On-The-Fly Symbolic Algorithm for Timed ATL with Abstractions

Nicolaj Ø. Jensen
Department of Computer Science, Aalborg University, Denmark

Kim G. Larsen
Department of Computer Science, Aalborg University, Denmark

Didier Lime
Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, F-44000 Nantes, France

Jiří Srba
Department of Computer Science, Aalborg University, Denmark

Abstract

Verification of real-time systems with multiple components controlled by multiple parties is a challenging task due to its computational complexity. We present an on-the-fly algorithm for verifying timed alternating-time temporal logic (TATL), a branching-time logic with quantifiers over outcomes that results from coalitions of players in such systems. We combine existing work on games and timed CTL verification in the abstract dependency graph (ADG) framework, which allows for easy creation of on-the-fly algorithms that only explore the state space as needed. In addition, we generalize the conventional inclusion check to the ADG framework which enables dynamic reductions of the dependency graph. Using the insights from the generalization, we present a novel abstraction that eliminates the need for inclusion checking altogether in our domain. We implement our algorithms in UPPAAL and our experiments show that while inclusion checking considerably enhances performance, our abstraction provides even more significant improvements, almost two orders of magnitude faster than the naive method. In addition, we outperform UPPAAL TIGA, which can verify only a strict subset of TATL. After implementing our new abstraction in UPPAAL TIGA, we also improve its performance by almost an order of magnitude.

2012 ACM Subject Classification Software and its engineering → Model checking; Theory of computation → Dynamic programming

Keywords and phrases Timed ATL, Symbolic Algorithms, Dependency Graphs, Timed Games

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.25

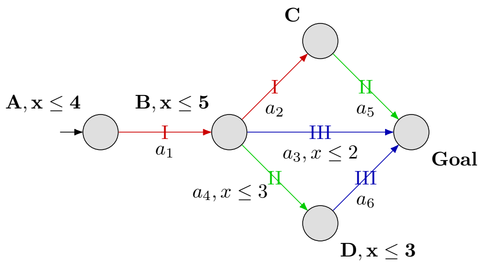
Related Version *Full Version*: `https://arxiv.org/abs/2506.07802`

Supplementary Material
Software (Reprod. Package): `https://doi.org/10.5281/zenodo.15195408`

Funding VILLUM INVESTIGATOR project S4OS and ANR project BisoUS ANR-22-CE48-0012.

1 Introduction

Correctness is essential for many real-time systems, including distributed communication networks, energy grid management, and air traffic control. Ensuring the correctness of such systems is challenging due to the number of concurrent internal and external components. The interleaved behavior of all these components leads to an exponential growth in the number of states, a phenomenon known as the state-space explosion problem [14]. Addressing this challenge is a core focus of the field of model checking. In this paper, we consider real-time systems with components controlled by more than two parties, some cooperative, others adversarial. Real examples of such systems include electronic circuits with multiple components, perhaps some counterfeit [27], and networks of routers located in different



$\mathcal{A} \nvDash \langle\langle \mathrm{I} \rangle\rangle \Diamond \mathbf{Goal}$
$\mathcal{A} \vDash \langle\langle \mathrm{II} \rangle\rangle \Diamond \mathbf{Goal}$
$\mathcal{A} \nvDash \langle\langle \mathrm{III} \rangle\rangle \Diamond \mathbf{Goal}$
$\mathcal{A} \vDash \langle\langle \mathrm{I}, \mathrm{III} \rangle\rangle \Diamond \mathbf{Goal}$
$\mathcal{A} \vDash [\![\mathrm{II}]\!] \Diamond \mathbf{Goal}$
$\mathcal{A} \nvDash \langle\langle \mathrm{II} \rangle\rangle (\neg \mathbf{C}\, \mathcal{U}\, \mathbf{Goal})$
$\mathcal{A} \vDash \langle\langle \mathrm{I}, \mathrm{III} \rangle\rangle (\neg \mathbf{C}\, \mathcal{U}\, \mathbf{Goal})$
$\mathcal{A} \vDash \langle\langle \mathrm{I} \rangle\rangle \Box \neg \langle\langle \mathrm{III} \rangle\rangle \Diamond \mathbf{Goal}$
$\mathcal{A} \nvDash \langle\langle \mathrm{II} \rangle\rangle \Box \neg \langle\langle \mathrm{III} \rangle\rangle \Diamond \mathbf{Goal}$
$\mathcal{A} \nvDash \langle\langle \mathrm{II} \rangle\rangle \Diamond_{<5} \mathbf{Goal}$

(a) An example timed game $\mathcal{A}$ with three players: I, II, and III and a single clock x that increases as time passes. Multiple TATL properties of the system are shown on the right.

$$\mathrm{I} : \langle \ell, \nu \rangle \mapsto \begin{cases} a_1 & \text{if } \ell = \mathbf{A} \\ a_2 & \text{if } \ell = \mathbf{B} \wedge \nu(x) = 5 \\ \lambda & \text{otherwise} \end{cases} \qquad \mathrm{III} : \langle \ell, \nu \rangle \mapsto \begin{cases} a_3 & \text{if } \ell = \mathbf{B} \wedge \nu(x) \leq 2 \\ a_6 & \text{if } \ell = \mathbf{D} \\ \lambda & \text{otherwise} \end{cases}$$

(b) An example strategy profile for player I and III that witnesses $\mathcal{A} \vDash \langle\langle \mathrm{I}, \mathrm{III} \rangle\rangle (\neg \mathbf{C}\, \mathcal{U}\, \mathbf{Goal})$.

Figure 1

countries [16, 6]. We model these systems as timed multiplayer games (TMG) and consider properties described in timed alternating-time temporal logic (TATL) [14, 4]. TMGs are generalized timed game automata [26, 7] and can have more than two players. Like in timed games, each discrete action belongs to a specific player and can be executed only at the discretion of that player. Consider the example TMG in Figure 1a with three players, I, II, and III, and a single real-valued clock x, initially at 0, that increases as time passes. From the starting location, **A**, player III has no strategy to ensure that the system reaches the **Goal** location, expressed as $\mathcal{A} \nvDash \langle\langle \mathrm{III} \rangle\rangle \Diamond \mathbf{Goal}$ in TATL. This is because player I can wait in location **A** until $x > 3$ and then direct the system to location **C** without giving III any influence. Here, the system can remain indefinitely. On the other hand, II can ensure **Goal** is reached as I and III cannot leave the system in a location where II cannot make progress. TATL also allows us to consider coalitions of collaborating players. If players I and III work together, then they can ensure **Goal** is reached without visiting location **C**, expressed as $\mathcal{A} \vDash \langle\langle \mathrm{I}, \mathrm{III} \rangle\rangle (\neg \mathbf{C}\, \mathcal{U}\, \mathbf{Goal})$. We also consider nested properties, such as $\mathcal{A} \vDash \langle\langle \mathrm{I} \rangle\rangle \Box \neg \langle\langle \mathrm{III} \rangle\rangle \Diamond \mathbf{Goal}$ which states that I can ensure that in all reached states, III does not have a strategy to ensure **Goal** is reached. Furthermore, timed properties add conditions on the time passed. For example, $\mathcal{A} \nvDash \langle\langle \mathrm{II} \rangle\rangle \Diamond_{<5} \mathbf{Goal}$ states that II cannot guarantee **Goal** is reached strictly within 5 time units.

For verification, it is often unnecessary to explore the entire state space to determine the correctness of a property. One approach that reduces unnecessary work is to decompose the problem into sub-problems and lazily structure them in a dependency graph [25, 17, 18]. As the process unfolds, sub-problems eventually become trivial, and their solutions are used to resolve the more complex dependent problems. In many cases, the root problem can be answered without first constructing the entire dependency graph and the process can terminate early. This framework was formalized in [17, 18] using extended abstract dependency graphs (EADGs) and has been used to create on-the-fly algorithms for other model checking domains [12, 13, 20].

Our contributions

We develop an on-the-fly TATL algorithm within the EADG framework by providing a sound encoding of the problem. Our approach builds upon previous work involving encodings for timed systems and alternating-time temporal logic, showing their orthogonality, and we provide many semantic details that were left out in previous work. Since TATL is a superset of TCTL [1, 2, 4], our algorithm is also the first on-the-fly algorithm for TCTL with the generic freeze operator. Moreover, we formalize vertex merging in EADGs as a generalization of inclusion checking from prior work [13] and incorporate it into our encoding. In addition, this generalization also leads to an expansion abstraction that removes the need for conventional inclusion checking in our encoding altogether. Finally, we implement and evaluate various configurations of our algorithm and compare their performance to the state-of-the-art model checker for real-time games UPPAAL TIGA [13, 9]. Our configuration with inclusion checking is, as expected, on par with TIGA on the subset of formulae that TIGA can handle, however, incorporating our expansion abstraction improves performance by nearly an order of magnitude.

Related work

The tool UPPAAL TIGA [13, 9], which has since been integrated in UPPAAL [19], uses a similar method to solve a simpler version of the problem: strategy synthesis in two-player timed games. In fact, [13] was published 20 years ago at CONCUR'05, and our paper brings the foundational work of that paper into the modern framework and expands it to the broader TATL logic. TIGA was the first to adapt the dependency graph-based on-the-fly algorithm by Liu and Smolka [25] to model checking, and the idea was eventually generalized into the EADG framework [17, 18]. TIGA relies on inclusion checking to merge vertices and improve performance, a detail that has not been generalized to the EADG framework until this paper. This paper also includes many semantic details that were left out in [13]. Alternating-time temporal logic (ATL), defined by T. Henzinger and R. Alur in [4] as an extension to branching-time logic [14], introduces ways to quantify over possible outcomes resulting from coalitions of players working together. ATL properties with a single coalition can be reduced to a synthesis problem in a two-player game, a problem that has already been intensively studied [26, 7, 15, 13]. However, it is also possible to nest the coalition quantifiers to express more intriguing properties. Recently, Carlsen et al. [12] used the EADG framework for on-the-fly verification of (untimed) ATL properties in concurrent games with multiple players. We extend this work to the setting of timed games and timed logics [1, 2]. An alternative approach to checking branching-time logic properties is the bottom-up algorithm [1, 14, 7]. This method begins by identifying all states that satisfy the relevant atomic propositions. It then iteratively propagates these results to compute states satisfying larger sub-properties until the original property is fully evaluated. While efficient when processing all reachable states is unavoidable, this approach may also waste time on unreachable states or irrelevant state-property combinations. Our on-the-fly algorithm mitigates these issues by computing only the necessary states and sub-properties on demand. Our new expansion abstraction relaxes the relevance criteria, bringing it closer to the bottom-up algorithm in this regard.

Paper structure

In Section 2, we present the TMG formalism and TATL logic followed by the general EADG framework. Then we present our encoding of the TATL problem in EADGs in Section 3. In Section 4, we introduce vertex merging to EADGs and an updated algorithm, allowing us to perform inclusion checking in our encoding, and we introduce our expansion abstraction. Finally, in Section 5, we evaluate our algorithm and compare it against Uppaal Tiga on the problem instances where it is possible.

2 Preliminaries

We shall first introduce the model of timed games with multiple players as well as a timed logics for describing properties of such systems. Afterwards, we present the extended abstract dependency graph framework.

2.1 Timed Multiplayer Games

Let X be a finite set of real-valued variables called *clocks*. Let $C(X)$ be the set of *clock constraints* over the clocks X generated by the following abstract syntax:

$$g ::= x \bowtie k \mid x - y \bowtie k \mid g_1 \wedge g_2$$

where $k \in \mathbb{Z}$ and $x, y \in X$ and $\bowtie\ \in \{<, \leq, =, \geq, >\}$. Let $B(X) \subseteq C(X)$ be the subset that does not use any diagonal constraints of the form $x - y \bowtie k$. Let $\hat{B}(X) \subseteq B(X)$ be the subset that also does not use any constraints of the form $x < k$. A clock *valuation* $\nu : X \to \mathbb{R}_{\geq 0}$ assigns each clock a real value. The valuation denoted $\overrightarrow{0}$ assigns 0 to each clock. We write $\nu[Y]$ for a valuation that assigns 0 to any clock $x \in Y$ and assigns $\nu(x)$ to any clock $x \in X \setminus Y$. If $\delta \in \mathbb{R}_{\geq 0}$ then $\nu + \delta$ denotes the valuation such that $(\nu + \delta)(x) = \nu(v) + \delta$ for all $x \in X$. If $g \in C(X)$, we write $\nu \vDash g$ and say that ν satisfies g if replacing all clocks in g with their respective value in ν makes the expression evaluate to true.

▶ **Definition 1** (Timed Automaton). *A* timed automaton (TA) *is a 6-tuple $\langle L, \ell_{\text{init}}, X, A, T, I\rangle$ where L is a finite set of locations, $\ell_{\text{init}} \in L$ is the initial location, A is a set of actions, X is a finite set of real-valued clocks, $T \subseteq L \times B(X) \times A \times 2^X \times L$ is a finite set of edges each with a unique action from A, and $I : L \to \hat{B}(X)$ assigns an invariant to each location.*

The semantics of a TA $\mathcal{A} = \langle L, \ell_0, X, A, T, I\rangle$ can be described with a labeled transition system. The states are pairs $\langle \ell, \nu\rangle$ where $\ell \in L$ is a location and $\nu : X \to R_{\geq 0}$ is a valuation such that $\nu \vDash I(\ell)$ and Q is the set of all states. Transition labels are discrete actions or real-valued delays, i.e. $A \cup \mathbb{R}_{\geq 0}$, and the transition relation $\to$ is a union of the binary relations $\xrightarrow{a}$ and $\xrightarrow{\delta}$ defined over states as follows:

- $\langle \ell, \nu\rangle \xrightarrow{a} \langle \ell', \nu'\rangle$ if there exists an edge $\langle \ell, g, a, Y, \ell'\rangle \in T$ such that $v \vDash I(\ell) \wedge g$, $v' = v[Y]$, and $v' \vDash I(\ell')$, and
- $\langle \ell, \nu\rangle \xrightarrow{\delta} \langle \ell, \nu'\rangle$ if $\delta > 0$, $v \vDash I(\ell)$, $v' = v + \delta$, and $v' \vDash I(\ell)$.

A *run* in a TA is a sequence $\langle \ell_0, \nu_0\rangle \xrightarrow{t_0} \langle \ell_1, \nu_1\rangle \xrightarrow{t_1} \langle \ell_2, \nu_2\rangle \cdots$ such that $t_i \in A \cup \mathbb{R}_{\geq 0}$. A run is *maximal* if it contains an infinite number of discrete action transitions, ends with an infinite number of delay transitions that sum to ∞ (divergence), or no transition is possible in the final state (a deadlock). For the latter case, there is always a well-defined final state since invariants of the form $x < k$ are disallowed. The set $Runs_{\mathcal{A}}(\langle \ell, \nu\rangle)$ contains all maximal runs starting from $\langle \ell, \nu\rangle$. Given a run σ we write $\sigma[i]$ for the ith state $\langle \ell_i, \nu_i\rangle$.

▶ **Definition 2** (Timed Multiplayer Game). *A* timed multiplayer game (TMG) *with N players is a timed automaton where the actions A have been partitioned into N disjoint sets, $A = A_1 \uplus \cdots \uplus A_N$. We denote by $\Sigma = \{1, \ldots, N\}$ the set of players.*

Strategies and outcomes

A (memoryless) *strategy* for a player $p \in \Sigma$ is a function $f_p : Q \to A_p \cup \{\lambda\}$ that maps each state to λ or a discrete action that is enabled in the state and belongs to p. The strategy informs p what to do in the given state and λ is a special symbol indicating that player p should do nothing and wait. For states where a delay transition is not possible, a strategy can only output λ if the player has no enabled actions available. Let F_p be the set of all strategies for the player p. By ξ_S we denote a strategy profile that assigns to each player $p \in S \subseteq \Sigma$ a strategy $f_p \in F_p$, and $\Xi(S)$ is the set of all such strategy profiles over S. When players pick actions according to their strategies, it restricts the possible runs. We write $Out_{\mathcal{A}}(\xi_S, \langle \ell, \nu \rangle) \subseteq Runs_{\mathcal{A}}(\langle \ell, \nu \rangle)$ for the subset of *outcomes* (runs) induced when the players of S adhere to the strategies of ξ_S.

▶ **Definition 3** (Outcomes). *Let $\mathcal{A} = \langle L, \ell_{\text{init}}, X, A, T, I \rangle$ be a TMG with players $\Sigma = \{1, \dots, N\}$, let $\langle \ell_0, \nu_0 \rangle$ be a state where $\nu_0 \models I(\ell_0)$, and let $\xi_S \in \Xi(S)$ be a strategy profile for $S \subseteq \Sigma$. A maximal run $\sigma = \langle \ell_0, \nu_0 \rangle \xrightarrow{t_0} \langle \ell_1, \nu_1 \rangle \xrightarrow{t_1} \langle \ell_2, \nu_2 \rangle \cdots \in Runs_{\mathcal{A}}(\langle \ell_0, \nu_0 \rangle)$ is in $Out_{\mathcal{A}}(\xi_S, \langle \ell_0, \nu_0 \rangle)$ iff the following conditions hold:*

- *for all $i \geq 0$ such that $t_i = \delta \in \mathbb{R}_{\geq 0}$ we have that $\xi_S(p)(\langle \ell_i, \nu_i + \delta' \rangle) = \lambda$ for all $0 \leq \delta' < \delta$ and all $p \in S$,*
- *for all $i \geq 0$ such that $t_i = a \in A$ we have either $a \notin \bigcup_{p \in S} A_p$ or $\xi_S(p)(\langle \ell_i, \nu_i \rangle) = a$ with $a \in A_p$ for some $p \in S$.*

Note that we allow Zeno behavior (an infinite subsequence consisting of no delays) from both the coalition S and the opposition $\overline{S}$.

2.2 Timed Alternating-Time Temporal Logic

The alternating-time temporal logic [4] is a logic that offers quantification over possible outcomes resulting from a coalition of players in a multiplayer game.

▶ **Definition 4** (Alternating-Time Temporal Logic). *Given a set of atomic propositions Π, a set of clocks X, and a set of players Σ, an* alternating-time temporal logic (ATL) *property conforms to the abstract syntax:*

$$\phi ::= \pi \mid g \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle\!\langle S \rangle\!\rangle \bigcirc \phi_1 \mid \langle\!\langle S \rangle\!\rangle (\phi_1 \mathcal{U} \phi_2) \mid [\![S]\!] (\phi_1 \mathcal{U} \phi_2)$$

where $\pi \in \Pi$, $g \in C(X)$, and $S \subseteq \Sigma$ is a coalition of players.

Given a TMG $\mathcal{A}$ with players $\Sigma = \{1, \dots, N\}$ and a function $Lab : L \to 2^{\Pi}$ labeling locations with atomic propositions, the satisfaction relation $\models$ over states and ATL properties is defined inductively as follows:

- $\langle \ell, \nu \rangle \models \pi$ iff $\pi \in Lab(\ell)$,
- $\langle \ell, \nu \rangle \models g$ iff $\nu \models g$,
- $\langle \ell, \nu \rangle \models \neg\phi$ iff $\langle \ell, \nu \rangle \not\models \phi$,
- $\langle \ell, \nu \rangle \models \phi_1 \vee \phi_2$ iff $\langle \ell, \nu \rangle \models \phi_1$ or $\langle \ell, \nu \rangle \models \phi_2$,
- $\langle \ell, \nu \rangle \models \langle\!\langle S \rangle\!\rangle \bigcirc \phi_1$ iff there exists a strategy profile $\xi_S \in \Xi(S)$ such that for all maximal runs $\sigma = \langle \ell_0, \nu_0 \rangle \xrightarrow{t_0} \langle \ell_1, \nu_1 \rangle \xrightarrow{t_1} \langle \ell_2, \nu_2 \rangle \cdots \in Out_{\mathcal{A}}(\xi_S, \langle \ell, \nu \rangle)$ we have $\sigma[i+1] \models \phi_1$ where $i \geq 0$ is the smallest i such that $t_i \in A$ and such an i must exist,[1]

[1] The next operator is usually left out of timed systems, but it is well-defined with action-based semantics like here.

- $\langle \ell, \nu \rangle \vDash \langle\langle S \rangle\rangle(\phi_1 \mathcal{U} \phi_2)$ iff there exists a strategy profile $\xi_S \in \Xi(S)$ such that for all maximal runs $\sigma \in Out_{\mathcal{A}}(\xi_S, \langle \ell, \nu \rangle)$ we have $\sigma \vDash_r \phi_1 \mathcal{U} \phi_2$ (defined below),
- $\langle \ell, \nu \rangle \vDash [\![S]\!](\phi_1 \mathcal{U} \phi_2)$ iff for all strategy profiles $\xi_S \in \Xi(S)$ there exists a maximal run $\sigma \in Out_{\mathcal{A}}(\xi_S, \langle \ell, \nu \rangle)$ such that $\sigma \vDash_r \phi_1 \mathcal{U} \phi_2$,

and for a maximal run $\sigma = \langle \ell_0, \nu_0 \rangle \xrightarrow{t_0} \langle \ell_1, \nu_1 \rangle \xrightarrow{t_1} \langle \ell_2, \nu_2 \rangle \cdots$ we have $\sigma \vDash_r \phi_1 \mathcal{U} \phi_2$ iff there exists an $i \geq 0$ such that:

- for all $j < i$:
  - if $t_j \in \mathbb{R}_{\geq 0}$ then for all $\delta \in [0, t_j)$ we have $\langle \ell_j, \nu_j + \delta \rangle \vDash \phi_1$, or
  - if $t_j \in A$ instead then $\langle \ell_j, \nu_j \rangle \vDash \phi_1$, and
- either $\langle \ell_i, \nu_i \rangle \vDash \phi_2$ or $t_i \in \mathbb{R}_{\geq 0}$ and there exists a $\delta \in [0, t_i)$ such that $\langle \ell_i, \nu_i + \delta \rangle \vDash \phi_2$, and for all $\delta' \in [0, \delta)$ we have $\langle \ell_i, \nu_i + \delta' \rangle \vDash \phi_1 \vee \phi_2$. We note that the disjunction $\phi_1 \vee \phi_2$ is necessary for timed big-step runs (instead of simply ϕ_1). See [5] for details.

We note that $[\![\emptyset]\!] \equiv \exists$ and $\langle\langle \emptyset \rangle\rangle \equiv \forall$ and thus ATL is a superset of CTL [4]. Other notable abbreviations and equivalences are $\pi \vee \neg\pi \equiv \mathbf{true}$ and $\langle\langle S \rangle\rangle(\mathbf{true} \mathcal{U} \phi) \equiv \langle\langle S \rangle\rangle \Diamond \phi$ and $\neg\langle\langle S \rangle\rangle \Diamond \phi \equiv [\![S]\!] \Box \neg\phi$ and $\neg\langle\langle S \rangle\rangle \bigcirc \phi \equiv [\![S]\!] \bigcirc \neg\phi$.

▶ **Definition 5** (Timed ATL)**.** Timed alternating-time temporal logic (TATL) *is the timed extension of ATL and is defined as ATL but with a new freeze operator: $z.\phi$, where $z \in X$ is a* formula clock *and ϕ is another TATL property. Note that a formula clock z may not be used in the guards and invariants of the TMG. Given a state $\langle \ell, \nu \rangle \in Q$ we have $\langle \ell, \nu \rangle \vDash z.\phi$ iff $\langle \ell, \nu[z] \rangle \vDash \phi$.*

The TATL freeze operator is commonly used indirectly through the more intuitive temporal operator $\mathcal{U}_{\lhd k}$ (where $\lhd \in \{<, \leq\}$). For example, if z is a new clock that does not appear elsewhere in the model or the formula then:

$$\langle\langle S \rangle\rangle(\phi_1 \mathcal{U}_{\lhd k} \phi_2) \equiv z.\langle\langle S \rangle\rangle((\phi_1 \wedge z \lhd k) \mathcal{U} \phi_2)$$

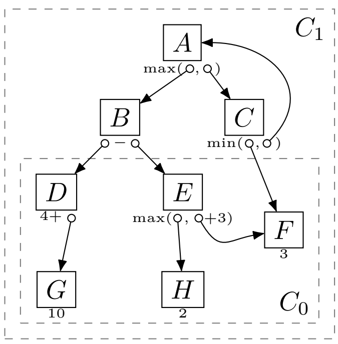
meaning that coalition $S \subseteq \Sigma$ has a strategy profile such that when adhered to, the system always reaches a state satisfying ϕ_2 (strictly) within k time units by a path where ϕ_1 invariantly holds.

An example strategy profile that witnesses $\mathcal{A} \vDash \langle\langle \mathrm{I}, \mathrm{III} \rangle\rangle(\neg\mathbf{C}\, \mathcal{U}\, \mathbf{Goal})$ for the TMG $\mathcal{A}$ in Figure 1a is given in Figure 1b. Since some states are never visited under this strategy, a partial strategy profile would also suffice as a witness. Such partial strategy witnesses enable our algorithm to terminate early for positive cases.

2.3 Extended Abstract Dependency Graphs

A *partially ordered set* $\langle \mathcal{D}, \sqsubseteq \rangle$ is a set $\mathcal{D}$ together with a binary relation $\sqsubseteq \subseteq \mathcal{D} \times \mathcal{D}$ that is reflexive, transitive and anti-symmetric. Given partially ordered sets $\langle \mathcal{D}_1, \sqsubseteq_1 \rangle$ and $\langle \mathcal{D}_2, \sqsubseteq_2 \rangle$, a function $f : \mathcal{D}_1 \to \mathcal{D}_2$ is *monotonically increasing* if $d \sqsubseteq_1 d'$ implies $f(d) \sqsubseteq_2 f(d')$ and *monotonically decreasing* if $d \sqsubseteq_1 d'$ implies $f(d) \sqsupseteq_2 f(d')$. When we just say *monotonic* we refer to increasing monotonicity. When a function has multiple inputs, monotonicity is with respect to a specific input. Unless we specify further, our monotonic multi-input functions are monotonic with respect to all inputs.

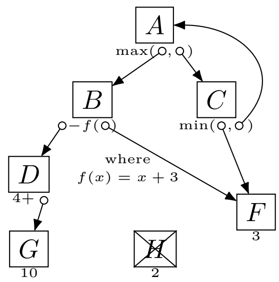
▶ **Definition 6** (Noetherian Ordering Relation)**.** *A* Noetherian ordering relation with a least element (NOR) *is a triple $\langle \mathcal{D}, \sqsubseteq, \bot \rangle$ such that $\langle \mathcal{D}, \sqsubseteq \rangle$ is a partially ordered set, $\bot \in \mathcal{D}$ is the least element such that $\bot \sqsubseteq d$ for all $d \in \mathcal{D}$, and $\sqsubseteq$ satisfies the stabilizing ascending chain condition: for any chain $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \cdots$ there exists a point i such that $d_i = d_j$ for all $j > i$.*



(a) Example EADG.

v	$\alpha_{\min}^{C_0}$	$\alpha_{\min}^{C_1}$
A	-	8
B	-	8
C	-	3
D	14	14
E	6	6
F	3	3
G	10	10
H	2	2

(b) Fixed points of components C_0 and C_1.



(c) The EADG after merge by $E \preceq_f F$ where $f(x) = x + 3$.

Figure 2 (a) An example EADG $\mathcal{G}$ over $\mathcal{D} = \langle \mathbb{N}_0 \cup \{\infty\}, \geq, \infty \rangle$. For each vertex v, the value function $\mathcal{E}(v)$ is displayed below the vertex, and the edges $E(v)$ are displayed using small circles inside the value functions, indicating the arity of the value function as well as which vertex assignment is used as input. Leaf nodes F, G, and H have no dependencies and their value function is a constant function. Since $\mathcal{E}(B)$ is non-monotonic, the graph has two components C_0 and C_1 which are highlighted with dashed boxes. (b) The fixed-point assignments of component C_0 and C_1 found by a fixed-point computation. (c) The EADG $\mathcal{G}[E \mapsto_f F]$ where vertex E has been removed through derivation $E \preceq_f F$. The value function of vertex B is pointwise composed with the derive function f on the second input which now points to the assignment of F instead. Vertex H now has no dependents and can be pruned.

▶ **Definition 7** (Abstract Dependency Graph). *An* abstract dependency graph (ADG) *is tuple* $\langle V, E, \mathcal{D}, \mathcal{E} \rangle$ *where* V *is a finite set of vertices,* $E : V \to V^*$ *is an edge function from vertices to strings of vertices,* $\mathcal{D}$ *is a NOR, and* $\mathcal{E}(v) : \mathcal{D}^n \to \mathcal{D}$ *is a monotonic value function at vertex* $v \in V$ *that takes* n *arguments where* $n = |E(v)|$.

We write $v \to v'$ if $v, v' \in V$ and $v' \in E(v)$, and $\to^+$ denotes the transitive closure of $\to$. The empty string of vertices is denoted ε. Vertices v where $E(v) = \varepsilon$ have constant value functions by definition.

▶ **Definition 8** (Extended Abstract Dependency Graph). *An* extended abstract dependency graph (EADG) *is a tuple* $\langle V, E, \mathcal{D}, \mathcal{E} \rangle$ *where* V*,* E*, and* $\mathcal{D}$ *are defined as for ADGs, but now* $\mathcal{E}(v) : \mathcal{D}^n \to \mathcal{D}$ *is a (possibly non-monotonic) value function at vertex* $v \in V$ *that takes* n *arguments with* $n = |E(v)|$*. Furthermore, if* $\mathcal{E}(v)$ *is non-monotonic, then* v *is not in a cycle, i.e. it is not the case that* $v \to^+ v$*.*

EADGs can be used to encode and solve various problems. Each vertex in the graph represents a problem and the NOR values $\mathcal{D}$ represent possible answers to problems with increasing accuracy. The value function at each vertex describes how the answer to the problem represented by the vertex can be derived from the answers to other (sub)problems. To describe this formally, we must partition the graph into components. Consider the function:

$$dist(v) = \max\{m \in \mathbb{N} \mid \exists v_0 v_1 \cdots v_n \in V^*, v = v_0, \forall i \in \{1, \ldots, n\}, v_{i-1} \to v_i, \\ m = |\{v_i \mid \mathcal{E}(v_i) \text{ is non-monotonic}\}|\} \quad (1)$$

describing how many non-monotonic value functions a vertex depends on. Since an EADG has no cycles with non-monotonic functions, $dist(v)$ is well-defined. The subgraphs induced by $dist$ are called components and $C_i \subseteq V$ is the ith component where $dist(v) \leq i$ for all $v \in C_i$. Component C_0 depends only on monotonic value functions.

Let $\alpha : V \to \mathcal{D}$ be an assignment from vertices to NOR values with $\alpha_\perp$ being the assignment such that $\alpha_\perp(v) = \perp$ for all vertices $v \in V$. Let F_0 be an update function such that:

$$F_0(\alpha)(v) = \mathcal{E}(v)(\alpha(v_1), \alpha(v_2), \ldots, \alpha(v_n)) \tag{2}$$

where $v \in C_0$ and $E(v) = v_1 v_2 \ldots v_n$. Since all value functions in C_0 are monotonic and $\mathcal{D}$ is a NOR, repeated application of F_0 on $\alpha_\perp$ eventually reaches a least fixed point, i.e. there exists an $m > 0$ such that $F_0^m(\alpha_\perp) = F_0^{m+1}(\alpha_\perp)$. We denote this fixed point $\alpha_{\min}^{C_0}$. For each component C_i where $i > 0$ the update function F_i is defined as follows:

$$F_i(\alpha)(v) = \begin{cases} \mathcal{E}(v)(\alpha(v_1), \alpha(v_2), \ldots, \alpha(v_n)) & \text{if } \mathcal{E}(v) \text{ is monotonic} \\ \mathcal{E}(v)(\alpha_{\min}^{C_{i-1}}(v_1), \alpha_{\min}^{C_{i-1}}(v_2), \ldots, \alpha_{\min}^{C_{i-1}}(v_n)) & \text{otherwise} \end{cases} \tag{3}$$

where $E(v) = v_1 v_2 \ldots v_n$ and the assignment $\alpha_{\min}^{C_{i-1}}$ is the minimal fixed point on component C_{i-1}, i.e. the fixed points are defined inductively [18]. Finally, let $C_{\max}$ be the component associated with the greatest *dist* in EADG G, and let $\alpha_{\min}^{G}$ denote $\alpha_{\min}^{C_{\max}}$. Given a vertex $v_0 \in V$, the value $\alpha_{\min}^{G}(v_0)$ can be efficiently found using the sound and complete on-the-fly algorithm introduced in [18].

A small example EADG can be seen in Figure 2a with the NOR $\mathcal{D} = \langle \mathbb{N}_0 \cup \{\infty\}, \geq, \infty \rangle$. In this domain, chains are descending, stabilizing before or at 0. In Figure 2b we show the minimum fixed-point assignment of the components in the example EADG.

3 Encoding TATL to EADGs

In this section we will introduce our symbolic encoding of the TATL problem to EADGs.

Symbolic Operations on Zones and Federations

Due to the uncountable size of the state space, automated verification of timed automata groups the concrete states into effectively representable sets of states and valuations [1, 10]. We shall now define some useful operators on such sets. Let $[\![g]\!] = \{\nu \in \mathbb{R}_{\geq 0}^X \mid \nu \vDash g\}$ be the set of all valuations satisfying $g \in C(X)$. A *zone* is a set $Z \subseteq \mathbb{R}_{\geq 0}^X$ of valuations where for some $g \in C(X)$ we have $[\![g]\!] = Z$. A finite union of zones is called a federation. If $W \subseteq \mathbb{R}_{\geq 0}^X$ is a set of valuations, then:

- $W^{\nearrow} = \{\nu + \delta \mid \nu \in W, \delta \in \mathbb{R}_{\geq 0}\}$ are the timed successors of W,
- $W^{\swarrow} = \{\nu' \in \mathbb{R}_{\geq 0}^X \mid \exists \nu \in W, \exists \delta \in \mathbb{R}_{\geq 0}, \nu' + \delta = \nu\}$ are the timed predecessors of W,
- $W[Y] = \{\nu[Y] \mid \nu \in W\}$ is a reset of the clocks $Y \subseteq X$ in W,
- $W\#x = \{\nu' \in \mathbb{R}_{\geq 0}^X \mid \exists \nu \in W, \forall y \in X, y \neq x \implies \nu(y) = \nu'(y)\}$ frees clock $x \in X$ in W.

Zones are closed under all the above operations [10]. Now consider a set of states $Q' \subseteq Q$ of a timed automaton $\langle L, \ell_{\text{init}}, X, A, T, I \rangle$. We extend these operations on clock valuations to the sets of states such that if $F \in \{\cdot^{\nearrow}, \cdot^{\swarrow}, \cdot[Y], \cdot\#x\}$ then $F(Q') = \{\langle \ell, \nu' \rangle \mid \exists \langle \ell, \nu \rangle \in Q', \nu' \in F(\{\nu\}) \cap [\![I(\ell)]\!]\}$. Given an action $a \in A$, we define the discrete a-predecessors and a-successors as $Pred_a(Q') = \{\langle \ell, \nu \rangle \mid \exists \langle \ell', \nu' \rangle \in Q', \langle \ell, \nu \rangle \xrightarrow{a} \langle \ell', \nu' \rangle\}$ and $Post_a(Q') = \{\langle \ell, \nu \rangle \mid \exists \langle \ell', \nu' \rangle \in Q', \langle \ell', \nu' \rangle \xrightarrow{a} \langle \ell, \nu \rangle\}$, respectively. Both $Pred_a$ and $Post_a$ preserve zones and federations [10]. When the automata is a TMG with players $\Sigma = \{1, \ldots, N\}$ and $S \subseteq \Sigma$ is a coalition, then we let A_S denote $\bigcup_{p \in S} A_p$. The S-controllable discrete predecessors of the coalition is defined by $Pred_S(Q') = \bigcup_{a \in A_S} Pred_a(Q')$. Similarly, the S-uncontrolled discrete predecessors is $Pred_{\overline{S}}(Q')$ where $\overline{S} = \Sigma \setminus S$. Safe timed predecessors are those that avoid a set of states $Q'' \subseteq Q$ even as time elapses. As defined in [13]:

$$Pred_\lambda(Q', Q'') = \{\langle \ell, \nu - \delta \rangle \mid \langle \ell, \nu \rangle \in Q', \delta \geq 0, \forall \delta' \in [0, \delta], \langle \ell, \nu - \delta' \rangle \notin Q''\} \ .$$

The states of Q' where time cannot pass are defined as $Q'^{\not\nearrow} = \{\langle \ell, \nu \rangle \in Q' \mid \exists\text{"}x \leq k\text{"} \in I(\ell), \nu(x) = k\}$. These states are also called *time-locked*. We note that the $\cdot^{\not\nearrow}$ operator preserves federations. Finally, let $Zone(Q) \subseteq 2^Q$ be all subsets of Q that can be represented with a location-zone pair $\langle \ell, Z \rangle$ (also called a symbolic state), and let $Fed(Q) \subseteq 2^Q$ be all subsets of Q that can be represented with a location-federation pair $\langle \ell, \bigcup_i Z_i \rangle$.

The Encoding

We shall now encode the TATL problem on TMGs as an EADG G in order to answer the question: given a TMG $\mathcal{A} = \langle L, \ell_{\text{init}}, X, A, T, I \rangle$ with players $\Sigma = \{1, \ldots, N\}$ and a TATL formula ϕ_0, is it the case that $\langle \ell_{\text{init}}, \overrightarrow{0} \rangle \vDash \phi_0$? The vertices in EADG will have the form $\langle R, \phi \rangle$ where $R \in Zone(Q)$ is a symbolic state and ϕ is a TATL formula (a sub-formula of ϕ_0). We shall sometimes specify the location and zone of R and write $\langle \ell, Z, \phi \rangle$ for vertices instead, but the notation is often simpler for sets of states so we use R in those cases. Our root vertex is $\langle \ell_{\text{init}}, \{\overrightarrow{0}\}^{\nearrow} \cap [\![I(\ell_{\text{init}})]\!], \phi_0 \rangle$ and all dependencies of all vertices are generated using operators that preserve symbolic states. Without loss of generality, we assume all clocks are bounded[2] [13, 10] and hence there are only finitely many symbolic states in practice, and therefore the dependency graph is finite. Our assignment domain is the NOR $\langle Fed(Q), \subseteq, \emptyset \rangle$ and we typically denote elements from our NOR using W. We restrict ourselves such that $\alpha^G_{\min}(\langle R, \phi \rangle) \subseteq R$ by construction and our goal is that $\langle \ell, \nu \rangle \in \alpha^G_{\min}(\langle R, \phi \rangle)$ iff $\langle \ell, \nu \rangle \vDash \phi$. The value functions of our EADG rely on the helper function $Forceable_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W})$ and its counterpart $Unavoidable_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W})$ where S is a coalition of players, W_{ϕ_1} and W_{ϕ_2} are sets of states (typically associated with two TATL formulae), and $\mathcal{W}$ is a set of states. We have $\langle \ell, \nu \rangle \in Forceable_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W})$ if and only if there exists a $\delta \geq 0$ such that $\langle \ell, \nu \rangle \xrightarrow{\delta} \langle \ell, \nu + \delta \rangle$ and

- for all $\delta' \in [0, \delta]$ either $\langle \ell, \nu + \delta' \rangle \in W_{\phi_2}$, or $\langle \ell, \nu + \delta' \rangle \in W_{\phi_1}$ and for all $a_i \in A_{\overline{S}}$ if $\langle \ell, \nu + \delta' \rangle \xrightarrow{a_i} \langle \ell', \nu' \rangle$ then $\langle \ell', \nu' \rangle \in \mathcal{W}$,
- and additionally
  - $\langle \ell, \nu + \delta \rangle \in W_{\phi_2}$, or
  - $\langle \ell, \nu + \delta \rangle \xrightarrow{a_j} \langle \ell', \nu' \rangle \in \mathcal{W}$ for some $a_j \in A_S$, or
  - $\langle \ell, \nu + \delta \rangle \overset{\delta''}{\not\longrightarrow}$ for any $\delta'' > 0$ and $\langle \ell, \nu + \delta \rangle \xrightarrow{a_j} \langle \ell', \nu' \rangle \in \mathcal{W}$ for some $a_j \in A_{\overline{S}}$.

That is, the set is the subset of $W_{\phi_1} \cup W_{\phi_2}$ where coalition S can controllably get to W_{ϕ_2} with delays or to $\mathcal{W}$ with delays and a discrete action all while staying in W_{ϕ_1} until then. The set can be described symbolically as follows:

$$Forceable_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W}) =$$
$$Pred_\lambda \left(W_{\phi_2} \cup Pred_S(\mathcal{W}) \cup \mathcal{H}, \left[\overline{W_{\phi_1}} \cup Pred_{\overline{S}}(\overline{\mathcal{W}})\right] \setminus W_{\phi_2}\right) \tag{4}$$
$$\text{where } \mathcal{H} = Q^{\not\nearrow} \cap Pred_{\overline{S}}(\mathcal{W}) \setminus Pred_\Sigma(\overline{\mathcal{W}}). \tag{5}$$

Similarly, we have $\langle \ell, \nu \rangle \in Unavoidable(W_{\phi_1}, W_{\phi_2}, \mathcal{W})$ if and only if there exists a $\delta \geq 0$ such that $\langle \ell, \nu \rangle \xrightarrow{\delta} \langle \ell, \nu + \delta \rangle$ and

---

[2] It is also possible to use extrapolation techniques instead [8].

- for all $\delta' \in [0, \delta)$ we have $\langle \ell, \nu + \delta' \rangle \in W_{\phi_1}$ and for all $a_i \in A_S$ if $\langle \ell, \nu + \delta' \rangle \xrightarrow{a_i} \langle \ell', \nu' \rangle$ then $\langle \ell', \nu' \rangle \in \mathcal{W}$,
- and additionally
  - $\langle \ell, \nu + \delta \rangle \in W_{\phi_2}$, or
  - $\langle \ell, \nu + \delta \rangle \in W_{\phi_1}$ and $\langle \ell, \nu + \delta \rangle \xrightarrow{a_j} \langle \ell', \nu' \rangle \in \mathcal{W}$ for some $a_j \in A_{\overline{S}}$, or
  - $\langle \ell, \nu + \delta \rangle \in W_{\phi_1}$ and $\langle \ell, \nu + \delta \rangle \overset{\delta''}{\not\longrightarrow}$ for any $\delta'' > 0$ and for all $a_i \in A_S$ if $\langle \ell, \nu + \delta' \rangle \xrightarrow{a_i} \langle \ell', \nu' \rangle$ then $\langle \ell', \nu' \rangle \in \mathcal{W}$ and there exists at least one $a_i \in A_S$ such that $\langle \ell, \nu + \delta \rangle \xrightarrow{a_i}$.

That is, the set contains the subset of $W_{\phi_1} \cup W_{\phi_2}$ where coalition S cannot avoid that the system stays in W_{ϕ_1} and eventually reaches W_{ϕ_2} through delays, or reaches $\mathcal{W}$ through a delay followed by a discrete action. The set can be described symbolically as follows:

$$\begin{aligned} &\mathit{Unavoidable}_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W}) = \\ &\quad \mathit{Pred}_\lambda\left(W_{\phi_2} \cup \mathit{Pred}_{\overline{S}}(\mathcal{W}) \cup \mathcal{H}, \left[\overline{W_{\phi_1}} \cup (\mathit{Pred}_S(\overline{\mathcal{W}}) \setminus \mathit{Pred}_{\overline{S}}(\mathcal{W}))\right] \setminus W_{\phi_2}\right) \qquad (6) \\ &\text{where } \mathcal{H} = Q^{\not\nearrow} \cap \mathit{Pred}_S(\mathcal{W}). \qquad (7) \end{aligned}$$

Assuming a fixed ordering of actions $A = \{a_1, \ldots, a_n\}$, we now define $E(v)$ and $\mathcal{E}(v)$ of the EADG as follows based on the form of vertex v:

- Case $v = \langle R, \pi \rangle$: $E(v) = \varepsilon$ and $\mathcal{E}(v)(\varepsilon) = \{\langle \ell, \nu \rangle \in R \mid \pi \in \mathit{Lab}(\ell)\}$.
- Case $v = \langle R, g \rangle$: $E(v) = \varepsilon$ and $\mathcal{E}(v)(\varepsilon) = \{\langle \ell, \nu \rangle \in R \mid \nu \in [\![g]\!]\}$.
- Case $v = \langle R, \neg\phi \rangle$: $E(v) = \langle R, \phi \rangle$ and $\mathcal{E}(v)(W) = R \setminus W$.
- Case $v = \langle R, \phi_1 \vee \phi_2 \rangle$: $E(v) = \langle R, \phi_1 \rangle \langle R, \phi_2 \rangle$ and $\mathcal{E}(v)(W_1, W_2) = W_1 \cup W_2$.
- Case $v = \langle R, \phi_1 \wedge \phi_2 \rangle$: $E(v) = \langle R, \phi_1 \rangle \langle R, \phi_2 \rangle$ and $\mathcal{E}(v)(W_1, W_2) = W_1 \cap W_2$.
- Case $v = \langle R, z.\phi \rangle$: $E(v) = \langle R[z]^{\nearrow}, \phi \rangle$ and $\mathcal{E}(v)(W) = (W \cap Q[z])\#z \cap R$.
- Case $v = \langle R, \langle\!\langle S \rangle\!\rangle \bigcirc \phi \rangle$:

$$E(v) = v_{a_1} \ldots v_{a_n} \quad \text{where } v_{a_i} = \langle \mathit{Post}_{a_i}(R)^{\nearrow}, \phi \rangle$$
$$\mathcal{E}(v)(W_{a_1}, \ldots, W_{a_n}) = \mathit{Forceable}_S(R, \emptyset, \bigcup_i W_{a_i})$$

- Case $v = \langle R, \langle\!\langle S \rangle\!\rangle (\phi_1 \mathcal{U} \phi_2) \rangle$:

$$E(v) = \langle R, \phi_1 \rangle \langle R, \phi_2 \rangle v_{a_1} \ldots v_{a_n} \quad \text{where } v_{a_i} = \langle \mathit{Post}_{a_i}(R)^{\nearrow}, \langle\!\langle S \rangle\!\rangle (\phi_1 \mathcal{U} \phi_2) \rangle$$
$$\mathcal{E}(v)(W_{\phi_1}, W_{\phi_2}, W_{a_1}, \ldots, W_{a_n}) = \mathit{Forceable}_S(W_{\phi_1}, W_{\phi_2}, \bigcup_i W_{a_i})$$

- Case $v = \langle R, [\![S]\!] (\phi_1 \mathcal{U} \phi_2) \rangle$:

$$E(v) = \langle R, \phi_1 \rangle \langle R, \phi_2 \rangle v_{a_1} \ldots v_{a_n} \quad \text{where } v_{a_i} = \langle \mathit{Post}_{a_i}(R)^{\nearrow}, [\![S]\!] (\phi_1 \mathcal{U} \phi_2) \rangle$$
$$\mathcal{E}(v)(W_{\phi_1}, W_{\phi_2}, W_{a_1}, \ldots, W_{a_n}) = \mathit{Unavoidable}_S(W_{\phi_1}, W_{\phi_2}, \bigcup_i W_{a_i})$$

We remark that the elements of the NOR $\langle \mathit{Fed}(Q), \subseteq, \emptyset \rangle$ have finite representations using the data structure known as a *difference bound matrix* [10, 14]. All value functions are effectively computable since all constituent operations have known algorithms on difference bound matrices [10, 13].

In Figure 3, we show a fragment of the dependency graph generated when checking if $\phi = \langle\!\langle \mathrm{I}, \mathrm{III} \rangle\!\rangle (\neg \mathbf{C}\ \mathcal{U}\ \mathbf{Goal})$ holds in the initial state of $\mathcal{A}$ from Figure 1a. Eventually, the algorithm assigns $[\![x \leq 3]\!]$ to the vertex $\langle \mathbf{A}, x \leq 4, \phi \rangle$ and then we can terminate as we now know that the ϕ holds for the initial state $\langle \mathbf{A}, \overrightarrow{0} \rangle$.

We will now state the correctness of the encoding and its supporting lemma.

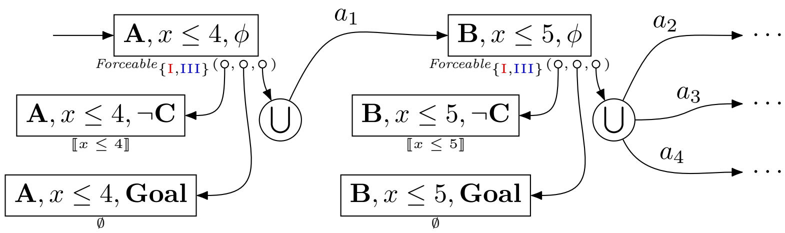


Figure 3 A fragment of the EADG for $\phi = \langle\langle \mathrm{I}, \mathrm{III} \rangle\rangle(\neg \mathbf{C}\,\mathcal{U}\,\mathbf{Goal})$ in $\mathcal{A}$ from Figure 1a.

▶ **Lemma 9** (Monotonically Safe). *The EADG encoding of TATL has no cycles of non-monotonic value functions and* $dist(\langle R, \phi\rangle)$ *is the greatest number of nested negations in* ϕ.

▶ **Theorem 10** (Encoding Correctness). *Given a TMG* $\mathcal{A} = \langle L, \ell_{\mathrm{init}}, X, A, T, I\rangle$*, a set of states* $R \in Zone(Q)$ *such that* $R = R^{\nearrow}$*, a state* $\langle \ell, \nu\rangle \in R$*, and a TATL formula* ϕ*, then* $\langle \ell, \nu\rangle \models \phi$ *iff* $\langle \ell, \nu\rangle \in \alpha^{G}_{\min}(\langle R, \phi\rangle)$ *using the presented EADG encoding of TATL.*

3.1 Unsatisfied states

We now detail an improvement also investigated in [13]. In the encoding presented above, the NOR domain represents the states for which the formula is definitely satisfied. It is also possible to define an encoding in which the NOR domain represents the states in which the formula is definitely not satisfied. It is easy to see that the composition of two NORs is also a NOR [17], so we can do both simultaneously. If the initial state is ever included in the set of unsatisfied valuations then we know that the property does not hold. Hence, with this extension, we can terminate early for many negative cases as well. The encoding using the unsat NOR is trivial for clock constraints $g \in C(X)$, and operators $\wedge$, $\vee$, and $\neg$, but for completeness, we state the encoding of formulae containing the $\bigcirc$ and $\mathcal{U}$ operator below. For all $v \in V$, we use the same edges $E(v)$, but different value functions, denoted $\mathcal{E}^c$, for updating the unsatisfied part of the composed NOR. Here M is used to denote elements in the unsat NOR and we assume the same ordering of actions $A = \{a_1, \ldots, a_n\}$ from earlier:

- Case $v = \langle R, \langle\langle S \rangle\rangle \bigcirc \phi$: $\mathcal{E}^c(v)(M_{a_1}, \ldots, M_{a_n}) = Unavoidable_S(R, \emptyset, \bigcup_i M_{a_i})$
- Case $v = \langle R, \langle\langle S \rangle\rangle(\phi_1 \mathcal{U} \phi_2)$:

$$\mathcal{E}^c(v)(M_{\phi_1}, M_{\phi_2}, M_{a_1}, \ldots, M_{a_n}) = Unavoidable_S(M_{\phi_2}, M_{\phi_1}, \bigcup_i M_{a_i})$$

- Case $v = \langle R, [\![S]\!](\phi_1 \mathcal{U} \phi_2)$:

$$\mathcal{E}^c(v)(M_{\phi_1}, M_{\phi_2}, M_{a_1}, \ldots, M_{a_n}) = Forceable_S(M_{\phi_2}, M_{\phi_1}, \bigcup_i M_{a_i})$$

As seen, the difference between $\mathcal{E}$ and $\mathcal{E}^c$, is that every use of *Forceable* has been replaced by *Unavoidable*, and vice versa, and the terms involving ϕ_1 and ϕ_2 have been swapped in $\mathcal{U}$ formulae.

4 Dynamical Vertex Merge in EADGs

Usually, automated verification of timed systems takes advantage of inclusion checking, i.e. if one discovered symbolic state is included within another, then there is no reason to investigate the smaller one. To do this in the EADG framework, we must first introduce the general concept of vertex merging, which can be used to dynamically reduce the size of the graph.

Consider the NOR $\langle \mathcal{D}, \sqsubseteq, \bot \rangle$ and an EADG $G = \langle V, E, \mathcal{D}, \mathcal{E} \rangle$. If $u \in V^*$ is a string of vertices, then $u[i]$ denotes the ith vertex in the string and $u[v_1 \mapsto v_2]$ denotes the string u where all occurrences of v_1 are replaced with v_2. Given an n-ary function $h : \mathcal{D}^n \to \mathcal{D}$, a unary function $f : \mathcal{D} \to \mathcal{D}$ and an index i, $1 \leq i \leq n$, we use $\circ_i$ to denote pointwise function composition on the ith input, defined as $(h \circ_i f)(x_1, \ldots, x_n) = h(x_1, \ldots, f(x_i), \ldots, x_n)$, and its extension to multiple points $\circ_{\{i_1,\ldots,i_k\}}$ given by $(h \circ_{\{i_1,\ldots,i_k\}} f) = ((h \circ_{i_1} f) \circ_{i_2} \cdots) \circ_{i_k} f$ where notably $h \circ_\emptyset f = h$.

▶ **Definition 11** (Derivation). *Let $v_1, v_2 \in V$ be vertices in EADG $G = \langle V, E, \mathcal{D}, \mathcal{E} \rangle$ with $dist(v_1) \leq dist(v_2)$ and let $f : \mathcal{D} \to \mathcal{D}$ be a monotonic function such that $f(\alpha^G_{\min}(v_2)) = \alpha^G_{\min}(v_1)$. We call f a* derive function *and say that v_1 is* derivable *from v_2 through f, denoted by $v_1 \preceq_f v_2$.*

Derivable vertices can be removed by merging without loss of precision.

▶ **Definition 12** (Vertex Merge). *Let $v_1, v_2 \in V$ be vertices in EADG $G = \langle V, E, \mathcal{D}, \mathcal{E} \rangle$ with $dist(v_1) \leq dist(v_2)$ and let $f : \mathcal{D} \to \mathcal{D}$ be a monotonic function such that $v_1 \preceq_f v_2$. A* vertex merge of v_1 into v_2 by f *is an operation that results in a new EADG denoted $G[v_1 \mapsto_f v_2] = \langle V', E', \mathcal{D}, \mathcal{E}' \rangle$ where*

- $V' = V \setminus \{v_1\}$,
- $E'(v) = E(v)[v_1 \mapsto v_2]$ *for all* $v \in V'$, *and*
- $\mathcal{E}'(v) = \mathcal{E}(v) \circ_I f$ *for all* $v \in V'$ *where* $I = \{i \mid E(v)[i] = v_1\}$.

We can now present a theorem that shows that a vertex merge does not change the minimum fixed-point value of any vertex in the dependency graph.

▶ **Theorem 13** (Merge Preserves $\alpha^G_{\min}$). *Let $G = \langle V, E, \mathcal{D}, \mathcal{E} \rangle$ be an EADG. If $v_1, v_2 \in V$ and $f : \mathcal{D} \to \mathcal{D}$ such that $v_1 \preceq_f v_2$ then $\alpha^G_{\min}(v) = \alpha^{G[v_1 \mapsto_f v_2]}_{\min}(v)$ for all $v \in V \setminus \{v_1\}$.*

There are two main reasons why merging vertices can be beneficial when $v_1 \preceq_f v_2$. First, f may be computationally cheaper than computing $\mathcal{E}(v_1)$. Second, it reduces the size of the graph, making other operations such as back-propagation of updates and pruning cheaper. However, if f is expensive, it may be better to keep v_1 for the memoization it provides.

On Figure 2c, we show how the EADG from Figure 2a looks after a merge by derivation $E \preceq_f F$ where $f(x) = x + 3$. The existence of this derivation depends on the user's domain knowledge. In this case, the user might know that the first input to the max function of E can never exceed 3, and hence its value can be derived from the second input F. As shown in the figure, the resulting graph is smaller and it is also possible to prune the vertex H.

4.1 Algorithm with Vertex Merging

Here we present our modification to the fixed-point algorithm from [18], enabling it to take advantage of vertex merging whenever $v_1 \preceq_f v_2$. The updated algorithm uses the following data structures:

- $\hat{v}$ is the current root vertex, initialized to v_0,
- $\hat{f}$ is the root deriving function, initialized to id (the identity function),
- $\alpha : V \to \mathcal{D}$ is the current assignment, initialized to $\alpha_\bot$,
- W is a waiting set of vertices pending exploration or reevaluation,
- $Pass$ is a set of explored vertices,
- $Dep : V \to 2^V$ is a function that for each vertex v returns a set of dependent vertices that should be reevaluated if the assignment of v changes,

Algorithm 1 Minimum fixed-point computation on an EADG. The lines highlighted in gray are our additions to the algorithm from [18].

```
Input: An EADG ⟨V, E, 𝒟, ℰ⟩ and v₀ ∈ V
Output: d ∈ 𝒟 s.t. d ⊒ α_min(v₀)
Dep(v) = ∅, Edges(v) := E(v), and Eval(v) = ℰ(v) for all v ∈ V;
v̂ := v₀; f̂ = id; α := α_⊥;
W := {v₀}; Pass := ∅; Active := {v₀};
while W ≠ ∅ do
    let v ∈ W where v is pickable;
    W := W \ {v};
    if v = v̂ or Dep(v) ≠ ∅ then
        if v ∉ Pass then
            v := EXPLORE(v);
        if v ∈ Pass or ℰ(v) is non-monotonic then
            let v₁v₂ ... v_k := Edges(v);
            d := ℰ(v)(α(v₁), α(v₂), ..., α(v_k));
            if α(v) ⊏ d then
                W := W ∪ {u ∈ Dep(v) | ∃i.Edges(u)[i] = v ∧ i ∉ IGNORE(α, u)};
                α(v) := d;
                if v = v̂ and {1, 2, ..., k} ⊆ IGNORE(α, v̂) then
                    return f̂(α(v̂))
        Pass := Pass ∪ {v};
return f̂(α(v̂))
proc EXPLORE(v) :
    v_* = v;
    for i = 1 to |Edges(v)| do
        let v_i = Edges(v)[i] (generate it if needed);
        if v_i ∉ Active then
            Active := Active ∪ {v_i};
            forall v' ∈ Active where v' ≠ v_i do
                if v_i ⪯_f v' for some f then
                    Active := Active \ {v_i};
                    Eval(v) := Eval(v) ∘_i f;
                    Edges(v) := Edges(v)[v_i ↦ v'];
                    continue outer loop;
                if v' ⪯_f v_i for some f then
                    REPLACE(v', f, v_i);
                    if v' = v̂ then
                        v̂ := v_i;
                        f̂ := f̂ ∘ f;
                    if v' = v_* then
                        /* We replaced the vertex we were exploring.  Record replacement.  */
                        v_* := v_i;
            if v_* ≠ v then
                return EXPLORE(v_i)
        Dep(v_i) := Dep(v_i) ∪ {v};
        if v_i ∉ Pass then W := W ∪ {v_i};
    return v
proc REPLACE(v₁, f, v₂) :
    Dep(v₂) := Dep(v₁);
    forall v' ∈ Active ∩ Dep(v₁) do
        Eval(v') := Eval(v') ∘_I f where I = {i | Edges(v')[i] = v₁};
        Edges(v') := Edges(v')[v₁ ↦ v₂]
    Pass := Pass \ {v₁};
    Active := Active \ {v₁};
    W := W \ {v₁};
```

- $Edges : V \to V^*$ is a function that for each vertex v returns the current edges of v, initialized to $E(v)$,
- $Eval(v) : \mathcal{D}^n \to \mathcal{D}$ is the current value function for vertex v where $n = |E(v)|$, initialized to $\mathcal{E}(v)$,
- *Active* is a set of vertices that are (still) relevant for the root node $\hat{v}$ and considered during vertex merging.

The variables $\hat{v}$ and $\hat{f}$, and the data structures *Edges*, *Eval*(v), and *Active* are introduced by us to keep track of the reduced graph. In Algorithm 1, we show the EADG minimum fixed point computation algorithm from [18] updated with vertex merging.[3] Incorporating vertex merging is relatively straightforward, but there are some edge cases to discuss. Vertex merging, highlighted in gray in Algorithm 1, occurs during the exploration of the vertex v (line 20-43). Each dependency v_i of v is checked against all other active vertices. If v_i can be merged into another active vertex v', then we make v depend on v' instead and make v_i inactive (lines 27-31). If an active vertex v' can be merged into v_i then we make v' inactive and replace v' with v_i in all our data structures (line 33). Additionally, if v' is the root vertex $\hat{v}$ then we make v_i the new root and update $\hat{f}$ by composing it with f so that the root assignment can be derived later (lines 34-36). In (presumably) rare cases, v' is the explored vertex v, and if so, exploration must be restarted (lines 37-40).

▶ **Lemma 14.** *At the start and end of the* Explore *procedure, there exists no $v_1, v_2 \in Active$ such that $v_1 \neq v_2$ and $v_1 \preceq_f v_2$.*

▶ **Theorem 15.** *The algorithm terminates and returns $\alpha^G_{\min}(v_0)$.*

4.2 Inclusion Checking as Derivation

Finally, we can do vertex merging in our problem domain. If one symbolic state is included within another and the associated query is the same, we merge the smaller one into the bigger one, since it contains the same valuations and more (conventional inclusion checking).

▶ **Theorem 16** (Derivation by Intersection)**.** *Let $\langle R, \phi\rangle, \langle R', \phi\rangle \in V$ be vertices and let $f_R : \mathcal{D} \to \mathcal{D}$ be a function such that $f_R(W) = W \cap R$. If $R \subseteq R'$, then $\langle R, \phi\rangle \preceq_{f_R} \langle R', \phi\rangle$ is a derivation.*

4.3 Expansion Abstraction

It is known that EADGs can be refined using abstractions [18, 21, 22]. We now present a simple abstraction for our domain that has superior performance compared to the inclusion checking. As discussed, R is a symbolic state and $\langle \ell, Z, \phi\rangle$ is the actual form of vertices in the dependency graph. For any vertex $v = \langle \ell, Z, \phi\rangle$ there exists a vertex $v' = \langle \ell, [\![I(\ell)]\!], \phi\rangle$ and v can be merged into v' if it is generated. v' will not be merged into another vertex as it is the biggest w.r.t. our derivation function in Theorem 16. Assuming that v' has a high chance of being generated whenever v is, it makes sense to use an abstraction that takes v directly to v' immediately, as the merge is likely to happen later anyway, and their value functions require a similar number of operations but v' holds more information. Doing so for all vertices induces a new smaller EADG where inclusion checking and merging are unnecessary, as for every location there is a unique zone to be explored. In other words, our abstracted vertices are defined only by the discrete part of the state. If the assumption is wrong, i.e. v' is not discovered although v was, we may end up exploring states and edges that are not relevant in practice as more discrete transitions may be enabled in the the zone $[\![I(\ell)]\!]$ than in Z. However, we assess that timed automata with useless edges are rare in real use cases, and most valuations are eventually reached. As a consequence, the downside of this abstraction is insignificant, as also demonstrated by our experiments.

---

[3] We also rearranged and renamed some procedures for clarity and omitted the pruning procedure. Additionally, Ignore returns a set of indices instead of vertices in our version.

Formally, we apply the following *expansion abstraction* $\mathfrak{X} : V \to V$ to each vertex upon generation:

$$\mathfrak{X}(\langle \ell, Z, \phi \rangle) = \langle \ell, [\![I(\ell)]\!], \phi \rangle \ . \tag{8}$$

▶ **Theorem 17.** *Applying abstraction $\mathfrak{X}$ from Equation* (8) *to all vertices preserves encoding correctness, i.e. $\langle \ell, \nu \rangle \models \phi$ iff $\langle \ell, \nu \rangle \in \alpha_{\min}^{\mathfrak{X}(G)}(\langle R, \phi \rangle)$ for every $\langle \ell, \nu \rangle \in R$.*

If follows, that the vertex merging using the derivation we defined in Theorem 16 becomes redundant when using $\mathfrak{X}$ because there is only one vertex per location-formula pair. The forward exploration done by the algorithm now only considers whether there exists a valuation such that a location can transition another location. This is less strict than a traditional reachability analysis, but it avoids expensive inclusion checks of zones while still restricting the backward propagation to the location-formula pairs we care about.

5 Implementation and Experimental Evaluation

We implement our algorithm in UPPAAL [19]. We use two stacks for the waiting list, one for vertices waiting for exploration and one for vertices waiting for updates. We always prioritize vertices from the update stack over vertices from the exploration stack, as this was found to be favorable in [24]. To improve performance, the EADG framework also allows us to specify a function IGNORE(α, v) describing which vertices of $E(v)$ can be skipped given the current assignment α. We use the universally sound ignore function suggested in [18] by leveraging that $\alpha(\langle R, \phi \rangle) \subseteq R$. We can reduce the number of non-monotonic functions and thus graph components by rewriting the TATL formulae and pushing negations downward to the atomic propositions when possible.

The Benchmark

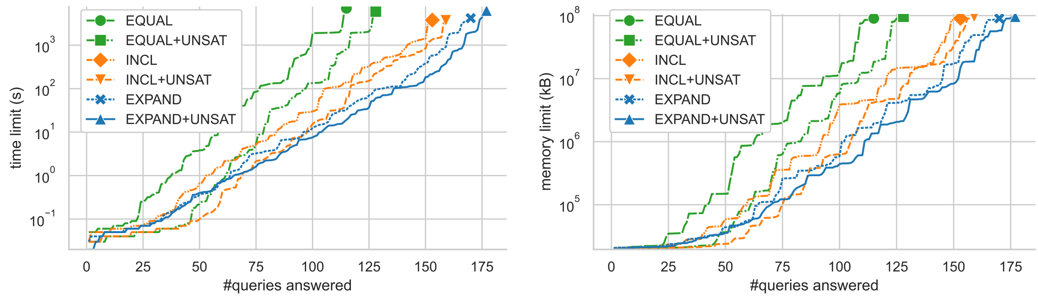
We benchmark various configurations:

- EQUAL: The traditional EADG algorithm without vertex merging.
- INCL: We use vertex merging through inclusion checking as by Theorem 16.
- EXPAND: We use the expansion abstraction $\mathfrak{X}$ described in Section 4.3 (and no vertex merging).
- +UNSAT: We additionally compute unsatisfied states as described in Section 3.1.
- TIGA(+UNSAT): The `verifyta` engine distributed with UPPAAL version 5.0.0 [9, 19] and specifically the TIGA algorithm.[4]

The benchmark suite consists of 3 types of TMG models, each instantiated in various sizes and with multiple associated TATL properties, totaling 236 model and query executions. The models are:

- Train Gate [3]: N trains share the same bridge and a controller signals the trains if the bridge is occupied. Each train and the controller is a different player. Example queries include:
  - $\langle\!\langle T_1, T_2, T_3 \rangle\!\rangle \Diamond \mathit{num\_crossing} \geq 1$; Can train T_1, T_2, and T_3 create a situation where more than one train is crossing at the same time? (No)
  - $\langle\!\langle T_1 \rangle\!\rangle \Diamond \mathit{crossed}(T_1)$; Can train T_1 ensure that it eventually crosses the bridge? (No)

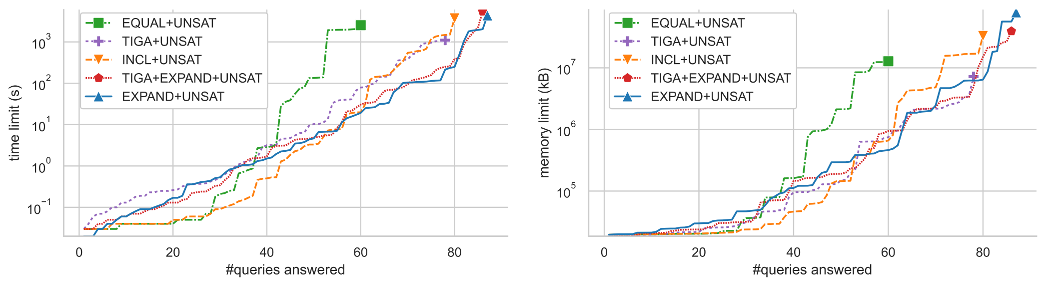
[4] TIGA always keeps track of losing states without a strategy, i.e. the UNSAT modification.



(a) Time usage.

(b) Memory usage.

Figure 4 Cactus plots showing how many queries are answered within the indicated per-query resource limit. A configuration performs better if it can answer more queries using fewer resources.



(a) Time usage.

(b) Memory usage.

Figure 5 Cactus plots showing how many TIGA-compatible queries are answered within the indicated per-query resource limit. A configuration performs better if it can answer more queries using fewer resources.

- Mexican Standoff [12] extended with reload time: N cowboys are shooting at each other and must reload their gun between each shot. Each cowboy is a different player and the gun reloading system is also a player. Example queries include:
  - $\langle\!\langle C_1 \rangle\!\rangle (alive(C_1)\, \mathcal{U}\, t > 1)$; Can cowboy C_1 guarantee staying alive for 1 second? (No)
  - $\langle\!\langle Guns, C_1, \ldots, C_{\lceil N/2 \rceil} \rangle\!\rangle \Box \neg \bigvee_{c \in \{C_1, \ldots, C_{\lceil N/2 \rceil}\}} alive(c)$; If half the cowboys cooperate and the gun reloading is on their side, can they ensure one of them survives? (Yes)
- Phase King [11]: A consensus algorithm for N nodes where time is divided into phases and rounds, and the king of each phase decides the voting tiebreaker. Each node is controlled by a unique player. Example queries include:
  - $[\![\,]\!] \Diamond consensus(n_1, n_2, n_3)$; Node n_1, n_2, n_3 will reach consensus in some trace? (Yes)
  - $\langle\!\langle\rangle\!\rangle \Diamond consensus(n_1, n_2, n_3)$; Node n_1, n_2, n_3 will reach consensus in all traces? (No)
  - $\langle\!\langle\rangle\!\rangle \Box(\neg consensus(n_1, n_2, n_3) \vee \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle \Box consensus(n_1, n_2, n_3))$; If nodes n_1, n_2, and n_3 have consensus, can they can stay in consensus? (Yes, if $N \leq 5$)

Evaluation

Performance comparison of the configurations can be seen in Figure 4. Here the query executions are ordered (independently for each method) by their running time. Note that the y-axis is logarithmic. The INCL configuration using vertex merging is more than one order of magnitude faster than the EQUAL configuration using no merging, and EXPAND is almost another order of magnitude faster for the most difficult instances. The UNSAT modification further improves the three configurations matching the conclusion in [13]. We find a similar improvements for memory usage. As described in Section 4.3, EXPAND may explore edges unnecessarily, since the abstraction enables all transitions in a location, assuming they will be relevant eventually. In the plots, we see that the EXPAND configuration performs worse on easy queries since these extra edges hinder early termination. However, after just 1 second this downside is eliminated by the advantages of the method.

Of our 236 total queries, 150 have no nested coalitions and can be solved by UPPAAL TIGA as well. However, for a large portion of them, modifications to the TMG model are required to get the edges' controllability to correspond to the actions of the coalition. In Figure 5, we compare the performance with TIGA on this subset of 150 queries. As expected, INCL+UNSAT and TIGA+UNSAT have similar performance on challenging queries, as these two configurations are very similar in practice. Our implementation seems slightly faster on easy queries. When we add the EXPAND abstraction to TIGA and remove its now redundant inclusion checking (configuration TIGA+EXPAND+UNSAT), the runtime performance matches that of our EXPAND+UNSAT, almost an order of magnitude faster than the previous state-of-the-art TIGA implementation.

A reproducibility package is available at [23].

6 Conclusion

We presented an encoding of the timed alternating-timed temporal logic problem in the extended abstract dependency graph (EADG) framework. This involved combining previous work on timed games, timed CTL, and alternating-time temporal logic. Our work is thus an example of how various encodings of model-checking problems in the EADG framework are orthogonal and can be combined to solve the combined logic extensions. We also took this opportunity to provide many details left out in the previous paper on UPPAAL TIGA [13]. Furthermore, we formalized a generalization of conventional inclusion checking for the EADG framework. The resulting vertex merging technique can be used to remove vertices that can be derived from other vertices, which is especially useful when the derivation is cheaper than the computation of the value function of the removed vertex. Other domains where we foresee vertex merging being useful include model checking for Petri nets where one marking can cover other markings with less behavior. The vertex merging also allowed us to easily show that we can better exploit the symbolic representation of states using an expansion abstraction. In essence, our abstraction simplifies our symbolic states to discrete locations and their invariants. Hence, *all* valuations satisfying the given property in the location are propagated backward through the dependency graph, instead of restricting it to valuations that the exploration currently considers reachable. Thus, this abstraction makes the algorithm slightly closer to a traditional backward algorithm.

Our implementation and experiments showed that inclusion checking improves the performance of the naive encoding, while the expansion abstraction outperforms it by almost an additional order of magnitude. We also found that our algorithm is comparable to state-of-the-art UPPAAL TIGA when both use inclusion checking. By integrating our expansion abstraction in TIGA we also improved its performance by almost an order of magnitude. The algorithms presented in this paper will be made available in an upcoming release of UPPAAL.

References

1 R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990. doi:10.1109/LICS.1990.113766.

2 R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993. doi:10.1006/inco.1993.1024.

3 Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *LNCS*, pages 8–22, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48683-6_3.

4 Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, September 2002. doi:10.1145/585265.585270.

5 Hugh Anderson and P.S. Thiagarajan. Verification of real-time systems. Course material for CS5270 at NUS Computing, chapter 6, 2007.

6 Ruwaifa Anwar, Haseeb Niaz, David R. Choffnes, Ítalo F. S. Cunha, Phillipa Gill, and Ethan Katz-Bassett. Investigating interdomain routing policies in the wild. *Proceedings of the 2015 Internet Measurement Conference*, 2015. doi:10.1145/2815675.2815712.

7 Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. *IFAC Proceedings Volumes*, 31(18):447–452, 1998. 5th IFAC Conference on System Structure and Control 1998 (SSC'98), Nantes, France, 8-10 July. doi:10.1016/S1474-6670(17)42032-5.

8 Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and Kim G. Larsen. Static guard analysis in timed automata verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 254–270, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/3-540-36577-X_18.

9 Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, LNCS, pages 121–125, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-7.

10 Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. LNCS. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-27755-2_3.

11 Piotr Berman and Juan A. Garay. Cloture votes:n/4-resilient distributed consensus int + 1 rounds. *Mathematical systems theory*, 26(1):3–19, January 1993. doi:10.1007/BF01187072.

12 Falke B. Ø. Carlsen, Lars Bo P. Frydenskov, Nicolaj Ø. Jensen, Jener Rasmussen, Mathias M. Sørensen, Asger G. Weirsøe, Mathias C. Jensen, and Kim G. Larsen. Cgaal: Distributed on-the-fly ATL model checker with heuristics. *Electronic Proceedings in Theoretical Computer Science*, 390:99–114, September 2023. doi:10.4204/eptcs.390.7.

13 Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, volume 3653 of *LNCS*, pages 66–80, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11539452_9.

14 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 1st edition, 2018. doi:10.1007/978-3-319-10575-8.

15 Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. Symbolic algorithms for infinite-state games. In Kim G. Larsen and Mogens Nielsen, editors, *CONCUR 2001 — Concurrency Theory*, LNCS, pages 536–550, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-44685-0_36.

16 Anne Edmundson, Roya Ensafi, Nick Feamster, and Jennifer Rexford. Nation-state hegemony in internet routing. *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, 2018. doi:10.1145/3209811.3211887.

17 S. Enevoldsen, K.G. Larsen, and J. Srba. Abstract dependency graphs and their application to model checking. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19)*, volume 11427 of *LNCS*, pages 316–333. Springer-Verlag, 2019. doi:10.1007/978-3-030-17462-0_18.

18 Søren Enevoldsen, Kim Guldstrand Larsen, and Jiří Srba. Extended abstract dependency graphs. *International Journal on Software Tools for Technology Transfer*, 24(1):49–65, February 2022. doi:10.1007/s10009-021-00638-8.

19 M. Hendriks, Wang Yi, P. Petterson, J. Hakansson, K.G. Larsen, A. David, G. Behrmann, M. Hendriks, Wang Yi, P. Petterson, J. Hakansson, K.G. Larsen, A. David, and G. Behrmann. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*, pages 125–126, 2006. doi:10.1109/QEST.2006.59.

20 Jonas Finnemann Jensen, Kim Guldstrand Larsen, Jiří Srba, and Lars Kaerlund Oestergaard. Efficient model-checking of weighted CTL with upper-bound constraints. *International Journal on Software Tools for Technology Transfer*, 18(4):409–426, August 2016. doi:10.1007/s10009-014-0359-5.

21 Nicolaj Ø. Jensen, Peter G. Jensen, and Kim G. Larsen. Dynamic extrapolation in extended timed automata. In Yi Li and Sofiène Tahar, editors, *Formal Methods and Software Engineering*, LNCS, pages 83–99, Singapore, 2023. Springer Nature Singapore. doi:10.1007/978-981-99-7584-6_6.

22 Nicolaj Ø. Jensen, Kim G. Larsen, and Jiří Srba. Token elimination in model checking of petri nets. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, 2025. doi:10.1007/978-3-031-90643-5_11.

23 Nicolaj Ø. Jensen, Kim G. Larsen, Didier Lime, and Jiri Srba. On-the-fly symbolic algorithm for timed atl with abstractions reproducibility package, April 2025. doi:10.5281/zenodo.15195408.

24 Peter Gjøl Jensen, Kim Guldstrand Larsen, Jiří Srba, and Nikolaj Jensen Ulrik. Elimination of detached regions in dependency graph verification. In Georgiana Caltais and Christian Schilling, editors, *Model Checking Software*, LNCS, pages 163–179, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-32157-3_9.

25 Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 53–66, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi:10.1007/BFb0055040.

26 Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. In Ernst W. Mayr and Claude Puech, editors, *STACS 95*, LNCS, pages 229–242, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. doi:10.1007/3-540-59042-0_76.

27 Mark (Mohammad) Tehranipoor, Ujjwal Guin, and Domenic Forte. *Counterfeit Integrated Circuits*, pages 15–36. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-11824-6_2.