Provable Performance Guarantees for Byzantine Systems

Jake Ginesin

1 Introduction

Distributed protocols are the lynchpin of the modern internet, underpinning every internet service. This has, in turn, motivated a massive amount of research ensuring the security, reliability, and performance of distributed protocols. One popular approach in the literature has been employing formal methods, the use of mathematically techniques to reason about programs, to rigorously *prove* distributed protocols maintain properties of interest. Formal methods, as a field, is characterized by the continual drive to specify and reason about systems that are as expressive as possible and as representative of real-world systems as possible. To this end, there exists a long line of work building mathematical tools for reasoning about ever more expressive and powerful distributed systems. One such gap in the current state of the field is the formal verification of *performance properties of byzantine systems*. That is, if a distributed system has a certain misbehaving nodes, how are performance properties affected? What guarantees for *global performance* can we *prove* given not all nodes of a distributed system behave correctly? This is a fundamental, historical problem in distributed systems that has been studied at a surface level, but has not been formally verified or given proper mathematical foundations. In particular, highly influential work presenting arguments for performance guarantees on byzantine replication and byzantine routing protocols have not been verified nor been given rigorous mathematical foundations [1, 2]. In this work, we seek to do so. We make the following contributions:

- Building on previous literature, we define a program logic for specifying and reasoning about both byzantine and performance guarantees simultaneously.
- Using our program logic, we specify performance guarantees for simple byzantine systems

- We mechanize our results in Dafny, a language for formally specifying and reasoning about programs.

2 Background & Related Works

This work builds on a long of historical work specifying computer systems using *explicit state* models. That is, systems and computation is specified using *state-transition* systems, or *automata*. While other methods exist for reasoning about computation systems, such as type systems or process calculi, in this work we focus on explicit state models for specifying systems.

Regular Automata. At the foundation of this work is the classical set of work on automata theory, complexity, and computability. For modeling systems, *explicit-state* models of computation have long been used as *model* of real-world systems for their explicit computation traces, simple computability and complexity results, and ease of implementation [3].

Definition 2.1 (Deterministic Finite Automaton)**:** A Deterministic Finite Automaton (DFA) is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states.
- Σ is a finite input alphabet.
- $\delta : Q \times \Sigma \to Q$ is the transition function.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of accepting (or final) states.

A DFA M accepts an input string $w = w_1 w_2 \dots w_n$ where each $w_i \in \Sigma$, if there exists a sequence of states $r_0, r_1, \dots, r_n \in Q$ such that:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $0 \leq i < n$,
3. $r_n \in F$.

Regular languages have well-studied properties, including closure under intersection, union, complement, and concatenation, as well as decidability (and complexity results) for language containment, language disjointness, emptiness, universality, and membership. Many of the additional automata we will discuss inherit variants of these properties [3].

I/O Automata. To model interactive and distributed systems, *Input/Output Automata* (I/O Automata) offer a structured approach by distinguishing between input, output, and internal actions [4]. This distinction makes I/O Automata particularly well-suited for describing the behavior of asynchronous distributed protocols, where components interact via inputs from the environment and produce outputs to other components.

Definition 2.2 (I/O Automaton)**:** An *Input/Output Automaton* (I/O Automaton) is a tuple $A = (S, S_0, \Sigma, \delta)$, where:

- S is a (possibly infinite) set of states,
- $S_0 \subseteq S$ is a non-empty set of initial states,
- $\Sigma = \Sigma^{in} \cup \Sigma^{out} \cup \Sigma^{int}$ is the set of actions, partitioned into input, output, and internal actions, with all three disjoint,
- $\delta \subseteq S \times \Sigma \times S$ is the transition relation, where $(s, a, s') \in \delta$ denotes a possible transition from state s to state s' on action a.

An I/O automaton is required to be input-enabled: for every input action $a \in \Sigma^{in}$ and every state $s \in S$, there exists some $s' \in S$

In general, I/O automata inherit the decidability, complexity, and closure guarantees from regular automata. For more details, see [4].

Büchi Automata. Büchi Automata are simply regular automata that accept infinite traces. That is, they are defined as follows:

Definition 2.3 (Büchi Automata)**:** A Büchi Automata is a tuple $B = (Q, \Sigma, \delta, Q_0, F)$ where:

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation,
- $Q_0 \subseteq Q$ is a set of initial states,
- $F \subseteq Q$ is a set of accepting states.

A run of a Büchi Automata is an infinite sequence of states $q_0, q_1, q_2, \ldots$ such that $q_0 \in Q_0$ and $(q_i, a, q_{i+1}) \in \delta$ for some $a \in \Sigma$ at each step i. The run is considered accepting if it visits states in F infinitely often.

Büchi Automata have a long history of being employed in-practice for verifying real-world programs. This is in-part due to the correspondence between Büchi Automata and linear temporal logic (LTL) [5, 6], a program logic expressive enough to reason about safety and liveness for finite-state systems. This correspondence is highly refined and efficient in practice — highly successful model checking tools such as SPIN, which critically rely upon the tight correspondence between LTL and the Büchi Automata [7], have existed for over 40 years, ultimately earning the 2002 ACM Software Systems award and spawning its own dedicated formal methods conference currently in its 32nd year[1]. In general, classical automata decision problems for Büchi Automata are PSPACE-complete [8]. Notably, the variant of Büchi Automata employed by SPIN (among other model checkers) is extended to support I/O, which is in-turn allows for more natural modeling of FIFO channels and classical distributed systems algorithms [9].

Timed Automata. Informally, a timed automaton is a finite-state machine extended with real-valued clocks, allowing for transitions to depend not only on input symbols but also on *timing* constraints over these clocks. This structure allows for greater ease in expressing *real-time* systems. Formally, timed automata are defined as such:

Definition 2.4 (Timed Automaton)**:** A Timed Automaton is a tuple $A = (S, \Sigma, C, s_0, E, F)$ where:

- S is a finite set of locations (or states),
- Σ is a finite input alphabet,
- C is a finite set of clocks,
- $s_0 \in S$ is the initial location,
- $E \subseteq S \times \Sigma \times \text{Guards}(C) \times 2^C \times S$ is the set of transitions, where each transition is labeled with an input symbol from Σ, a guard $g \in \text{Guards}(C)$, which is a conjunction of constraints of the form $x \bowtie k$ or $x - y \bowtie k$, with $x, y \in C, \bowtie \in \{<, \leq, =, \geq, >\}$, and $k \in \mathbb{Q}$, a set of clocks to reset, and target state.
- $F \subseteq S$ is the set of accepting states.

A run of a timed automaton includes a valuation of clocks and is accepting if it eventually reaches a location in F under valid timing constraints.

[1] https://spin-web.github.io/SPIN2025/

Like Büchi Automata, timed automata have a suite of decidability theorems [10, 11], tools for reasoning about them [12], and extensions to support I/O [13].

Parametric Timed Automata. Taking timed automata a step further, rather than constraining guards for a given transition with a pre-chosen variable, we *parameterize* the guard parameter so timing guarantees can be reasoned about in greater generality. Although generalizing timed transitions in this way allows much greater expressive power, we lose almost all reasonable automation that we would expect with a finite-state representation of our system, as many classical decision problems become undecidable [14].

Definition 2.5 (Parametric Timed Automaton)**:** A *parametric timed automaton* (PTA) is a tuple

$$\mathcal{A} = (S, \Sigma, C, P, S_0, F, E)$$

where:

- S is a finite set of locations,
- Σ is a finite input alphabet,
- C is a finite set of clocks,
- P is a finite set of parameters,
- $S_0 \subseteq S$ is a non-empty set of initial locations,
- $F \subseteq S$ is a set of accepting locations,
- $E \subseteq S \times \Sigma \times \mathit{Guard}(C, P) \times 2^C \times S$ is a finite set of edges,

and each edge $(s, a, g, \lambda, s') \in E$ represents a transition from s to s' under action $a \in \Sigma$, conditioned on guard $g \in \mathit{Guard}(C, P)$, with clocks in $\lambda \subseteq C$ reset upon transition.

A guard $g \in \mathit{Guard}(C, P)$ is a conjunction of atomic constraints of the form $x \bowtie k$ or $x - y \bowtie k$, where $x, y \in C$, $\bowtie \in \{<, \leq, =, \geq, >\}$, and $k \in \mathbb{Q} \cup P$.

3 Program Logic Design

In this section, we design our program logic intended to specify performance guarantees for byzantine systems. In designing our logic, we pick and choose features from the aforementioned automata to fit our specific verification needs. In order to allow for more generalized reasoning across different automata about upper bounds for timing guarantees, we adopt parametric timing. In order to reason

effectively about communicating processes, we adopt I/O. In order to reason about safety and liveness properties, we adopt infinite traces (i.e. Büchi Automata). Cumulatively, we define the *parametric timed I/O Büchi automaton* (PTIOBA for short):

Definition 3.1 (Parametric Timed I/O Büchi Automaton)**:** A *parametric timed I/O Büchi automaton* (PTIOBA) is a tuple

$$\mathcal{A} \;=\; \big(\Sigma^{\text{in}}, \Sigma^{\text{out}}, S, S_0, F, C, P, E\big)$$

where

1. Σ^{in} and Σ^{out} are disjoint finite alphabets of *input* and *output* actions; set $\Sigma = \Sigma^{\text{in}} \cup \Sigma^{\text{out}}$.
2. S is a finite set of locations with $S_0 \subseteq S$ non-empty initial and $F \subseteq S$ Büchi accepting locations.
3. C is a finite set of real-valued clocks.
4. P is a finite set of unknown real constants called *parameters*.
5. $E \subseteq S \times \Sigma \times \mathit{Guard}(C, P) \times 2^C \times S$ is a finite edge relation. A guard $g \in \mathit{Guard}(C, P)$ is a conjunction of (strict or non-strict) atomic constraints of the form $x \bowtie k$ or $x - y \bowtie k$ with $x, y \in C$, comparator $\bowtie \in \{<, \leq, =, \geq, >\}$, and k either an integer constant or a parameter from P.

An edge $(s, \sigma, g, \lambda, s')$ is written $s \xrightarrow{\sigma, g, \lambda} s'$ and means: when action σ is observed and the current clock valuation satisfies guard g, the automaton may move from s to s' resetting all clocks in $\lambda \subseteq C$.

Definition 3.2 (Clock Valuation)**:** A *clock valuation* is a function $\nu : C \to \mathbb{R}_{\geq 0}$. For $\delta \in \mathbb{R}_{\geq 0}$ we write $\nu + \delta$ for the valuation incrementing every clock by δ, and $\nu[\lambda := 0]$ for the valuation that additionally resets every $x \in \lambda$ to 0.

Definition 3.3 (Timed I/O Trace and Run)**:** Fix a valuation $\pi : P \to \mathbb{R}_{>0}$ of the parameters. A *timed I/O trace* is an infinite sequence

$$(\sigma_0, \tau_0)(\sigma_1, \tau_1) \cdots \in (\Sigma \times \mathbb{R}_{\geq 0})^\omega$$

with non-decreasing absolute times $0 = \tau_0 \leq \tau_1 \leq \ldots$.

A *run* of $\mathcal{A}$ under π on such a trace is an infinite sequence of locations and valuations

$$(s_0, \nu_0) \xrightarrow{\sigma_0,\, \tau_1 - \tau_0} (s_1, \nu_1) \xrightarrow{\sigma_1,\, \tau_2 - \tau_1} (s_2, \nu_2) \to \ldots$$

where $s_0 \in S_0$, $v_0(x) = 0$ for all $x \in C$, and for every $i \geq 0$ there exists an edge $s_i \xrightarrow{\sigma_i, g_i, \lambda_i} s_{i+1}$ such that

$$(v_i + (\tau_{i+1} - \tau_i)) \models g_i[\pi] \quad \text{and} \quad v_{i+1} = (v_i + (\tau_{i+1} - \tau_i))[\lambda_i := 0].$$

A run is *accepting* iff some state $s \in F$ is visited infinitely often. The language $\mathcal{L}(\mathcal{A}, \pi) \subseteq (\Sigma \times \mathbb{R}_{\geq 0})^\omega$ contains the traces of all accepting runs under π.

Definition 3.4 (Synchronous Composition of PTIOBA)**:** Let

$$\mathcal{A}_i \;=\; \big(\Sigma_i^{\text{in}}, \Sigma_i^{\text{out}}, S_i, S_{0i}, F_i, C_i, P_i, E_i\big), \qquad i \in \{1, 2\},$$

be two *compatible* PTIOBA, i.e.

1. $C_1 \cap C_2 = \varnothing$ and $P_1 \cap P_2 = \varnothing$ (clock/parameter names are distinct);
2. $\Sigma_1^{\text{out}} \cap \Sigma_2^{\text{out}} = \varnothing$ (no output conflict).

Define the set of *synchronising actions*

$$\mathit{Sync} \;=\; (\Sigma_1^{\text{out}} \cap \Sigma_2^{\text{in}}) \;\cup\; (\Sigma_2^{\text{out}} \cap \Sigma_1^{\text{in}}).$$

The *synchronous composition*

$$\mathcal{A}_1 \parallel \mathcal{A}_2 \;=\; \big(\Sigma^{\text{in}}, \Sigma^{\text{out}}, S, S_0, F, C, P, E\big)$$

is defined component-wise:

1. Alphabet:

$$\begin{aligned} \Sigma^{\text{in}} &= \big(\Sigma_1^{\text{in}} \cup \Sigma_2^{\text{in}}\big) \setminus \mathit{Sync}, \\ \Sigma^{\text{out}} &= \big(\Sigma_1^{\text{out}} \cup \Sigma_2^{\text{out}}\big) \setminus \mathit{Sync}. \end{aligned}$$

   Every $a \in \mathit{Sync}$ becomes *internal*: it is neither input nor output in the composite automaton but triggers a joint transition of the two components.

2. States: $S = S_1 \times S_2$, $S_0 = S_{01} \times S_{02}$, $F = F_1 \times F_2$.
3. Clocks and Parameters: $C = C_1 \cup C_2$, $P = P_1 \cup P_2$.
4. Edges: E is the least set generated by the following rules:

   (i) (*Independent step*) If $e_1 = (s_1, \sigma, g_1, \lambda_1, s_1') \in E_1$ with $\sigma \notin \mathit{Sync}$, then

$$(\,\langle s_1, s_2\rangle,\ \sigma,\ g_1,\ \lambda_1,\ \langle s_1', s_2\rangle\,) \in E \quad \text{for all } s_2 \in S_2.$$

   The symmetric rule holds for edges of E_2.

(ii) (*Synchronised step*) If $e_1 = (s_1, a, g_1, \lambda_1, s_1') \in E_1$ and $e_2 = (s_2, a, g_2, \lambda_2, s_2') \in E_2$ with $a \in \mathit{Sync}$, then

$$\big(\langle s_1, s_2 \rangle,\ a,\ g_1 \wedge g_2,\ \lambda_1 \cup \lambda_2,\ \langle s_1', s_2' \rangle \big) \in E.$$

In both cases the guard is the conjunction of the participating guards (omitting the vacuous $\top$ when only one component moves), and the reset set is the union of the resets performed.

A run of $\mathcal{A}_1 \parallel \mathcal{A}_2$ is accepting iff the projection onto each component visits its respective accepting set F_i infinitely often.

3.1 Basic Properties

Proposition 3.1 (Emptiness is Decidable under Fixed Parameters)**:** For any PTIOBA $\mathcal{A}$ and fixed π, non-emptiness of $\mathcal{L}(\mathcal{A}, \pi)$ is decidable in time exponential in $|\mathcal{A}|$ and polynomial in the size of the region graph induced by $\sim_\pi$.

Corollary 3.2 (Universality Reduction)**:** Let $\mathcal{A}$ be a PTIOBA and π a parameter instantiation. The universality problem $\mathcal{L}(\mathcal{A}, \pi) = (\Sigma \times \mathbb{R}_{\geq 0})^\omega$ reduces, via language complementation, to the emptiness problem of a dual PTIOBA of size $O(|\mathcal{A}|)$.

Example 3.3 (Timeout Controller)**:** Consider one clock c, one parameter θ, two states $\{\mathrm{idle}, \mathrm{err}\}$ with $F = \{\mathrm{idle}\}$, and edges

$$\mathrm{idle} \xrightarrow{\texttt{tick},\, c \leq \theta,\, \{c\}} \mathrm{idle}, \qquad \mathrm{idle} \xrightarrow{\texttt{TIMEOUT},\, c > \theta,\, \varnothing} \mathrm{err}.$$

For every $\pi(\theta) > 0$ the trace consisting solely of infinitely many `tick` actions every $\pi(\theta)$ time-units is accepting, whereas any trace that eventually delays longer than $\pi(\theta)$ between two `tick`s is rejected by reaching `err`.

4 Specifying Simple Systems

Example 4.1 (Parametric Time-bounded Channel)**:** The automaton PTChannel in Fig. 1 specifies a reliable FIFO channel that *guarantees* delivery of every queued message within a parameterised upper bound $b \in \mathbb{R}_{\geq 0}$. The second parameter M is an arbitrary type representing the payload carried by each message.

State variable `queue` stores a sequence of packets <m, d>, where m:M is the message and d its absolute deadline. A `send` appends a fresh packet with deadline now+b, while a `receive` removes the head packet provided its message

matches the caller's argument. Time flows at unit rate as long as no packet has reached its deadline.

```
automaton PTChannel(b : Real, M : Type)
-- b is parameterized

type Packet = ⟨ message : M ; deadline : Real ⟩

signature
    external send(m : M)
    external receive(m : M)

state
    queue : Queue[Packet] := ⟨⟩
    now   : Real          := 0

assume                          -- static assumptions
    b ≥ 0

transition external send(m)
    eff
        queue := append(queue, ⟨ m , now + b ⟩)

transition external receive(m)
    pre
        head(queue).message = m
    eff
        queue := tail(queue)

flow                            -- continuous-time evolution
    stop   when ∃p∈queue ∧ now = p.deadline
    evolve where ṅow = 1
```

Figure 1: Parametric time-bounded FIFO channel. Design heavily inspired by the timed FIFO channel described in [13].

Given our definition of a parameterized time bounded FIFO channel, we can reason about *collections* of FIFO channels and collectively bound delays from either direction. For example, consider a scenario where we have n communicating peers, $p_1, \ldots, p_n$, and each peer is connected to each other peer by two one-way FIFO channels. That is, there exists $2 \cdot n^2$ FIFO channels, denoted $c_{a,b}$, where $1 \leq a, b \leq n$ are the sending and receiving peers respectively. In this scenario, we can collectively reason about p_1's cumulative message delay; consider the b parameters of $c_{1,2}, \ldots, c_{1,n}$, denoted $b_{c_{1,2}}, \ldots, b_{c_{1,n}}$. If given some cumulative upper timing bound D we wish to enforce on all outwards communications from p_1, we can simply set the constraint $b_{c_{1,2}} \leq D \wedge \ldots \wedge b_{c_{1,n}} \leq D$.

Example 4.2 (Alternating Bit Protocol)**:** The automaton ABP in Fig. 2 specifies

the classic *alternating-bit protocol*: a single-sender, single-receiver data link that tolerates message loss and duplication by tagging each payload with a parity bit and retransmitting until an appropriate acknowledgement is observed. Parameter $\mathsf{T} \in \mathbb{R}_{>0}$ bounds the *re-transmission timeout*, ensuring every application message is either delivered or re-sent within T time-units. All variables live inside one PTIOBA, so no auxiliary FIFO channel is reused from Example 4.1.

```
automaton ABP(T : Real, M : Type)
-- T is the retransmission timeout (parameterised)

type Packet = ⟨ msg : M ; bit : Bool ; deadline : Real ⟩

signature                        -- interactions with environment
    external appsend(m : M)      -- from sender's application
    external apprecv(m : M)      -- to   receiver's application
    external netin(pkt : Packet) -- network delivers data packet
    external netinack(b : Bool) -- network delivers acknowledgement

state
    sndbit  : Bool          := false   -- expected ACK parity
    rcvbit  : Bool          := false   -- next data parity expected
    pending  : Option[Packet] := None  -- outstanding packet (if any)
    timer    : Real          := +∞      -- next timeout instant
    now      : Real          := 0

assume
    T ≥ 0

-- application issues a new message (only when nothing pending)
transition external appsend(m)
    pre
        pending = None
    eff
        pending := Some(⟨m , sndbit , now + T⟩)
        timer   := now + T

-- data packet arrives at receiver
transition external netin(pkt)
    pre
        pkt.bit = rcvbit
    eff
        apprecv(pkt.msg)              -- deliver to application
        rcvbit := ¬rcvbit             -- expect opposite bit next
        netinack(pkt.bit)             -- emit matching ACK

-- expected acknowledgement arrives at sender
transition external netinack(b)
    pre
        pending ≠ None ∧ b = sndbit
    eff
        pending := None
        sndbit := ¬sndbit
        timer   := +∞

-- retransmission after timeout
transition internal timeout
    pre
        pending ≠ None ∧ now ≥ timer
    eff
        let ⟨ m,b ⟩ = value(pending) in
            pending := Some(⟨ m,b,now + T ⟩)
            timer   := now + T

flow                                -- continuous time evolution
    stop   when pending ≠ None ∧ now = timer
    evolve where ṅow = 1
```


Figure 2: Parametric timed I/O Büchi automaton for the alternating-bit protocol.

5 Dafny Mechanization

Dafny is a verification-aware programming language whose compiler emits verification conditions to SMT solvers (primarily Z3) before it ever generates executable code [15]. Because types, specifications, and ghost state live side-by-side with executable constructs, we can encode each PTIOBA component as a mix of datatypes (for traces and regions), ghost variables (for clocks and parameters), and methods annotated with pre-/post-conditions that mirror our logical rules. While our PTIOBA formulation is very undecidable in practice, Dafny's proof automation allows us to reason about our programs formulated within a PTIOBA in a *semi-automated* way, aiding in completing proofs.

We formulate our PTIOBA automata, including methods for composition and state-space search, within Dafny. Additionally, we formalized parametric timed FIFO channels as described in 1, and ABP as described in 2. Additionally, we formulate the PBFT algorithm using standard I/O Büchi Automata (evidently a subset of PTIOBA). Combining the state-space search operation with constraints on parameters of FIFO channels, we are able to nicely specify timing global guarantees. However, Dafny's proof automation, despite enforcing the necessary conditions, was never able to reason about both state-space search and parameters on timings for FIFO channels. We suspect we would need to formalize the PTIOBA within a constructive, interactive theorem prover such as Coq to successfully automate reasoning.

6 Conclusion

We introduced PTIOBA as a formalism tailored to reason about timing and correctness in byzantine settings. By fusing I/O, parametric timing, and Büchi acceptance, we enable compositional reasoning over global guarantees in systems with adversarial components. Our Dafny mechanization exposes limits of automation, suggesting future work in constructive provers. This is a step towards real proofs of performance for real protocols in real fault models.

References

[1] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, p. 564–577, Jul. 2011.

[2] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens, "An on-demand secure routing protocol resilient to byzantine failures," in *Proceedings of the 1st ACM workshop on Wireless security*. Atlanta GA USA: ACM, Sep. 2002, p. 21–30. [Online]. Available: https://dl.acm.org/doi/10.1145/570681.570684

[3] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. Cengage Learning, 2012.

[4] N. A. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '87. New York, NY, USA: Association for Computing Machinery, 1987, p. 137–151. [Online]. Available: https://doi.org/10.1145/41840.41852

[5] M. Y. Vardi and P. Wolper, "Reasoning about infinite computations," *Information and Computation*, vol. 115, no. 1, p. 1–37, 1994.

[6] M. Y. Vardi, *The Büchi Complementation Saga*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4393, p. 12–22. [Online]. Available: http://link.springer.com/10.1007/978-3-540-70918-3_2

[7] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, *Simple On-the-fly Automatic Verification of Linear Temporal Logic*. Boston, MA: Springer US, 1996, p. 3–18. [Online]. Available: https://doi.org/10.1007/978-0-387-34892-6_1

[8] C. Löding, *Optimal Bounds for Transformations of Omega-Automata*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, vol. 1738, p. 97–109. [Online]. Available: http://link.springer.com/10.1007/3-540-46691-6_8

[9] J. Esparza, T. Munchen, O. Kupferman, and M. Y. Vardi, "Automata-theoretic verification."

[10] J. Bengtsson and W. Yi, *Timed Automata: Semantics, Algorithms and Tools*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, vol. 3098, p. 87–124. [Online]. Available: http://link.springer.com/10.1007/978-3-540-27755-2_3

[11] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, p. 183–235, Apr. 1994.

[12] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, *Kronos: A model-checking tool for real-time systems*, ser. Lecture Notes in Computer

Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, vol. 1427, p. 546–550. [Online]. Available: http://link.springer.com/10.1007/BFb0028779

[13] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata*, ser. Synthesis Lectures on Distributed Computing Theory. Cham: Springer International Publishing, 2011. [Online]. Available: https://link.springer.com/10.1007/978-3-031-02003-2

[14] E. Andre, "What's decidable about parametric timed automata?" *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 2, p. 203–219, Apr. 2019, arXiv:1907.01721 [cs].

[15] K. R. M. Leino, *Dafny: An Automatic Program Verifier for Functional Correctness*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6355, p. 348–370. [Online]. Available: http://link.springer.com/10.1007/978-3-642-17511-4_20