h + 1$ that is active at all states except *accept* and *start*. All transitions to *accept* are guarded by $x_{week} \leqslant 168h$. The state *week* has incoming transitions from all states except *accept* and transitions to all states except *start*. All these incoming transitions are guarded by $x_{week} = 168h$. The outgoing transitions have guard $x_{week} = 168h - 1$ actions $x_{week} := 0$ and $c_{dd} := 0$ (see the implementation of Article 6.1 above). This ensures that every accepting computation of $\mathbb{A}$ enters *week* for 0 time units exactly every week, i.e., every $168h$.

Additionally, we want the automaton to switch from *week* back to the state it came from. To this end we introduce a bit b_q for each state $q \neq$ *accept* with a transition to *week*. We give the transition from State q to *week* the action $b_q := 1$, and the transition from *week* to State q the guard $b_q = 1$ and the action $b_q := 0$. The transition from *week* to *accept* has no guard involving the bits b_q. This uses the constant $t_{10} := 168h = 10080$.

Much of the following implementation work is done by adding guards and actions to the transitions from and to *week*. For example, we can readily implement

> Article 6.2: The weekly driving time shall not exceed 56 hours and shall not result in the maximum weekly working time laid down in Directive 2002/15/EC being exceeded.
>
> Article 6.3: The total accumulated driving time during any two consecutive weeks shall not exceed 90 hours.

The time laid down by Directive 2002/15/EC is $60h$. Use a stopwatch x_{ww} with bound $60h + 1$ that is active at *drive* and *other work*. Use a stopwatch x_{dw} with bound $56h + 1$ active at *drive*. To implement Article 6.2, the transitions to *week* and *accept* have guard $x_{dw} \leqslant 56h, x_{ww} \leqslant 60h$, and the transitions from *week* have action $x_{dw} := 0, x_{ww} := 0$. Note that accepting computations contain only nodes with assignments satisfying $x_{dw} \leqslant 56h$ and $x_{ww} \leqslant 60h$. This implements Article 6.2.

To implement Article 6.3 we have to remember the value x_{dw} of the previous week. We use a register x'_{dw} with the same bound as x_{dw} and give the transitions from *week* the action $x'_{dw} := x_{dw}$. Note x'_{dw} functions like a register in that it just stores a value. We then guard all transitions to *accept* by $x'_{dw} + x_{dw} \leqslant 90h$. These articles use constants $t_{11} := 56h = 3360, t_{12} := 60h = 3600$ and $t_{13} := 90h = 5400$.

We now treat the articles concerning weekly rest periods (again, these have little to do with the formal definition of calendar week). According to Article 4.(h), weekly rest periods can be regular or reduced, the former meaning at least $45h$ of rest, the latter means less than $45h$ but at least $24h$ of rest. These rest periods are represented by the states *regular weekly* and *reduced weekly*.

To implement their definition we use a stopwatch x_{wr} with bound $45h$ active in these two states to record the resting time. For the two states we add transitions from and to *drive* and *other work* and transitions to *accept*: those from *regular weekly* have guard $x_{wr} \geqslant 45h$ and action $x_{wr} := 0$, and those from *reduced weekly* have guards

[11] Alternatively, the week-status could be coded in the raw data by preprocessing it, thereby avoiding this large constant.

$45h > x_{wr} \geqslant 24h$ and action $x_{wr} := 0$. Later we shall refer to these guards as *definitorial guards* for *regular weekly* and *reduced weekly*, respectively. This uses the constants $t_{14} := 45h = 2700, t_{15} := 24h = 1440$.

We start with some easy implementations:

> Article 8.6 (3rd part): A weekly rest period shall start no later than at the end of six 24-hour periods from the end of the previous weekly rest period.
>
> Article 8.3: A daily rest period may be extended to make a regular weekly rest period or a reduced weekly rest period.
>
> Article 8.4: A driver may have at most three reduced daily rest periods between any two weekly rest periods.

Article 8.6 (3rd part) is implemented by adding the guard $x_{week} \leqslant 6 \cdot 24h$ to the transitions to *regular weekly* and *reduced weekly*. This law uses constant $t_{16} := 6 \cdot 24h = 8640$.

To implement Article 8.3 we simply copy the guards and actions of the transitions from *drive* and *other work* to *regular daily* to the corresponding transitions to both *regular weekly* and *reduced weekly*. Below we shall add more guards and actions. For Article 8.4 we use a counter c_{rd} with bound 4. We add guard $c_{rd} \leqslant 2$ and action $c_{rd} := c_{rd} + 1$ to the transitions to *reduced daily* and the action $c_{rd} := 0$ to the transitions from *reduced weekly* and *regular weekly*.

We still have to implement Article 8.1 for weekly rest periods, and additionally

> Article 8.9: A weekly rest period that falls in two weeks may be counted in either week, but not in both.

We use two bits b_{wr}, b_{used} meant to indicate whether a weekly rest period has been taken in the current week, and whether the current weekly rest period is used for this. The transitions from *drive* or *other work* to *reduced weekly* or *regular weekly* are duplicated: one gets guard $b_{wr} = 0$ and action $b_{used} := 1;\ b_{wr} := 1$, the other gets no further guards and actions. Transitions from *reduced weekly* or *regular weekly* get action $b_{used} := 0$. The transitions to *week* get guard $b_{wr} = 1$.

Each transition from *week* to *reduced weekly* or *regular weekly* is triplicated: the first gets additional guard $b_{used} = 1$ and action $b_{used} := 0;\ b_{wr} := 0$, the second gets guard $b_{used} = 0$ and action $b_{wr} := 0$, and the third gets guard $b_{used} = 0$ and action $b_{used} := 1;\ b_{wr} := 1$. This means that when the week changes during a weekly rest period and this rest period is not used, it can be used for the next week.

The most complicated part of Regulation 561 are the rules governing reductions of weekly rest periods. The regulation starts as follows:

> Article 8.6 (1st part): In any two consecutive weeks a driver shall take at least two regular weekly rest periods, or one regular weekly rest period and one reduced weekly rest period of at least 24 hours.

We use a bit b_{rw} indicating whether the previous weekly rest period was reduced: transitions to *reduced weekly* have guard $b_{rw} = 0$ and action $b_{rw} := 1$. Transitions to *regular weekly* have action $b_{rw} := 0$.

The regulation continues as follows:

Article 8.6 (2nd part): However, the reduction shall be compensated by an equivalent period of rest taken en bloc before the end of the third week following the week in question.

Remark 20. Article 8.6 introduces much combinatoric complexity and it is not clear if this was intended when Regulation 561 was first formulated. In [25] and [24] it is shown that the legality of a given week may depend on what happens during weeks that are arbitrary far in time removed from it.

The next article reads:

Article 8.7: Any rest taken as compensation for a reduced weekly rest period shall be attached to another rest period of at least nine hours.

We introduce two registers x_{c1}, x_{c2} with bounds $45h{-}24h$. We shall use the following informal mode of speech for the discussion: a reduced weekly rest period creates a 'compensation obligation', namely an additional resting time $x_{c1} > 0$ or $x_{c2} > 1$. The obligations are 'fulfilled' by setting these registers back to 0. Note that compensation obligations are created by reduced weekly rest periods and, by Article 8.6 (1st part), this can happen at most every other week. As obligations have to be fulfilled within 3 weeks, at any given time a legal driver can have at most two obligations.

We now give the implementation. Obligations are produced by transitions from *reduced weekly* (recall x_{wr} records the resting time in *reduced weekly*): duplicate each such transition, give one guard $x_{c1} = 0$ and action $x_{c1} := 45h - x_{wr}$, and the other guard $x_{c1} > 0$; $x_{c2} = 0$ and action $x_{c2} := 45h - x_{wr}$. The 3 week deadline to fulfill the obligations is implemented by two counters c_{c1}, c_{c2} with bound 4. These counters are increased by transitions from *week* but only if some obligation is actually recorded: transitions from *week* get action $c_{c1} := c_{c1} + \mathrm{sgn}(x_{c1})$; $c_{c2} := c_{c2} + \mathrm{sgn}(x_{c2})$. To ensure the deadline, transitions to *week* get guard $c_{c1} \leqslant 3$; $c_{c2} \leqslant 3$.

We now implement the way obligations can be fulfilled, i.e., how to set x_{c1} and x_{c2} back to 0. This is done with the states *compensate1* and *compensate2* whose λ-label is r. We use a stopwatch x_{cr} with bound $45h - 24h$ active at these states. We describe the transitions involving *compensate1*. It receives transitions from the states with λ-label r, that is, *regular daily*, *reduced daily*, *regular weekly*, *reduced weekly* and *break*. The transition from *break* has guard $x_{break} \geqslant 9h$, the others have their respective definitorial guards (e.g., the one from *regular weekly* has guard $x_{wr} \geqslant 45h$). Transitions from *compensate1* go to *drive*, *other work* and *accept*. These have guard $x_{cr} \geqslant x_{c1}$ and action $x_{c1} := 0 : \ c_{c1} := 0$. Additionally, we already introduced transitions from and to *week*: the transition to *week* is duplicated, one gets guard $x_{cr} < x_{c1}$, the other gets guard $x_{cr} \geqslant x_{c1}$ and action $x_{c1} := 0$. Thus, when the week changes during compensation and at a time-point when the obligation is fulfilled, the counter is not increased.

The transitions from and to *compensate2* are analogous with x_{c2}, c_{c2} replacing x_{c1}, c_{c1}. These laws use the constant $t_{16} := 9h = 540$; the bound $45h - 24h = 1560$ equals $t_{14} - t_{15}$.

This finishes the definition of our automaton.

6 Expressivity of Stopwatch Automata

In this section we will gauge the expressivity of Stopwatch automata and see an equivalent alternative definition for SWAs.

6.1 Regularity

The next lemma establishes an easy lower-bound on the expressibility of stopwatch automata. In a sense, we can consider non-deterministic finite automata as a special case of stopwatch automata.

Lemma 21. *Every regular language is the language of some stopwatch automaton.*

Proof. Given a non-deterministic finite automaton $\mathbb{B} = (S, \Sigma, I, F, \Gamma)$ as described in Subsection 2.1, we will define the stopwatch automaton $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ so that a word is in $L(\mathbb{B})$ if and only if it is accepted by $\mathbb{A}$.

To this end we define the states Q of $\mathbb{A}$ to be *start* and *accept* together with $\{(s,a) \mid \exists\, s' \in S\ (s,a,s') \in \Gamma\}$ so that the old transition symbol a is hardwired in a state of $\mathbb{A}$ in the sense that we define $\lambda\big((s,a)\big) := a$ (the labelling on *start* and *accept* is irrelevant by construction). In the case $I \cap F \neq \varnothing$[12] we add a transition from *start* to *accept* with condition nor action.

To mimic the behaviour of $\mathbb{B}$ we will use a single clock x that is active everywhere so that $X = \{x\}$ and $\zeta(x) = Q$. We use the stopwatch x to force each computation to stay exactly one time unit in states $(s,a) \in Q$. Thus, a bound of $\beta(x) := 2$ suffices and we define that each transition from some (s,a) to (s',a') has condition $x = 1$ and action $x := 0$. We only allow those transitions from (s,a) to (s',a') when $(s,a,s') \in \Gamma$. This fixes Δ when the states are not *start* or *accept*. We further define Δ by stipulating that there is a transition from *start* to any $(s,a) \in Q$ so that $s \in I$ with guard $x = 0$ and action $x := 0$. Likewise, we complete the definition of Δ by stipulating that there is a transition from any $(s,a) \in Q$ so that $s \in F$ to *accept* with guard $x = 0$ and action $x := 0$.

It is now easy to see that $L(\mathbb{A}) \subseteq L(\mathbb{B})$ and the other inclusion could have only failed in case $I \cap F \neq \varnothing$ which is why we added an extra transition in this case. □

We shall now see in Theorem 23 below that the converse of this lemma is also true and that any language recognised by a SWA is regular. The construction to go from a SWA $\mathbb{A}$ to an equivalent finite automata $\mathbb{B}(\mathbb{A})$ is given by the following defintion.

Definition 22. Given a stopwatch automaton $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$, we define the following finite automaton $\mathbb{B}(\mathbb{A}) = (S, \Sigma, I, F, \Gamma)$. For S we take the set of nodes of $TS(\mathbb{A})$; we let $I := \{(start, \xi_0)\}$ where ξ_0 is constantly 0, and F contain the nodes (q,ξ) of $TS(\mathbb{A})$ such that $(q,\xi) \xrightarrow{0^*} (accept, \xi')$ for some assignment ξ'. Here, $\xrightarrow{0^*}$ denotes the transitive and reflexive closure of $\xrightarrow{0}$. We let Γ contain $\big((q,\xi), a, (q',\xi')\big)$ if $\lambda(q') = a$ and there is ξ'' such that $(q,\xi) \xrightarrow{0^*} (q',\xi'') \xrightarrow{1} (q',\xi')$.

[12] For example, the minimal automaton $\mathbb{B}$ with $L(\mathbb{B}) = \{\lambda\}$. $I \cap F \neq \varnothing$ can be replaced by requiring that $I \cap F$ contains isolated points.

With this construction we now gauge the expressivity of Stopwatch Automata.

Theorem 23. *A language is regular if and only if it is the language of some stopwatch automaton.*

Proof. One direction follows from Lemma 21. Conversely, given a language that is recognised by some SWA $\mathbb{A}$ we easily see that $L(\mathbb{A}) = L(\mathbb{B}(\mathbb{A}))$ where $\mathbb{B}(\mathbb{A})$ is as in Definition 22. That $L(\mathbb{B}(\mathbb{A})) \subseteq L(\mathbb{A})$ is immediate and for $L(\mathbb{A}) \subseteq L(\mathbb{B}(\mathbb{A}))$ we observe that a step in $TS(\mathbb{A})$ of duration n can be obtained by n consecutive steps of duration 1. □

The proof of Lemma 21 gives a polynomial time computable function mapping every finite automaton to an equivalent stopwatch automaton. There is no such function for the converse translation, in fact, stopwatch automata are exponentially more succinct than finite automata.

Proposition 24. *For every k there is a stopwatch automaton $\mathbb{A}_k$ of size $O(\log k)$ such that every finite automaton accepting $L(\mathbb{A}_k)$ has size at least k.*

We defer the proof to the end of Section 8.2.

6.2 A definitorial variation

To showcase the robustness of our definition and for later use, we mention a natural variation of our definition and show it is inessential.

Define a *$P(\Sigma)$-labeled (specific) stopwatch automaton* $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ like a (specific) stopwatch automaton but with $\lambda : Q \to P(\Sigma) \setminus \{\varnothing\}$. A computation

$$(q_0, \xi_0) \xrightarrow{t_0} (q_1, \xi_1) \xrightarrow{t_1} (q_2, \xi_2) \xrightarrow{t_2} \cdots \xrightarrow{t_{\ell-1}} (q_\ell, \xi_\ell). \tag{1}$$

is said to *read* any word $a_0^{t_0} \cdots a_{\ell-1}^{t_{\ell-1}}$ with $a_i \in \lambda(q_i)$ for every $i < \ell$. The language $L(\mathbb{A})$ of $\mathbb{A}$ is defined as before.

A stopwatch automaton can be seen as a $P(\Sigma)$-labeled stopwatch automaton whose state labels are singletons. Conversely, given a $P(\Sigma)$-labeled stopwatch automaton $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ we define a stopwatch automaton $\mathbb{A}' = (Q', \Sigma, X, \lambda', \beta, \zeta', \Delta')$ as follows: its states Q' consist of *start*, *accept* and of pairs (q, a) such that $a \in \lambda(q)$. No stopwatches are active in *start*, nor in any of the states $(accept, a)$ with $a \in \lambda(accept)$. The λ'-label of such a state (q, a) is a and stopwatch $x \in X$ is active (according ζ') in (q, a) if and only if it is active in q (according ζ). We let Δ' contain a transition $((q, a), g, \alpha, (\tilde{q}, \tilde{a}))$ if and only if $(q, a), (\tilde{q}, \tilde{a}) \in Q'$ and $(q, g, \alpha, \tilde{q}) \in \Delta$.

Given a computation of $\mathbb{A}$ as above, choose any $a_i \in \lambda(q_i)$ for every $i < \ell$. Then

$$((q_0, a_0), \xi_0) \xrightarrow{t_0} ((q_1, a_1), \xi_1) \xrightarrow{t_1} ((q_2, a_2), \xi_2) \xrightarrow{t_2} \cdots \xrightarrow{t_{\ell-1}} ((q_\ell, a_\ell), \xi_\ell). \tag{2}$$

is a computation of $\mathbb{A}'$. The choice of the a_i can be made so that this computation reads the same word as the computation (1). Conversely, if (2) is a computation of $\mathbb{A}'$, then (1) is a computation of $\mathbb{A}$ that reads the same word. To sum up we now have:

Proposition 25. *There is a polynomial time computable function that maps every $P(\Sigma)$-labeled stopwatch automaton $\mathbb{A}$ to a stopwatch automaton $\mathbb{A}'$ with $B_{\mathbb{A}} = B_{\mathbb{A}'}$ and $L(\mathbb{A}) = L(\mathbb{A}')$.*

7 Model checking for Stopwatch Automata

In this section we see how the classical automata problems behave for SWAs: consistency checking and validity checking.

7.1 Consistency-checking

By Theorem 23 we know that the languages accepted by SWAs are exactly the regular languages. In particular we know that those languages are closed under intersections. Since we will need explicit bounds on a SWA that computes the intersection of two languages, we will now provide and analyse a construction for such an automaton.

The idea is that the product automaton can allocate simultaneous computations of the respective automata. An additional fresh clock variable is used to flag when the two automata computations start to differ.

Definition 26. For $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ and $\mathbb{A}' = (Q', \Sigma, X', \lambda', \beta', \zeta', \Delta')$. We define the product automaton $\mathbb{A} \otimes \mathbb{A}'$ as follows.

Without loss of generality we can assume that

- X and X' are disjoint;
- both $\mathbb{A}$ and $\mathbb{A}'$ have distinct start and accept states;
- neither $\mathbb{A}$ nor $\mathbb{A}'$ contains transitions from its accept state.

Let y be a fresh clock variable not yet in $X \cup X'$. We define

$$\mathbb{A} \otimes \mathbb{A}' := (Q \times Q', \Sigma, X \cup X' \cup \{y\}, \lambda^{\otimes}, \beta^{\otimes}, \zeta^{\otimes}, \Delta^{\otimes}),$$

where

- the start state is the pair of the start states of $\mathbb{A}$ and $\mathbb{A}'$ and the accept state is the pair of accept states of $\mathbb{A}$ and $\mathbb{A}'$;
- $\lambda^{\otimes}$ maps $(q, q') \in Q \times Q'$ to $\lambda(q)$;
- $\beta^{\otimes}$ extends $\beta \cup \beta'$ with $\beta^{\otimes}(y) = 1$;
- $\zeta^{\otimes}$ maps $x \in X$ to $\zeta(x) \times Q'$ and $x \in X'$ to $Q \times \zeta'(x)$ and $\zeta^{\otimes}(y) = Q \times Q'$;
- $\Delta^{\otimes}$ contains $((q_0, q_0'), g^*, \alpha^*, (q_1, q_1'))$ iff one of the following holds
  - For some guards g, g' and actions α, α' we have that $(q_0, g, \alpha, q_1) \in \Delta$ and $(q_0', g', \alpha', q_1') \in \Delta'$ and α^* executes α and α' in parallel setting the value of y to zero. If $\lambda(q) = \lambda(q')$, then g^* simply computes the conjunction of g and g', and in case $\lambda(q) \neq \lambda(q')$, then g^* computes the conjunction of g, g' and the requirement that $y = 0$;
  - For some guard g and action α we have that $(q_0, g, \alpha, q_1) \in \Delta$ and $q_0' = q_1'$ and α^* executes α in parallel with the identity map on X' and sets $\alpha^*(y) = 0$. If $\lambda(q) = \lambda(q')$, then the guard g^* only checks g. If $\lambda(q) \neq \lambda(q')$, then the guard g^* checks g in conjunction with $y = 0$;
  - For some guard g' and action α' we have that $q_0 = q_1$ and $(q_0', g', \alpha', q_1') \in \Delta'$ and α^* executes the identity map on X in parallel with α' and sets $\alpha^*(y) = 0$. If $\lambda(q) = \lambda(q')$, then the guard g^* only checks g'. If $\lambda(q) \neq \lambda(q')$, then the guard g^* checks g' in conjunction with $y = 0$.

It is an elementary exercise to see that the thus defined product automaton possesses the required behaviour.

Lemma 27. *Given stopwatch automata $\mathbb{A}, \mathbb{A}'$ with bounds $B_{\mathbb{A}}, B_{\mathbb{A}'}$ one can compute in time $O(\|\mathbb{A}\| \cdot \|\mathbb{A}'\|)$ the stopwatch automaton $\mathbb{A} \otimes \mathbb{A}'$ with bound $B_{\mathbb{A}} \cdot B'_{\mathbb{A}}$ such that $L(\mathbb{A} \otimes \mathbb{A}') = L(\mathbb{A}) \cap L(\mathbb{A}')$.*

Proof. Note that computations of $\mathbb{A} \otimes \mathbb{A}'$ cannot leave a state (q, q') where one of q, q' is the accept state of $\mathbb{A}$ or $\mathbb{A}'$. Hence an initial accepting computation of $\mathbb{A} \otimes \mathbb{A}'$ visits such a state exactly in its last step. It follows that such computations correspond to pairs of initial accepting computations of $\mathbb{A}$ and $\mathbb{A}'$. However, the two computations in this pair need not read the same word. We have that the words in $L(\mathbb{A}) \cap L(\mathbb{A}')$ are precisely the words accepted by $\mathbb{A} \otimes \mathbb{A}'$ via accepting computations that do not reside for a positive amount of time in a *bad* state (q, q') with $\lambda(q) \neq \lambda'(q')$.

However, the new clock variable y was declared active in every state $(q, q') \in Q \times Q'$ and all actions set y to 0. Moreover each guard of a transition from a bad state check that $y = 0$. Thus indeed, no computation can leave a bad state after having resided in it for a positive amount of time. We note that $\mathbb{A} \otimes \mathbb{A}'$ has size $O(\|\mathbb{A}\| \cdot \|\mathbb{A}'\|)$ and bound $B_{\mathbb{A} \otimes \mathbb{A}'} = B_{\mathbb{A}} \cdot B_{\mathbb{A}'}$. Clearly, $\mathbb{A} \otimes \mathbb{A}'$ can be computed from $\mathbb{A}$ and $\mathbb{A}'$ in time linear in the size of $\mathbb{A} \otimes \mathbb{A}'$, that is, linear in $O(\|\mathbb{A}\| \cdot \|\mathbb{A}'\|)$. □

The following algorithm can be used to check if the intersection of two languages is empty or not. Informally, we can perceive this as an algorithm that checks whether a certain type of behaviour is illegal according to a law when both the type of behaviour and the law are specified by stopwatch automata.

Theorem 28. *There is an algorithm that given stopwatch automata $\mathbb{A}, \mathbb{A}'$ with bounds $B_{\mathbb{A}}, B_{\mathbb{A}'}$, respectively, decides whether $L(\mathbb{A}) \cap L(\mathbb{A}') \neq \varnothing$ in time*

$$O\big(\|\mathbb{A}\|^3 \cdot \|\mathbb{A}'\|^3 \cdot B_{\mathbb{A}}^2 \cdot B_{\mathbb{A}'}^2\big).$$

Proof. The algorithm first computes the product automaton $\mathbb{A} \otimes \mathbb{A}'$ from Definition 26. Next, the algorithm computes the finite automaton $\mathbb{B}(\mathbb{A} \otimes \mathbb{A}') = (S, \Sigma, I, F, \Gamma)$ as given in Definition 22. Note $|S| \leqslant |Q| \cdot |Q'| \cdot B_{\mathbb{A} \otimes \mathbb{A}'}$.

To compute Γ we first compute the graph on S with edges $\xrightarrow{0}$: cycle through all $(q, \xi) \in S$ and transitions in $\Delta^{\otimes}$ and evaluate its guard and action on ξ. Each evaluation can be done in time linear in the size of the circuits, so in time $O(\|\mathbb{A}^{\otimes}\|)$. Thus, the graph can be computed in time $O(|S| \cdot |\Delta^{\otimes}| \cdot \|\mathbb{A}^{\otimes}\|)$.

Linear time in the size of this graph suffices to compute $\xrightarrow{0^*}$, i.e., to determine for each pair of vertices whether the second is reachable from the first. Each of the at most $|S|^*$ edges in $\xrightarrow{0^*}$ determines a transition in Γ. Thus $\mathbb{B}(\mathbb{A} \otimes \mathbb{A}')$ can be computed in time $O(|S| \cdot |\Delta^{\otimes}| \cdot \|\mathbb{A}^{\otimes}\| + |S|^2) \leqslant O((\|\mathbb{A}\| \cdot \|\mathbb{A}'\|)^3 \cdot B_{\mathbb{A}}^2 \cdot B_{\mathbb{A}'}^2)$.

Observe $L(\mathbb{B}(\mathbb{A} \otimes \mathbb{A}')) \neq \varnothing$ if and only if some final state is reachable from the initial state. Checking this takes linear time in the size of the automaton. □

The algorithm solves the consistency problem for stopwatch automata by fixing input $\mathbb{A}'$ to some stopwatch automaton with $L(\mathbb{A}') = \Sigma^*$.

Corollary 29. *There is an algorithm that given a stopwatch automaton $\mathbb{A}$ with bound $B_{\mathbb{A}}$, decides whether $L(\mathbb{A}) \neq \varnothing$ in time*

$$O\big(\|\mathbb{A}\|^3 \cdot B_{\mathbb{A}}^2\big).$$

7.2 Model-checking

By Theorems 23 and 5, stopwatch automata are as expressive as MSO over finite words. In this section we verify that its model-checking complexity is comparatively tame.

In fact, the algorithm of Theorem 28 can be used to solve the model-checking problem: note $w \in L(\mathbb{A})$ if and only if $L(\mathbb{B}_w) \cap L(\mathbb{A}) \neq \varnothing$ for a suitable size $O(|w|)$ automaton $\mathbb{B}_w$ with $L(\mathbb{B}_w) = \{w\}$. A more direct model-checking algorithm achieves a somewhat better time complexity, in particular, linear in $B_{\mathbb{A}}$:

Theorem 30. *There is an algorithm that given a word w and a stopwatch automaton $\mathbb{A}$ with bound $B_{\mathbb{A}}$ decides whether $w \in L(\mathbb{A})$ in time*

$$O\big(\|\mathbb{A}\|^2 \cdot B_{\mathbb{A}} \cdot |w|\big).$$

Proof. Let $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ have bound $B_{\mathbb{A}}$. Let $G = (V, E)$ be the directed graph whose vertices V are the nodes of $TS(\mathbb{A})$ and whose directed edges E are given by $\xrightarrow{0}$. Note $|V| = |Q| \cdot B_{\mathbb{A}}$ and $|E| \leqslant B_{\mathbb{A}} \cdot |\Delta|$. Let w be a word over Σ, say of length t. We define a directed graph with vertices $\{0, \ldots, t\} \times V$ and the following edges. Edges within each copy $\{i\} \times V$ are copies of E. So these account for at most $(t+1) \cdot B_{\mathbb{A}} \cdot |\Delta|$ many edges, each determined by evaluating guards and actions in time $O(\|\mathbb{A}\|)$. Further edges lead from vertices in the i-th copy $\{i\} \times V$ to vertices in the $(i+1)$th copy $\{i+1\} \times V$, namely from $(i, (q, \xi))$ to $(i+1, (q, \xi'))$ if

$$(q, \xi) \xrightarrow{1} (q, \xi') \text{ and } q \neq \mathit{accept} \text{ and } \lambda(q) = w_i. \tag{3}$$

There are at most $t \cdot |Q| \cdot B_{\mathbb{A}}$ such edges between copies. This graph has size $O(t \cdot \|A\| \cdot B_{\mathbb{A}})$ and can be computed in time $O(t \cdot \|A\|^2 \cdot B_{\mathbb{A}})$.

It is clear that $w \in L(\mathbb{A})$ if and only if $(t, (\mathit{accept}, \xi'))$ for some assignment ξ' is *reachable* in the sense that there is a path from $(0, (\mathit{start}, \xi_0))$ with ξ_0 constantly 0 to it. Checking this takes time linear in the size of the graph. □

This theorem tells us that temporal algorithmic laws that can be represented as SWA (in the above theorem $\mathbb{A}$) have relatively good behaviour to check whether a data-set (above w) is legal according to the law.

8 Pushing the limits for Stopwatch Automata

Oftentimes scheduling a future record is computationally harder than checking legality of a given record. In this section that this is not the case for our Regulation 561 implementation where scheduling is not much harder than checking legality. And this holds for SWAs in general. We also see how our SWAs can be generalised in a natural fasion so that we remain decidable yet go beyond regular languages.

8.1 Scheduling

We strengthen the model-checker of Theorem 30 to solve the scheduling problem: the model-checker treats the special case for inputs with $n = 0$.

Theorem 31. *There is an algorithm that given a stopwatch automaton $\mathbb{A}$ with bound $B_{\mathbb{A}}$ and alphabet Σ, a word $w \in \Sigma^*$, a letter $a \in \Sigma$ and $n \in \mathbb{N}$, rejects if there does not exist a word v over Σ of length n such that $wv \in L(\mathbb{A})$ and otherwise computes such a word v with maximal $\#_a(v)$. It runs in time*

$$O\big(\|\mathbb{A}\|^2 \cdot B_{\mathbb{A}} \cdot (|w| + n)\big).$$

Proof. Consider the graph constructed in the poof of Theorem 30 but with $t+n$ instead of t and the following modification: in (3) for $t \leqslant i < t + n$ drop the condition $\lambda(q) = w_i$ for edges between the i-th and the $(i+1)$-th copy.

In the resulting graph there is a reachable vertex $(t + n, (\mathit{accept}, \xi'))$ for some assignment ξ' if and only if there exists a length n word v such that $wv \in L(\mathbb{A})$. We now show how to compute the maximum value $\#_a(v)$ for such v.

Successively for $i = 0, \dots, n$ compute a label $V_i(q, \xi)$ for each vertex $(t + i, (q, \xi))$ in the $(t + i)$-th copy. For $i = 0$ all these labels are $\#_a(w)$. For $i > 0$ label $(t + i, (q, \xi))$ with the maximum value

$$\begin{cases} V_{i-1}(q, \xi') + 1 & \text{if } \lambda(q) = a, \\ V_{i-1}(q, \xi') & \text{else} \end{cases}$$

taken over ξ' such that there is an edge from $(t + i - 1, (q, \xi'))$ to $(t + i, (q, \xi))$. Then the desired maximum value $\#_a(v)$ is the maximum label $V_n(q, \xi)$ such that $q = \mathit{accept}$ and $(t + n, (q, \xi))$ is reachable.

Additionally we are asked to compute a word v witnessing this value. To do so the labeling algorithm computes a set of directed edges, namely for each $(t + i, (q, i))$ with $i > 0$ to a vertex $(t + i - 1, (q, \xi'))$ witnessing the maximum value above. This set of edges defines a partial function that, for each $i > 0$, maps vertices in the $(t + i)$-th copy to vertices in the $(t + i - 1)$-th copy. To compute v as desired start at a vertex $(t + n, (q_n, \xi_n))$ witnessing the maximal value $\#_a(v)$ and iterate this partial function to get a sequence of vertices $(t + i, (q_i, \xi_i))$. Then $v := \lambda(q_1) \cdots \lambda(q_n)$ is as desired.

It is clear that all this can be done in time linear in the size of the graph. □

8.2 Model-checking beyond regularity

A straightforward generalization of bounded stopwatch automata allows β to take value ∞. An *unbounded stopwatch automaton* is a stopwatch automaton where β is the function constantly ∞.

We shall now prove that model-checking is undecidable already for *simple* such automata (see [16, Proposition 1] for a similar proof). These simple automata use two stopwatches x, y that are nowhere active (i.e., ζ is constantly $\varnothing$), all guards check $z = 0$ or $z \neq 0$, and all actions are either $z := z + 1$ or $z := z \dot{-} 1 = \min\{|z - 1|, z\}$ for some $z \in \{x, y\}$.

Proposition 32. *There is no algorithm that given a simple unbounded stopwatch automaton decides whether it accepts the empty word.*

Proof. Recall, a *two counter machine* operates two variables x, y called *counters* and is given by a finite non-empty sequence $(\pi_0, \dots, \pi_\ell)$ of *instructions* π_i, namely, either $z := z + 1, z := z\dot{-}1$, "Halt" or "if $z = 0$, then goto j, else goto k" where $z \in \{x, y\}$ and $j, k \leqslant \ell$; exactly π_ℓ is "Halt". The computation (without input) of the machine is straightforwardly explained. It is long known that it is undecidable whether a given two counter machine halts or not.

Given such a machine $(\pi_0, \dots, \pi_\ell)$ it is easy to construct a simple automaton that accepts the empty word if and only if the two counter machine halts. It has states $Q = \{0, 1, \dots, \ell\}$ understanding *start* $= 0$ and $\ell =$ *accept*; Σ and λ are unimportant, and Δ is defined as follows. If π_i is the instruction $z := z+1$, then add the edge $(i, g, \alpha, i+1)$ where g is trivial and α changes z to $z + 1$. If π_i is the instruction $z := z\dot{-}1$, proceed similarly. If π_i is "if $z = 0$, then goto j, else goto k" add edges $(i, g, \alpha, j), (i, g', \alpha, k)$ where g checks $z = 0$ and g' checks $z \neq 0$ and α computes the identity. □

What seems to be a middle ground between unbounded stopwatches and stopwatches with a constant bound is to let the bound grow with the length of the input word.

The definition of a bounded stopwatch automaton $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ can be generalized letting $\beta : X \times \mathbb{N} \to \mathbb{N}$ be *monotone* in the sense that $\beta(x, n) \leqslant \beta(x, n')$ for all $x \in X$, $n, n' \in \mathbb{N}$ with $n \leqslant n'$. We call this a *β-bounded stopwatch automaton* and call $B_{\mathbb{A}} : \mathbb{N} \to \mathbb{N}$ defined by

$$B_{\mathbb{A}}(n) := \prod_{x \in X} (\beta(x, n) + 1)$$

the *bound of* $\mathbb{A}$. For each $n \in \mathbb{N}$ we have a stopwatch automaton $\mathbb{A}(n) := (Q, \Sigma, X, \beta_n, \zeta, \lambda, \Delta)$ where $\beta_n : X \to \mathbb{N}$ maps $x \in X$ to $\beta(x, n)$; note $B_{\mathbb{A}(n)} = B_{\mathbb{A}}(n)$.

The *language* $L(\mathbb{A})$ *accepted* by a β-bounded stopwatch automaton $\mathbb{A}$ contains a word w over Σ if and only if $w \in L(\mathbb{A}(|w|))$.

Proposition 33. *A language is accepted by some stopwatch automaton if and only if it is accepted by some β-bounded stopwatch automaton with bounded β.*

Proof. Let $\mathbb{A}$ be a β-bounded stopwatch automaton for bounded β. There is $n_0 \in \mathbb{N}$ such that $\beta(x, n) = \beta(x, n_0)$ for all $x \in X$ and $n \geqslant n_0$. Hence $L(\mathbb{A}(n_0))$ and $L(\mathbb{A})$ contain the same words of length at least n_0. Since there are only finitely many shorter words, and $L(\mathbb{A}(n_0))$ is regular by Theorem 23, also $L(\mathbb{A})$ is regular. □

Theorem 30 on feasible model checking generalizes:

Corollary 34. *Let X be a finite set and assume $\beta : X \times \mathbb{N} \to \mathbb{N}$ is such that $\beta(x, n)$ is computable from $(x, n) \in X \times \mathbb{N}$ in time $O(n)$. Then there is an algorithm that given a word w and a β-bounded stopwatch automaton $\mathbb{A}$ with bound $B_{\mathbb{A}} : \mathbb{N} \to \mathbb{N}$ decides whether $w \in L(\mathbb{A})$ in time*

$$O\big(\|\mathbb{A}\|^2 \cdot B_{\mathbb{A}}(|w|) \cdot |w|\big).$$

The interesting feature is that if the bounds are allowed to grow, then we can recognise non-regular languages. Even if the growth is extremely slow. In a sense, if $\beta(x,n)$ grows slowly in n this can be considered tractable.

Proposition 35. *Let $f : \mathbb{N} \to \mathbb{N}$ be an unbounded function. Then there is a β-bounded stopwatch automaton $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ with $\beta(x,n) = f(n)$ for all $x \in X$ and all $n \in \mathbb{N}$ such that $L(\mathbb{A})$ is not regular.*

Proof. Let Σ be the three letter alphabet $\{a,b,c\}$, and let L contain a length t word over Σ if it has the form $a^s b^s c^*$ for some $s < f(t)$. Since f is unbounded, L contains such words for arbitrarily large s. It thus follows from the Pumping Lemma, that L is not regular.

It suffices to define a β-bounded stopwatch automaton $\mathbb{A}$ such that that accepts a word of sufficiently large length t if and only if it belong to L. The states are $\mathit{start}, \mathit{accept}, q_a, q_b, q_c$ with λ-labels a, a, a, b, c, respectively. We use stopwatches x_a, y_a, x_b all with bound $f(t)$ and declare x_a, y_a active in q_a and *start*, and x_b active in q_b. There are transitions from *start* to q_a, from q_a to q_b, from q_b to q_c, and from q_c to *accept* – described next.

The transition from *start* to q_a has guard $x_a = 0$ and action $y_a := 1$. For sufficiently large t, the bound $f(t)$ of x_a is positive. Then any initial accepting computation (of $\mathbb{A}$ on a word of length t) spends 0 time in *start*, and thus starts $(\mathit{start}, [0,0,0]) \xrightarrow{0} (q_a, [0,1,0])$; we use a notation like $[1,2,3]$ to denote the assignment that maps x_a to 1, y_a to 2, and x_b to 3.

The transition from q_a to q_b has guard $x < y$ and trivial action. An initial accepting computation on a word of length t can stay in q_a for some time r reaching $(q_a, [r, r{+}1, 0])$ for $r < f(t)$, or reaching $[f(t), f(t), 0]$ for $r \geqslant f(t)$ due to the bound of x, y. In the latter case the transition to q_b is disabled and *accept* cannot be reached. Staying in q_a for any time $s < f(t)$ allows the transition to q_b.

The transition from q_b to q_c has guard $x_a = x_b$ and trivial action. The transition from q_c to *accept* has trivial guard and action. □

We can now proof that stopwatch automata are exponentially more succinct than finite automata as was expressed in Proposition 24.

Proof of Proposition 24. Consider the previous proof for the function f constantly k. Clearly, L is regular. The constant c in the Pumping Lemma can be taken to be the number of states of a finite automaton accepting L. It follows that every such automaton has at least k many states. The stopwatch automaton $\mathbb{A}$ accepts L and has size $O(\log k)$. Indeed, the size of a binary encoding of $\mathbb{A}$ is dominated by the bits required to write down the bound k of the 2 stopwatches. □

9 Discussion and a lower bound

We suggest the model-checking problem for stopwatch automata and finite words (over some finite alphabet) as an answer to our central question in Section 1.2, the quest for a model for algorithmic laws concerning activity sequences. This section discusses to

what extent this model meets the three desiderata listed in Section 1.2, and mentions some open ends for future work.

9.1 Summary

Expressivity Stopwatch automata are highly expressive, namely, by Theorem 23, equally expressive as MSO. In particular, [24] argued that Regulation 561 is expressible in MSO, so it is also expressible by stopwatch automata. In Section 8.2 we showed that a straightforward generalization of stopwatch automata can go even beyond MSO. Future research might show whether this is useful for modeling actual laws.

Example 36. Imagine an employee who can freely schedule his work and choose among various activities Σ to execute at any given time point. The employer favors an activity $a \in \Sigma$ and checks at random time-points that the employee used at least a third of his work-time on activity a since the previous check. The set of $w \in \Sigma^*$ with $\#_a(w) \geqslant |w|/3$ is not regular but is accepted by a simple β-bounded stopwatch automaton with one stopwatch x and bound $\beta(x,t) = \lceil t/3 \rceil$.

Naturality We stressed that expressivity alone is not sufficient, *natural* expressivity is required. This is an informal requirement, roughly, it means that the specification of a law should be readable, and in particular, not too large. In particular, as emphasized in Section 2.1, constants appearing in laws bounding durations of certain activities should not blow up the size of the formalization (like it is the case for LTL). We suggest that our expression of Regulation 561 by a stopwatch automaton is natural.

There is a possibility to use stopwatch automata as a *law maker*[13]: an interface that allows to specify laws in a formally rigorous way without assuming much mathematical education. It is envisionable to use graphical interfaces akin to the one provided by UPPAAL[14] to draw stopwatch automata. A discussion of this possibility as well as the concept of "readability" is outside the scope of this paper.

Tractability The main constraint of a model-checking problem as a formal model for algorithmic law is its computational tractability. In particular, the complexity of this problem should scale well with the constants appearing in the law. This asks for a fine-grained complexity analysis taking into account various aspects of a typical input, and, technically, calls for a complexity analysis in the framework of parameterized complexity theory. Theorem 30 gives a model-checker for stopwatch automata. Its worst case time complexity upper bound scales transparently with the involved constants, and, most importantly, the running time is not exponential in these constants. This overcomes a bottleneck of model-checkers designed in the context of system verification (see Section 10.2). Theorems 28 and 31 give similar algorithms for consistency checking and scheduling.

[13] Actually, a version of a product along these lines which goes by the name LawMaker© is currently being developed.

[14] https://uppaal.org/

9.2 Parameterized model-checking

We have an upper bound $O(\|\mathbb{A}\|^2 \cdot B_{\mathbb{A}} \cdot |w|)$ to the worst case runtime of our model-checker. The troubling factor is $B_{\mathbb{A}}$: the runtime grows fast with the stopwatch bounds of the automaton. Intuitively, these bounds stem from the constants mentioned by the law as duration constraints on activities. At least, this is the case for Regulation 561: we explicitly mentioned a tuple of 17 constants $\bar{t} = (t_0, \ldots, t_{16})$ which determine our automaton, specifically its bounds, guards and actions. To wit, $\bar{t}$ determines the bounds on stopwatches as follows:

x_{break}	x_{cd}	x_{day}	x_{dr}	x_{dd}	x_{week}	x_{ww}	x_{dw}, x'_{dw}	x_{wr}	x_{c1}, x_{c2}
t_{16}	$t_0 + 1$	$t_3 + 1$	t_4	$t_8 + 1$	$t_{10} + 1$	$t_{12} + 1$	$t_{11} + 1$	t_{14}	$t_{14} - t_{15}$

The other stopwatches have bounds independent of $\bar{t} \in \mathbb{N}^{17}$. For any choice of $\bar{t}$ we get an automaton $\mathbb{A}(\bar{t})$ that accepts exactly the words that represent activity sequences that are legal according to the variants of Regulations 561 obtained by changing these constants to $\bar{t}$. It is a matter of no concern to us that not all choices for $\bar{t}$ lead to meaningful laws. We are interested in how the runtime of our model-checker for Regulation 561 depends on these constants.

By Theorem 30 we obtain the following:

Corollary 37. *There is an algorithm that given $\bar{t} \in \mathbb{N}^{17}$ and a word w decides whether $w \in L(\mathbb{A}(\bar{t}))$ in time*

$$O\big(t_{16} \cdot t_0 \cdot t_3 \cdot t_4 \cdot t_8 \cdot t_{10} \cdot t_{12} \cdot t_{11}^2 \cdot t_{14} \cdot (t_{14} - t_{15})^2 \ \cdot \ |w|\big).$$

For the actual values of $\bar{t}$ in Regulation 561 the above product of the t_i's evaluates to the number

$$14600295444748092781363200000000000000.$$

This casts doubts whether the factor $B_{\mathbb{A}}$ in our worst-case running time $O(\|\mathbb{A}\|^2 \cdot B_{\mathbb{A}} \cdot |w|)$ should be regarded tractable. Can we somehow improve the runtime dependence from the constants?

For the sake of discussion, note that $B_{\mathbb{A}}$ is trivially bounded by $t_{\mathbb{A}}^{c_{\mathbb{A}}}$ where $c_{\mathbb{A}}$ is the number of stopwatches of $\mathbb{A}$ and $t_{\mathbb{A}}$ is the largest bound of some stopwatch of $\mathbb{A}$ (as in Section 1.3). Intuitively, $c_{\mathbb{A}}$ is "small" but $t_{\mathbb{A}}$ is not. In the spirit of parameterized complexity theory it is natural to ask whether the factor ($B_{\mathbb{A}}$ or) $t_{\mathbb{A}}^{c_{\mathbb{A}}}$ can be replaced by $f(c_{\mathbb{A}}) \cdot t_{\mathbb{A}}^{O(1)}$ for some computable function $f : \mathbb{N} \to \mathbb{N}$. We now formulate this question precisely in the framework of parameterized complexity theory.

The canonical parameterized version of our model-checking problem is

Input:	a stopwatch automaton $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ and $w \in \Sigma^*$.
Parameter:	$\|\mathbb{A}\|$.
Problem:	$w \in L(\mathbb{A})$?

Our model-checker of Theorem 30 witnesses that this problem is fixed-parameter tractable. Indeed, $\|\mathbb{A}\|^2 \cdot B_{\mathbb{A}} \leqslant f(\|\mathbb{A}\|)$ for some computable $f : \mathbb{N} \to \mathbb{N}$ because the circuits in $\mathbb{A}$ have size $\leq \log B_{\mathbb{A}}$. Intuitively, that $B_{\mathbb{A}}$ is bounded in terms of the parameter $\|\mathbb{A}\|$

means that the parameterized problem above models instances where $B_{\mathbb{A}}$ is "small", in particular β takes "small" values. But there are cases of interest where this is not true: the constant $t_{10} := 10080$ in Regulation 561 is not "small"[15]. In the situation of such an algorithmic law, the above parameterized problem is the wrong model.

A better model parameterizes a model-checking instance $(\mathbb{A}, w)$ by the size of $\mathbb{A}$ but discounts the stopwatch bounds. More precisely, consider the following parameterized problem:

p-SWA	
Input:	a stopwatch automaton $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ and $w \in \Sigma^*$.
Parameter:	$\lvert Q\rvert + \lvert\Sigma\rvert + \lvert X\rvert + \lvert\Delta\rvert$.
Problem:	$w \in L(\mathbb{A})$?

Note that the algorithm of Theorem 30 does not witness that this problem would be fixed-parameter tractable. We arrive at the precise question:

Is p-SWA fixed-parameter tractable?

9.3 A lower bound

In this section we prove that the answer to the above question is likely negative in the sense of Theorem 38 below. We would like to stress that the proof of the theorem is self-contained and in particular, no knowledge of the mentioned complexity classes is required.

Theorem 38. *p-SWA is not fixed-parameter tractable unless every problem in the W-hierarchy is fixed-parameter tractable.*

We refer to any of the monographs [27, 36, 28] for a definition of the *W-hierarchy* $\mathsf{W}[1] \subseteq \mathsf{W}[2] \subseteq \cdots$. As mentioned in Section 1.2, the central hardness hypothesis of parameterized complexity theory is that already the first level $\mathsf{W}[1]$ contains problems that are not fixed-parameter tractable. We thus consider Theorem 38 as strong evidence that the answer to our question is negative.

We prove Theorem 38 by a reduction from a parameterized version of the *Longest Common Subsequence Problem (LCS)*. This classical problem takes as inputs an alphabet Σ, finitely many words $w_0, \ldots, w_{k-1}$ over Σ and a natural number m. The problem is to decide whether the given words have a common subsequence of length m: such a subsequence is a length m word $a_0 \cdots a_{m-1}$ over Σ (the a_i are letters from Σ) that can be obtained from every $w_i, i < k$, by deleting some letters. In other words, for every $i < k$ there are $j^i_0 < \cdots < j^i_{m-1} < |w_i|$ such that for all $\ell < m$ the word w_i has letter a_ℓ at position j^i_ℓ. For example, both *bbaccb* or *bbaacb* are common subsequences of *abbaaccb* and *bbacccacbb*.

This problem received considerable attention in the literature and has several natural parameterized versions [9, 10, 8, 51]. We consider the following one:[16]

[15] Lowering the particular week constant t_{10} can be achieved by pre-processing the data and labelling each Monday 0:00 with a special label, but in general, constants are of substantial size.

[16] In [36] the notation p-LCS refers to a different parameterization of LCS.

p-LCS	
Input:	an alphabet Σ, words $w_0, \ldots, w_{k-1} \in \Sigma^*$ for some $k \in \mathbb{N}$, and $m \in \mathbb{N}$.
Parameter:	$k + \lvert\Sigma\rvert$.
Problem:	do $w_0, \ldots, w_{k-1}$ have a common subsequence of length m ?

The statement that p-LCS is fixed-parameter tractable means that it can be decided by an algorithm that on an instance $(\Sigma, w_0, \ldots, w_{k-1}, m)$ runs in time

$$f(k + |\Sigma|) \cdot (|w_0| + \cdots + |w_{k-1}|)^{O(1)}$$

for some computable function $f : \mathbb{N} \to \mathbb{N}$. The existence of such an algorithm is unlikely due to the following result:

Theorem 39 ([8])**.** p-LCS *is not fixed-parameter tractable unless every problem in the W-hierarchy is fixed-parameter tractable.*

Proof of Theorem 38: Let $(\Sigma, w_0, \ldots, w_{k-1}, m)$ be an instance of p-LCS, so Σ is an alphabet, $w_0, \ldots, w_{k-1} \in \Sigma^*$ and $m \in \mathbb{N}$. Let $w := w_0 \cdots w_{k-1}$ be the concatenation of the given words, and consider w^m, the concatenation of m copies of w. We construct a $P(\Sigma)$-labeled stopwatch automaton $\mathbb{A} = (Q, \Sigma, X, \lambda, \beta, \zeta, \Delta)$ that accepts w^m if and only if $w_0, \ldots, w_{k-1}$ have a common subsequence of length m.

An initial accepting computation of $\mathbb{A}$ on w^m proceeds in m *rounds*, each round reads a copy of w. In round $\ell < m$ the computation guesses a position within each of the words $w_0, \ldots, w_{k-1}$ copied within w, and ensures they all carry the same letter. These positions are stored in registers (i.e., nowhere active stopwatches) $x_0, \ldots, x_{k-1}$ with bounds $|w_0| + 1, \ldots, |w_{k-1}| + 1$, respectively. Our intention is that the value of x_i after round $\ell < m$ equals the position j_ℓ^i in the definition of a common subsequence.

Our intention is that an initial accepting computation in round $\ell < m$ cycles though k many *guess parts* of the automaton. Within guess part 0, the computation reads w_0 (within copy ℓ of w in w^m), within guess part 1 the computation reads w_1 and so on. The states of $\mathbb{A}$ are the states of the guess parts plus an an additional state *accept*. Each guess part consists of a copy of the states *start*, *end*, and *guess*(a) for $a \in \Sigma$. The λ-labels of *start* and *end* are Σ, the λ-label of *guess*(a) is $\{a\}$. The start state of $\mathbb{A}$ is *start* in guess part 0.

We intend that the computation in guess part $i < k$ spends some time $t < |w_i|$ in *start*, then spends exactly one time unit in some state *guess*(a), and then spends time $|w_i| - t$ in *end* before switching to the next guess part. The position guessed is t and stored as the value of x_i. Writing momentarily $w_i = a_0 a_1 \cdots a_{|w_i|-1}$ the computation reads the (possibly empty) word $a_0 \cdots a_{t-1}$ in state *start*, then reads a_t in state *guess*(a_t), and then reads the (possibly empty) word $a_{t+1} \cdots a_{|w_i|-1}$ in state *end*.

We enforce this behavior as follows. There are transitions from *start* (in guess part i) to *guess*(a) for every $a \in \Sigma$, and for every $a \in \Sigma$ from *guess*(a) to *end*. We use a stopwatch y_i with bound $|w_i| + 1$ active in all states of guess part i and a stopwatch z with bound 2 active in the states *guess*(a), $a \in \Sigma$, of any guess part. It will be clear that initial accepting computations enter guess part i with both y_i and z having value 0. The transitions from *start* to *guess*(a), $a \in \Sigma$, have guard checking $x_i < y_i < |w_i|$ and action setting $x_i := y_i$. The transitions from *guess*(a), $a \in \Sigma$, to *end* have guard checking

$z = 1$ and action setting $z := 0$. The state *end* in guess part $i < k-1$ has a transition to *start* in guess part $i+1$; for $i = k-1$ this transition is to *start* in guess part 0. These transitions have guard checking $y_i = |w_i|$ and action setting $y_i := 0$.

Observe that the computation spends time $|w_i|$ in guess part $i < k$ and increases the value of x_i. Hence the values of x_i after each round form an increasing sequence of positions $< |w_i|$. We have to ensure that the values of $x_0, \dots, x_{k-1}$ after a round are positions in the words $w_0, \dots, w_{k-1}$, respectively, that carry the same letter. Write $\Sigma = \{a_0, \dots, a_{|\Sigma|-1}\}$. We use a register $\tilde{x}$ with bound $|\Sigma| - 1$. In guess part 0, the action of the transition from $guess(a_j)$ to *end* also sets $\tilde{x} := j$. In the guess parts $i < k$ for $i \neq 0$, the guards of the transitions from *start* to $guess(a_j)$ check that $\tilde{x} = j$.

We count rounds using a register $\tilde{y}$ with bound m. We let the action of the transition from *end* in guess part $k-1$ to *start* in guess part 0 set $\tilde{y} := \tilde{y} + 1$. From copy 0 of *start* there is a transition to *accept*. The guard of this transition checks $\tilde{y} = m$.

This completes the construction of $\mathbb{A}$.

To prove the theorem, assume p-SWA is fixed-parameter tractable, i.e., there is an algorithm deciding p-SWA that on an instance $(\mathbb{A}, w)$ runs in time $f(k') \cdot |w|^{O(1)}$ where k' is the parameter of the instance, and $f : \mathbb{N} \to \mathbb{N}$ is a nondecreasing computable function. By Theorem 39 is suffices to show that p-LCS is fixed-parameter tractable.

Given an instance $(\Sigma, w_0, \dots, w_{k-1}, m)$ of p-LCS answer "no" if $m > |w_0|$. Otherwise compute the automaton $\mathbb{A}$ as above and then compute an equivalent stopwatch automaton $\mathbb{A}'$ as in the construction behind Proposition 25. It is clear that $(\mathbb{A}', w^m)$ is computable from $(\Sigma, w_0, \dots, w_{k-1}, m)$ in polynomial time (since $m \leqslant |w_0|$). Then $(\mathbb{A}', w^m)$ is a "yes"-instance of p-SWA if and only if $(\Sigma, w_0, \dots, w_{k-1}, m)$ is a "yes"-instance of p-LCS. Hence to decide p-LCS it suffices to run the algorithm for p-SWA on $(\mathbb{A}', w^m)$. This takes time $f(k') \cdot |w^m|^{O(1)}$ where k' is the parameter of $(\mathbb{A}', w^m)$. By construction, it is clear that $k' \leqslant g(k + |\Sigma|)$ for some computable $g : \mathbb{N} \to \mathbb{N}$ (in fact, $k' \leqslant (k + |\Sigma|)^{O(1)}$). Since $m \leqslant |w_0|$, the time $f(k') \cdot |w^m|^{O(1)}$ is bounded by $f(g(k + |\Sigma|)) \cdot (|w_0| + \cdots + |w_{k-1}|)^{O(1)}$. Thus, p-LCS is fixed-parameter tractable. □

Recall, p-SWA is meant to formalize the computational problem to be solved by general purpose model-checkers in algorithmic law. Being general purpose, the set of activities Σ should be part of the input, it varies with the laws to be modeled. Nevertheless one might ask whether the hardness result in Theorem 38 might be sidestepped by restricting attention to some fixed alphabet Σ.

This is unlikely to be the case. Let p-SWA$(\{0,1\})$ denote the restriction of p-SWA to instances with $\Sigma = \{0,1\}$. We have the following variant of Theorem 38:

Theorem 40. *p-SWA$(\{0,1\})$ is not fixed-parameter tractable unless* FPT = W[1].

Proof. Note that the reduction $(\Sigma, w_0, \dots, w_{k-1}, m) \mapsto (\mathbb{A}', w^m)$ (for $m \leqslant |w_0|$) in the proof above constructs an automaton $\mathbb{A}'$ over the same alphabet Σ. It is thus a reduction from the restriction of p-LCS to instances with $\Sigma = \{0,1\}$ to p-SWA$(\{0,1\})$. Now, [51] showed that this restriction is W[1]-hard. □

In particular, under the assumption that FPT ≠ W[1], we know that MC(Σ^*, SWA) cannot be decided in time $\big(|\mathbb{A}| \cdot f(x) \cdot t \cdot |w|\big)^{O(1)}$ for any computable function $f : \mathbb{N} \to \mathbb{N}$.

10 Bounded Stopwatch Automata, a motivation

As mentioned, the problem with the running time $f(\|\varphi\|)\cdot|w|$ of Büchi's model-checker is that the parameter dependence $f(k)$ grows extremely fast: it is non-elementary in the sense that it cannot be bounded by $2^{2^{\cdot^{\cdot^{2^k}}}}$ for any fixed height tower of 2's.

This is due to the fact that in general the size $\|\mathbb{B}_\varphi\|$ of (a reasonable binary encoding of) $\mathbb{B}_\varphi$ is non-elementary in $\|\varphi\|$. Under suitable hardness hypotheses this non-elementary parameter dependence cannot be avoided, not even when restricting to first-order logic FO [38].

This motivates the quest for fragments of MSO or less succinct variants thereof that allow a tamer parameter dependence. In system verification, LTL has been proposed: an LTL formula of size k can be translated to a size $2^{O(k)}$ Büchi automaton [58] or a size $O(k)$ alternating automaton [56].

The model-checking problem asks given a system modeled by a finite automaton $\mathbb{A}$ whether all (finite or infinite) words accepted by the automaton satisfy the given LTL-sentence φ. The model-checker decides emptiness of a suitable product automaton accepting $L(\mathbb{A})\cap L(\mathbb{B}_{\neg\varphi})$ and takes time $2^{O(\|\varphi\|)}\cdot\|\mathbb{A}\|$. This is the dominant approach to model-checking in system verification.

10.1 Linear Temporal Logic and Regulation 561

The paper [24] formalizes part of Regulation 561 in LTL. In part, these formalizations rely on Kamp's theorem (cf. [53]) stating that LTL and FO have the same expressive power over Σ^*. But the translation of an FO-sentence to an LTL-sentence can involve a non-elementary blow-up in size. Indeed, [24] proves lower bounds on the length of LTL-sentences expressing parts of Regulation 561. Very large sentences are not natural and lead to prohibitive model-checking times.

Example 41. To illustrate the point, consider the following law in Regulation 561:

> Article 6.2: The weekly driving time shall not exceed 56 hours...

Restrict attention to words representing one week of activities, i.e., words of length $7\cdot24\cdot60$ over the alphabet $\Sigma=\{d,w,r\}$. A straightforward formalization of Article 6.2 in LTL is (using $d\in\Sigma$ as a propositional variable) the huge disjunction of

$$\bigwedge_{d\leqslant D}\Big(\bigwedge_{r_d\leqslant i<\ell_{d+1}}\bigcirc^i\neg d\wedge\bigwedge_{\ell_d\leqslant i<r_d}\bigcirc^i d\Big)$$

for all $D\leqslant 7\cdot24\cdot60$ and all $r_0:=0\leqslant\ell_1<r_1<\cdots<\ell_D<r_D<\ell_{D+1}:=7\cdot24\cdot60$ with $\sum_{1\leqslant j\leqslant D}(r_j-\ell_j)\leqslant 56\cdot60$. These are $>\binom{7\cdot24\cdot60}{56\cdot60}>10^{2784}$ many disjuncts.

To conclude, MSO gives the wrong model because it does not allow sufficiently fast model-checkers, and LTL is the wrong model because it is not sufficiently (expressive nor) succinct, hence not natural. It can be expected that, like Regulation 561, many algorithmic laws concerning sequences of activities state lower and upper bounds on the duration of certain activities or types of activities. The constants used to state these bounds are not necessarily small, and this is an important aspect to take into account when analyzing the model-checking complexity.

10.2 Regulation 561 and timed modal logics

The above motivates to look at models with built-in timing constraints: "In practice one would want to use 'sugared' versions of LTL, such as metric temporal logic (MTL; [48]) which allow for expressions such as $\bigcirc^{n+1}$ to be represented succinctly"[24]. MTL has modalities like $\Diamond_{[5,8]}\varphi$ expressing that φ holds within 5 and 8 time units from now. Such logics have been extensively investigated in system verification. We give a brief survey, guided by our central question to model Regulation 561.

For Regulation 561, cases are tachograph recordings which, formally, are *timed words* $(a_0, t_0)\ (a_1, t_1)\cdots$ where the a_i are letters and the t_i an increasing sequence of time-points; intuitively, activity a_0 is observed until time point t_0, then a_1 until t_1, and so on. Alur and Dill [4] extended finite automata to *timed automata* that accept sets of timed words – see [12] for a survey. Roughly speaking, computations of such automata happen in time and are governed by finitely many clocks: transitions from one state to another are enabled or blocked depending on the clock values, and transitions can reset some clocks (to value 0). Alur and Dill [4] proved that timed automata have decidable emptiness, thus enabling the dominant model-checking paradigm.

Consequently, a wealth of timed temporal logics have been investigated – [41, 13] are surveys. The following table lists some of the most important choices when defining a timed temporal logic:

semantics		*time*		*clocks*
finite words	signal-based	continuous $\mathbb{R}_{\geqslant 0}$	branching	internal
infinite words	event-based	discrete $\mathbb{N}$	linear	external

Let us briefly comment on some of the choices put forward by this table. A subtle choice is between signal- or event-based semantics. It means, roughly and respectively, that the modalities quantify over all time-points or only over the t_i appearing in the timed word; MTL is known to be less expressive in the latter semantics over finite timed words [29]. A crucial choice is between time $\mathbb{N}$ or $\mathbb{R}_{\geqslant 0}$. Internal clocks appear only on the side of the automata, external clocks appear in sentences which reason about their values. We briefly survey the most important results.

An early success [2] concerns the infinite word signal-based branching continuous time logic TCTL (timed computation tree logic): over (systems modeled by) timed automata it admits a model-checker with runtime $t^{O(c)} \cdot k \cdot n$ where n is the automaton size, k the size of the input sentence, c the number of clocks, and t is the largest time constant appearing in the input. The paper [43] extends this allowing external clocks. However, continuous branching time is semantical and syntactical overkill for Regulation 561.

For linear continuous time we find MTL and TPTL (timed propositional temporal logic), a more expressive [15] extension with external clocks. Since model-checking is undecidable for these logics [6, 5], fragments have been investigated. Surprisingly, [48] found an FPT model-checker for MTL over event-based finite words via a translation to alternating automata with one clock, albeit with intolerable parameter dependence (non-primitive recursive).

Metric interval temporal logic (MITL) [5] is the fragment of MTL disallowing singular time constraints as, e.g., in $\Diamond_{[1,1]}\varphi$. The papers [34, 45] give an elegant translation

of MITL to timed automata and thereby a model-checker with runtime[17] $2^{O(t \cdot k)} \cdot n$. Over discrete time, [6] adapts the mentioned translation of LTL to Büchi automata and gives a model-checker for TPTL with runtime $2^{O(t^c \cdot k)} \cdot n$.

As said, from the perspective of algorithmic laws, t is not typically small and runtimes exponential in $t = 56h = 3360$ *min* are prohibitive. Tamer runtimes with t moved out of the exponent have been found for a certain natural MITL-fragment $\mathsf{MITL}_{0,\infty}$ both over discrete and continuous time – see [41, 5].

However, "standard real-time temporal logics [...] do not allow us to constrain the accumulated satisfaction time of state predicates" [3, p.414]. It seems that this is just what is required to formalize the mentioned Article 6 (2).

There are various attempts to empower the logics with some reasoning about durations. *Stopwatch automata* [26] are timed automata that can not only reset clocks but also stop and activate them. However, emptiness is undecidable already for a single stopwatch [42]. Positive results are obtained in [3] for *observer stopwatches*, i.e., roughly, stopwatches not used to govern the automaton's transitions. On the logical side, [11] and [17] study fragments and restrictions for TCTL with (observer) stopwatches.

On another strand, [20] puts forward the *calculus of durations*, but already tiny fragments turn out undecidable [19]. For discrete time, [40] gives an fpt model-checker via a translation to finite automata. For continuous time, [37] obtains fpt results under certain reasonable restrictions of the semantics. A drawback is that these fpt results have non-elementary parameter dependence.

To conclude, the extensive research on "'sugared' versions" of LTL in system verification does not reveil a good answer to our central question for a model-checking problem modeling algorithmic laws concerning activity sequences. In particular, some model-checkers in system verification, like the one for TPTL over discrete time, are too slow in that they do not scale well with time constants mentioned in the law.

10.3 The perspective from algorithmic law

The new perspective on model-checking from algorithmic law seems orthogonal to the dominant perspectives from database theory and system verification in the sense that it seems to guide incomparable research directions.

In database theory there is special interest in model-checking problems for a rich class $\mathcal{K}$, formalizing a large class of databases, and possibly weak logics L formalizing simple basic queries. In algorithmic law (concerning activity sequences) it is the other way around, focussing on $\mathcal{K} = \Sigma^*$.

System verification gives special interest to infinite words and continuous time (cf. e.g. [2]) while algorithmic law focusses on finite words and discrete time. Most importantly, system verification focusses on structures specifying *sets* of words: its model-checking problem corresponds to (a generalization of) the consistency problem in algorithmic law. In algorithmic law the consistency problem is secondary, the main interest is in evaluating sentences over single words.

[17] In fact, t can be replaced by a typically smaller number, called the *resolution* of the formula – see [34].

Finally, the canonical parameterization of a model-checking problem takes the size $\|\varphi\|$ of the input sentence φ as the parameter. Intuitively, then parameterized complexity analysis focusses attention on inputs of the problem where $\|\varphi\|$ is relatively small. Due to large constants on time constraints appearing in the law to be formalized this parameterization does not seem to result in a faithful model of algorithmic law. We shall come back to this point in Section 9.2.

Compared to system verification this shift of attention in algorithmic law opens the possibility to use more expressive logics while retaining tractability of the resulting model. In particular, complexity can significantly drop via the shift from continuous time, infinite words and consistency-checking, to discrete time, finite words and model-checking. While discrete time is well investigated in system verification, it has been noted that both finite words and model-checking have been neglected – see [35] and [46], respectively. To make the point: over finite words consistency-checking LTL is PSPACE-complete but model-checking is PTIME, even for the more succinct extensions of LTL with past- and now-modalities [46], or even finite variable FO [57].[18]

We have taken advantage of this possibility to use more expressive logics and have suggested (a version of) discrete time stopwatch automata SWA as an answer to our central question, that is, we proposed $\mathrm{MC}(\Sigma^*, \mathsf{SWA})$ as a model for algorithmic laws concerning sequences of activities.

Acknowledgements

We thank Raül Espejo Boix for a critical reading of Section 5. Part of this work has been done while the first author has been employed by Formal Vindications S.L. The second author leads a covenant between the University of Barcelona and Formal Vindications S.L. The second author received funding under the following schemes: ICREA Acadèmia, projects PID2020-115774RB-I00 and PID2019-107667GB-I00 of the Spanish Ministry of Science and Innovation, 2022 DI 051, Generalitat de Catalunya, Departament d'Empresa i Coneixement and 2017 SGR 270 of the AGAUR.

References

[1] Ana de Almeida Borges, Mireia González Bedmar, Juan Conejero Rodríguez, Eduardo Hermo Reyes, Joaquim Casals Buñuel, and Joost J. Joosten. FV Time: a formally verified Coq library. arXiv:2209.14227 [cs.SE], 2022.

[2] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[3] Rajeev Alur, Costas Courcoubetis, and Thomas A. Henzinger. Computing accumulated delays in real-time systems. *Formal Methods in System Design*, 11(2):137–155, 1997.

[18] 3-variable (2-variable) FO has the same expressive power as (unary) LTL over finite words but is much more succinct [32]. [23] gives a fine calibration of the parameterized complexity of finite variable FO.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[6] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[8] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, Michael T. Hallett, and Harold T. Wareham. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences*, 11(1):49–57, 1995.

[9] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Harold T. Wareham. The parameterized complexity of sequence alignment and consensus. In Maxime Crochemore and Dan Gusfield, editors, *Combinatorial Pattern Matching, 5th Annual Symposium, CPM 94, Asilomar, California, USA, June 5-8, 1994, Proceedings*, volume 807 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 1994.

[10] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Harold T. Wareham. The parameterized complexity of sequence alignment and consensus. *Theoretical Computer Science*, 147(1&2):31–54, 1995.

[11] Ahmed Bouajjani, Rachid Echahed, and Joseph Sifakis. On model checking for real-time properties with durations. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*, pages 147–159. IEEE Computer Society, 1993.

[12] Patricia Bouyer. An introduction to timed automata. In *Actes de la 4ème École Temps-Réel (ETR'05)*, pages 111–123, Nancy, France, September 2005.

[13] Patricia Bouyer. Model-checking timed temporal logics. *Electronic Notes in Theoretical Computer Science*, 231:323–341, 2009.

[14] Patricia Bouyer and Fabrice Chevalier. On conciseness of extensions of timed automata. *Journal of Automata, Languages and Combinatorics*, 10(4):393–405, 2005.

[15] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of TPTL and MTL. *Inf. Comput.*, 208(2):97–116, 2010.

[16] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Updatable timed automata. *Theor. Comput. Sci.*, 321(2-3):291–345, 2004.

[17] Thomas Brihaye, Véronique Bruyère, and Jean-François Raskin. On model-checking timed automata with stopwatch observers. *Information and Computation*, 204(3):408–433, 2006.

[18] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, page 77–90, New York, NY, USA, 1977. Association for Computing Machinery.

[19] Zhou Chaochen, Michael R. Hansen, and Peter Sestoft. Decidability and undecidability results for duration calculus. In Patrice Enjalbert, Alain Finkel, and Klaus W. Wagner, editors, *STACS 93, 10th Annual Symposium on Theoretical Aspects of Computer Science, Würzburg, Germany, February 25-27, 1993, Proceedings*, volume 665 of *Lecture Notes in Computer Science*, pages 58–68. Springer, 1993.

[20] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[21] Hubie Chen and Moritz Müller. The fine classification of conjunctive queries and parameterized logarithmic space. *ACM Transactions on Computation Theory*, 7(2):7:1–7:27, 2015.

[22] Hubie Chen and Moritz Müller. One hierarchy spawns another: Graph deconstructions and the complexity classification of conjunctive queries. *ACM Transaction on Computational Logic*, 18(4):29:1–29:37, 2017.

[23] Yijia Chen, Michael Elberfeld, and Moritz Müller. The parameterized space complexity of model-checking bounded variable first-order logic. *Logical Methods Computer Science*, 15(3), 2019.

[24] Ana de Almeida Borges, Juan José Conejero Rodríguez, David Fernández-Duque, Mireia González Bedmar, and Joost J. Joosten. To drive or not to drive: A logical and computational analysis of European transport regulations. *Information and Computation*, 280:104636, 2021.

[25] J. del Castillo Tierz. When the laws of logic meet the logic of laws. Master's thesis, Master of Pure and Applied Logic, 2018.

[26] Catalin Dima. Timed shuffle expressions. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2005.

[27] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[28] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013.

[29] Deepak D'Souza and Pavithra Prabhakar. On the expressiveness of mtl in the pointwise and continuous semantics. *International Journal on Software Tools for Technology Transfer*, 9(1):1–4, 2007.

[30] Wolfgang Thomas Erich Grädel and Thomas Wilke. *Automata Logics, and Infinite Games*. Lecture Notes in Computer Science. Springer, 2002.

[31] G. Errezil Alberdi. Industrial Software Homologation: Theory and case study. Industrial Software Homologation: Theory and case study Analysis of the European tachograph technology with EU transport Regulations 3821/85, 799/2016, and 561/06 and their consequences for Europeans citizens. Technical report, Formal Vindications S.L., 2019.

[32] Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke. First-order logic with two variables and unary temporal logic. *Information and Computation*, 179(2):279–295, 2002.

[33] David Fernández-Duque, Mireia González Bedmar, Daniel Sousa, Joost J. Joosten, and Guillermo Errezil Alberdi. To drive or not to drive: A formal analysis of Requirements (51) and (52) from Regulation (EU) 2016/799. In *Personalidades jurídicas difusas y artificiales*, TransJus Working Papers, pages 159–171. Institut de Recerca TransJus, 2019.

[34] Thomas Ferrère, Oded Maler, Dejan Nickovic, and Amir Pnueli. From real-time logic to timed automata. *Journal of the ACM*, 66(3):19:1–19:31, 2019.

[35] Valeria Fionda and Gianluigi Greco. LTL on finite and process traces: Complexity results and a practical reasoner. *Journal of Artificial Intelligence Research*, 63:557–623, 2018.

[36] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[37] Martin Fränzle. Model-checking dense-time duration calculus. *Formal Aspects Comput.*, 16(2):121–139, 2004.

[38] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130(1-3):3–31, 2004.

[39] Martin Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM*, 54(1):1:1–1:24, 2007.

[40] Michael R. Hansen. Model-checking discrete duration calculus. *Formal Aspects Comput.*, 6(6A):826–845, 1994.

[41] Thomas A. Henzinger. It's about time: Real-time logics reviewed. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 439–454. Springer, 1998.

[42] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

[43] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[44] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Temporal logic with forgettable past. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 383–392. IEEE Computer Society, 2002.

[45] Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to timed automata. In Eugene Asarin and Patricia Bouyer, editors, *Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006, Paris, France,*

September 25-27, 2006, Proceedings, volume 4202 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2006.

[46] Nicolas Markey and Philippe Schnoebelen. Model checking a path. In Roberto M. Amadio and Denis Lugiez, editors, *CONCUR 2003 - Concurrency Theory, 14th International Conference, Marseille, France, September 3-5, 2003, Proceedings*, volume 2761 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2003.

[47] Thomas Schwentick Nicole Schweikardt and Luc Segoufin. Database theory: query languages. In Mikhail J. Atallah and Marina Blanton, editors, *Algorithms and theory of computation handbook: special topics and techniques*. Chapman and Hall/CRC, 2010.

[48] Joël Ouaknine and James Worrell. On the decidability and complexity of metric temporal logic over finite words. *Logical Methods in Computer Science*, 3(1), 2007.

[49] Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. *Journal of Computer and System Sciences*, 58(3):407–427, 1999.

[50] European Parliament and Council of the European Union. Regulation (ec) no 561/2006 of the European Parliament and of the Council of 15 march 2006 on the harmonisation of certain social legislation relating to road transport. *Official Journal of the European Union*, 2006.

[51] Krzysztof Pietrzak. On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. *Journal of Computer and System Sciences*, 67(4):757–771, 2003.

[52] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[53] Alexander Rabinovich. A proof of Kamp's theorem. *Logical Methods Computer Science*, 10(1), 2014.

[54] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.

[55] Wolfgang Thomas. Languages, automata, and logic. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 389–455. Springer, 1997.

[56] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1995.

[57] Moshe Y. Vardi. On the complexity of bounded-variable queries. In Mihalis Yannakakis and Serge Abiteboul, editors, *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, pages 266–276. ACM Press, 1995.

[58] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.