RTWBS Timed Automata: Theory to Implementation

Project Documentation

September 18, 2025

Contents

1 Difference Bound Matrices (DBM)

1.1 Mathematical Foundations

A *zone* over a finite set of clocks $C = \{x_1, \ldots, x_n\}$ is a set of clock valuations satisfying conjunctions of constraints of the form:

$$x_i - x_j \triangleleft c \quad (0 \leq i, j \leq n,\ c \in \mathbb{Z},\ \triangleleft \in \{<, \leq\})$$

with the special clock x_0 fixed to 0. Zones are convex polyhedra in $\mathbb{R}^n_{\geq 0}$.

A Difference Bound Matrix (DBM) is an $(n+1) \times (n+1)$ matrix M where each entry $M[i, j]$ encodes a constraint $x_i - x_j \triangleleft c$. The canonical form (a closed DBM) ensures minimal bounds by computing the all-pairs shortest paths closure (Floyd–Warshall variant). The empty zone is detected if any diagonal entry violates $x_i - x_i \leq 0$.

1.2 Core Operations

Let D denote a DBM:

- **Initialization** (Zero zone): All clocks equal 0: $x_i = 0$. DBM: $M[i, 0] = (0, \leq)$, $M[0, i] = (0, \leq)$.
- **Up** (Time elapse): Removes upper bounds that prevent uniform time increase while preserving differences: formally $\mathrm{Up}(Z) = \{\, v + d \mid v \in Z,\ d \in \mathbb{R}_{\geq 0} \,\}$ restricted by invariants.
- **Guard intersection**: Add constraint $x_i - x_j \triangleleft c$ by tightening $M[i, j]$.
- **Reset** of clock x_k: Replace constraints involving x_k by those implied from setting $x_k := 0$ (i.e., x_k becomes aligned with reference x_0 after closure and guard application).
- **Invariants**: Intersect zone with location invariant constraints before allowing time to pass or after transitions.
- **Extrapolation**: Abstract zone by widening bounds exceeding maximal constants (LU / Max-bounds) to ensure finiteness of the symbolic state space.

1.3 Relations Between DBMs

Given two canonical DBMs D_1 and D_2 over same dimension:

- Inclusion: $D_1 \subseteq D_2$ iff for all i, j: $\mathit{bound}(D_1[i, j]) \leq \mathit{bound}(D_2[i, j])$.
- Equality: Mutual inclusion.
- Emptiness: Diagonal $M[i, i]$ strictly negative (encoded sentinel) after closure.

1.4 Closure Algorithm

Closure computes minimal constraints: for all i, j, k, update $M[i, j] = \min(M[i, j], M[i, k] + M[k, j])$. Complexity: $O(n^3)$, with $n = |C| + 1$. Needed after each batch of constraint tightenings (guards, resets) to maintain canonical form.

1.5 Finiteness via Extrapolation

For timed automata with maximal constant K (from guards/invariants), zone graph may still explode. Extrapolation enforces an abstraction: if an upper-bound exceeds K, replace it by ∞; if a lower-bound is below -K adjust to -K. LU-extrapolation refines this using per-clock separate lower (L) and upper (U) bounds.

1.6 Implementation Touchpoints

In code, helper calls include:

```
dbm_init(zone.data(), dim);        // zero zone
if(!dbm_close(zone.data(), dim))   // closure / emptiness
    ...
dbm_up(zone.data(), dim);          // Up operation
dbm_constrain1(zone.data(), dim, i, j, value); // add guard
// resets handled in apply_transition (not shown here)
dbm_extrapolateMaxBounds(zone.data(), dim, U.data()); // widening
```

All operations act in-place on the contiguous matrix (`std::vector<raw_t>`). Canonical pointers are not stored; structural sharing is avoided for simplicity.

1.7 Worked Example (2 Clocks)

Consider clocks x and y with reference $x_0 = 0$. Start in the zero zone (both clocks 0). We apply:

1. Guard $x \leq 5$.
2. Guard $y - x < 3$.
3. Invariant at location: $y \leq 7$.
4. Time elapse (Up).
5. Reset x (e.g. transition resets x).
6. Extrapolation with $U(x) = U(y) = 5$.

We encode each DBM entry as $(c, \prec)$ where $\prec \in \{<, \leq\}$. Initial (zero) DBM (row index i, column j encodes $x_i - x_j \prec c$):

	x_0	x	y
x_0	$(0, \leq)$	$(0, \leq)$	$(0, \leq)$
x	$(0, \leq)$	$(0, \leq)$	$(0, \leq)$
y	$(0, \leq)$	$(0, \leq)$	$(0, \leq)$

After applying guard $x \leq 5$: tighten entry for $x - x_0$ giving $(5, \leq)$. Guard $y - x < 3$ tightens $M[y, x]$ to $(3, <)$. Closure propagates $y - x_0 < 8$ but invariant $y \leq 7$ later tightens to $(7, \leq)$. Time elapse (Up) removes upper bounds from the reference to allow delay: entries of form $x_i - x_0 \prec c$ with $i > 0$ may widen to $(\infty, \leq)$ except those restricted by invariants (so y retains ≤ 7). Reset x sets $x = 0$: copy row/column of x_0 except constraints involving other clocks; closure re-tightens $y - x < 3$.

Final (simplified) canonical constraints before extrapolation (assuming invariant applied):

$$x \leq 5, \quad y \leq 7, \quad y - x < 3, \quad x \geq 0,\ y \geq 0.$$

Extrapolation with upper bound 5 widens $y \leq 7$ to $y < \infty$ (since $7 > 5$) but keeps $x \leq 5$. Thus we obtain an abstracted zone ensuring finiteness.

Code Trace. The sequence in code:

```
dbm_init(dbm.data(), dim);            // zero
dbm_constrain1(dbm.data(), dim, X, 0, pack_le(5));      // x <= 5
dbm_constrain1(dbm.data(), dim, Y, X, pack_lt(3));      // y - x < 3
dbm_close(dbm.data(), dim);                            // closure
apply_invariant(dbm, loc_inv);                         // y <= 7
```

```
dbm_up(dbm.data(), dim);                                    // Up
reset_clock(dbm, X);                                        // x := 0
dbm_close(dbm.data(), dim);
dbm_extrapolateMaxBounds(dbm.data(), dim, U.data());    // extrapolate
```

This mirrors the theoretical progression: guard intersection, closure, invariant, Up, reset, closure, extrapolation.

Emptiness Corner Case. If we also added guard $y - x < 0$ after $y - x < 3$, closure would derive $y - y < 0$ (diagonal negative) and report emptiness.

2 Time Elapse Semantics

2.1 Theory

Given a zone Z, its time successor Up(Z) is the largest set reachable by letting all clocks advance synchronously by any non-negative real delay d while remaining within location invariants Inv(l):

$$Up(Z) = \{v + d \mid v \in Z, d \in \mathbb{R}_{\geq 0}, \forall t \in [0, d] : v + t \models Inv(l)\}$$

In DBM terms, Up removes upper-bounds of the form $x_i - x_0 \leq c$ (i.e., constraints on absolute clock values), except those imposed indirectly by invariants. Differences $x_i - x_j$ remain invariant under uniform delay, so relative constraints are preserved.

2.2 Finiteness: Extrapolation and LU Bounds

To guarantee a finite symbolic state space, extrapolation widens bounds above per-clock maximal constants (collected from guards/invariants). LU-extrapolation uses lower L(x) and upper U(x) to generalize bounds individually per clock, leading to coarser but finite partition of time.

2.3 Implementation Details

Relevant code (excerpt) from `time_elapse`:

```
std::vector<raw_t> TimedAutomaton::time_elapse(const std::vector<raw_t>&
    zone) const {
    std::vector<raw_t> result = zone;
    dbm_up(result.data(), dimension_);
    if (!clock_max_bounds_.empty() && clock_max_bounds_.size() ==
        dimension_) {
        int global_max = get_max_timing_constant();
        std::vector<int32_t> U = clock_max_bounds_;
        std::vector<int32_t> L = clock_min_lower_bounds_;
        for (cindex_t i = 1; i < dimension_; ++i) {
            if (U[i] <= 0) U[i] = global_max;
        }
        dbm_extrapolateMaxBounds(result.data(), dimension_, U.data());
    }
    return result;
}
```

Steps:

1. Copy zone to mutable buffer.
2. Apply `dbm_up` (canonical Up). Distances to reference loosen.

3. Determine per-clock maximal bounds (fallback to global max if missing).
4. Apply Max-bounds extrapolation: any x_i upper value beyond $U[i]$ replaced by ∞.

Error handling ensures mismatched dimensions or negative delays yield empty zones.

2.4 Fixed Delay Variant

The overload with a concrete delay d approximates exact passage by first applying general Up(Z) then optionally constraining with coarse bounds expressing that at least d time has elapsed. For large delays > 1000 this degenerates to standard Up (abstraction).

3 Zone Graph

3.1 Symbolic State Space

A symbolic state is a pair (l, Z) where l is a control location and Z a zone over clocks consistent with the invariant Inv(l). The zone graph is a directed graph whose nodes are symbolic states reachable from the initial symbolic state through alternation of time elapse and discrete transitions.

3.2 Successor Generation Semantics

For a state (l, Z):

1. Intersect with invariants: $Z' = Z \cap Inv(l)$.
2. Let time pass: $Z'' = Up(Z')$. (Implicit invariant restriction during delay).
3. For each enabled transition $e = (l, g, R, a, l')$ with guard g, resets R:
   (a) Check enabled: $Z'' \cap g \neq \emptyset$.
   (b) Apply reset: $Z''' = Reset(Z'' \cap g, R)$.
   (c) Apply target invariant: $Z_{succ} = Z''' \cap Inv(l')$.
   (d) Canonical closure and extrapolation keep zone finite; add node if new.

This is breadth-first in the implementation to avoid deep recursion and to enable early pruning or potential future heuristics.

3.3 Implementation (Excerpt)

```
void TimedAutomaton::explore_state(int state_id) {
    const auto& current_state = *states_[state_id];
    auto zone_with_inv = apply_invariants(current_state.zone,
        current_state.location_id);
    if (zone_with_inv.empty()) return;
    auto elapsed_zone = time_elapse(zone_with_inv);
    if (elapsed_zone.empty()) return;
    auto outs = outgoing_transitions_.find(current_state.location_id);
    if (outs != outgoing_transitions_.end()) {
        for (int tidx : outs->second) {
            const auto& t = transitions_[tidx];
            if (is_transition_enabled(elapsed_zone, t)) {
                auto post = apply_transition(elapsed_zone, t);
                if (!post.empty()) {
                    auto final_zone = apply_invariants(post, t.
                        to_location);
```

```
                        if (!final_zone.empty()) {
                            int succ_id = add_state(t.to_location,
                               final_zone);
                            zone_transitions_[state_id].push_back(succ_id);
                        }
                    }
                }
            }
        }
}
```

Key optimisations:

- Hash-consing of zones via `state_map_` to avoid duplicates.
- Extrapolation performed inside time elapse / post-processing to guarantee finiteness.
- BFS queue (`waiting_list_`) ensures systematic expansion.

3.4 Initial State

The initial zone is the canonical zero DBM (all clocks set to 0). Construction begins at default initial location (configurable) with zone closure applied immediately.

3.5 Inclusion Checks

Canonical DBM comparison enables relation seeding in refinement (mutual inclusion equals equality). Hash value embedded in `ZoneState` accelerates lookups.

4 Relaxed Weak Timed Bisimulation (RTWBS)

4.1 Motivation

Classical timed bisimulation is often too strict for component refinement where receivers may accept messages over a wider timing window and senders may commit earlier. RTWBS introduces *direction-sensitive* relaxation over weak timed semantics.

4.2 Classical Timed Bisimulation Recap

Two timed automata A and B are timed bisimilar if there exists a relation R over states such that $(s_A, s_B) \in R$ implies:

- Time: For all delays d, $s_A \xrightarrow{d} s'_A$ implies $\exists s'_B$ with $s_B \xrightarrow{d} s'_B$ and $(s'_A, s'_B) \in R$ (and symmetrically).
- Action: For all observable a, $s_A \xrightarrow{a} s'_A$ implies $\exists$ matching $s_B \xrightarrow{a} s'_B$ with $(s'_A, s'_B) \in R$ (and symmetrically).

Weak variants replace direct a by the weak form $\Rightarrow a \Rightarrow$ where each $\Rightarrow$ denotes a (possibly empty) sequence of internal (tau) and delay steps.

4.3 Asymmetric Timing Relaxation

In RTWBS we distinguish channels with direction (send !, receive ?). Intuition:

Send (!) Refined must not allow more time than abstract before sending: enabling zone refined $\subseteq$ abstract.

Receive (?) Refined may be more permissive: abstract enabling zone ⊆ refined.

Internal Standard weak inclusion refined ⊆ abstract.

This yields a *preorder* rather than equivalence; mutual refinement recovers a symmetric notion.

4.4 Formal Rule (Symbolic Form)

Let (l_r, Z_r) and (l_a, Z_a) be symbolic states with $(l_r, Z_r)\mathcal{R}(l_a, Z_a)$. For each observable action a performed by refined via transition t_r with guard g_r and resets R_r leading to (l'_r, Z'_r):

$$Up(Z_r \cap Inv(l_r) \cap g_r) =: E_r$$

Similarly for abstract candidate t_a: $E_a = Up(Z_a \cap Inv(l_a) \cap g_a)$. The timing side condition depends on direction:

$$\text{SEND: } E_r \subseteq E_a \quad \text{RECEIVE: } E_a \subseteq E_r \quad \text{INTERNAL: } E_r \subseteq E_a$$

Then there must exist t_a with same observable label a and direction for which the side condition holds and $\exists (l''_r, Z''_r) \in post_{weak}(l'_r, Z'_r), (l''_a, Z''_a) \in post_{weak}(l'_a, Z'_a)$ such that $(l''_r, Z''_r)\mathcal{R}(l''_a, Z''_a)$. The converse direction need not hold (refinement).

4.5 Weak Successors

Weak post uses τ-closure before and after the observable step: $post_{weak}(X) = closure_\tau(post_a(closure_\tau(X)))$. Closure enumerates only internal transitions respecting invariants.

4.6 Soundness Rationale

The asymmetric inclusion ensures the refined system does not introduce earlier-or-later behaviours violating the abstract specification per direction constraints. Using zones preserves relative timing invariants; extrapolation keeps graph finite without violating inclusion monotonicity.

5 Game-Based Checking Algorithm

5.1 Attacker/Defender Paradigm

We cast refinement as a one-sided safety game. States of the game are candidate pairs in relation R. The *attacker* chooses an observable transition of the refined automaton; the *defender* must reply with a weak $(\tau^* a \tau^*)$ transition of the abstract automaton satisfying timing side conditions.

If defender cannot match, the pair is losing and removed. Greatest fixed point of non-losing pairs constitutes the refinement relation.

5.2 Game Graph (Implicit)

The algorithm avoids explicit construction of the full product graph; instead it iteratively filters a hash set of pairs. Successor generation (weak $\tau^* a \tau^*$) is recomputed on demand.

5.3 Termination

Number of zone pairs is finite due to extrapolation. Each iteration strictly removes at least one pair or stops. Thus algorithm terminates.

5.4 Pseudo-Code

```
R = { (z_r, z_a) | loc(z_r)=loc(z_a) }
repeat
  removed = false
  for (p in R):
     for each observable tr from p.r:
        if no matching abstract weak transition exists in p.a:
            mark p for removal; break
  remove all marked from R
until !removed
return !R.empty()
```

5.5 Complexity Considerations

Worst-case: $O(|R| \cdot (deg_r \cdot (W_r + W_a)))$ where W_x is cost of computing weak successors. Without caching τ-closures may repeat. Memoisation can reduce recomputation to linear in number of distinct (zone,action) pairs.

5.6 Optimisation Opportunities

- Memoize τ-closure and weak successors.
- Pre-filter abstract transitions by label/direction.
- Maintain reverse dependency graph (predecessors) to localise removals.
- Early pruning using DBM inclusion seeding (exclude impossible pairs initially).

5.7 Implemented Optimisations

The current code base implements the four opportunities above:

Closure Cache Map `ZoneState*` to vector of τ-reachable states (function: `tau_closure_cached`). Avoids repeated BFS.

Weak Successor Cache Keyed by (zone, action) (struct `WeakKey`); stores $\tau^* a \tau^*$ endpoints.

Early Pruning Seed Initial relation only includes pairs with same location and refined zone $\subseteq$ abstract zone (DBM inclusion).

Reverse Dependencies Map from supporting pair to dependents that relied on it to justify at least one match (`reverse_deps_`). When a pair is removed, only its dependents are re-validated (worklist), avoiding full rescans.

Complexity Impact. If C_τ is average τ-closure size, naive recomputation costs $O(|R| \cdot deg \cdot C_\tau)$ per iteration. Caching reduces it to amortised $O(N_\tau + N_{weak})$ where N_τ and N_{weak} are number of distinct closure / weak-successor queries.

Memory Tradeoff. Caches store vectors of raw pointers only; reverse dependency graph stores adjacency lists restricted to observed supporting edges, typically much smaller than full $|R|^2$ worst-case.

5.8 Worked Example (Mini Game)

Assume refined automaton R and abstract automaton A each have two locations $L0, L1$ and one observable label a plus internal τ. Initial zones are trivial (all clocks 0). Transitions:

$$\begin{array}{c|l} R & L0 \xrightarrow{a,\, x\leq 2,\, x:=0} L1 \\ A & L0 \xrightarrow{\tau,\, x<1} L0, \text{ then } L0 \xrightarrow{a,\, x\leq 3,\, x:=0} L1 \end{array}$$

Game iteration for pair $(L0, Z0), (L0, Z0)$:

1. Attacker picks a from R (enabled for delays d with $d \leq 2$).
2. Defender computes weak successors in A: τ-closure allows any number of τ steps while $x < 1$, then must delay to some $d \leq 3$ for the a transition. Combined pattern: delay $d_1 < 1$, zero or more times, then additional delay d_2 with $d_1 + d_2 \leq 3$.
3. Timing side condition: every refined enabling delay $d \leq 2$ must be matchable. Choose defender decomposition with $d_1 = \min(d, 0.9)$ (staying under 1) and $d_2 = d - d_1$. Since $d \leq 2$, total $d_1 + d_2 = d \leq 2 \leq 3$ holds; guard $x \leq 3$ satisfied.
4. Defender succeeds; pair survives.

If refined guard were $x \leq 4$, delays $d \in (3, 4]$ would not be matchable (abstract guard caps at 3). Pair would be removed, implying no refinement.

Code Path. In C++: attacker side enumerated in `observable_edges(refined)`. Defender calls `weak_observable_successors(abstract, 'a')` producing zones after $\tau^* a \tau^*$. Function `timing_ok` reconstructs enabling DBMs and checks inclusion: refined enabling zone $\subseteq$ abstract enabling zone for action a. Failure triggers marking for removal.

Caching Impact. If multiple pairs share the same abstract zone and label a, memoising the weak successors avoids recomputing the τ-closure, reducing complexity from repeated closure traversals to a single stored vector of resulting zones.

6 Mapping: Theory ↔ Implementation

6.1 DBM Layer

Theory Concept	Code Primitive
Zone initialization	dbm_init (time_elapse prep)
Closure	dbm_close (implied inside guard/reset helpers)
Up operation	dbm_up (inside time_elapse)
Guard intersection	dbm_constrain1 (timing_ok / enabling)
Extrapolation	dbm_extrapolateMaxBounds (time_elapse)
Inclusion test	dbm_relation (timing_ok)

6.2 Zone Graph

- BFS list: `waiting_list_`
- Canonical state store: `state_map_` (hash of ZoneState)
- Successor expansion: `explore_state`
- Transition enablement: `is_transition_enabled`

6.3 Weak Semantics

τ-closure Function: `tau_closure`. BFS over internal edges applying invariants + time + transitions.

Weak successors `weak_observable_successors`: $\tau^* a \tau^*$ pattern.

6.4 Timing Side Conditions

Function `timing_ok`: constructs enabling zones $(Up((Z \cup Inv) \cup g))$ and compares with inclusion direction based on synchronization kind.

6.5 Game Loop

Function `RTWBSChecker::check_rtwbs_equivalence`: greatest fixed point elimination. Relation container: unordered_set of pairs with custom hash.

6.6 Statistics

Accumulated in `last_stats_`: refined/abstract state counts, surviving relation size, wall-clock time, approximate memory.

6.7 Potential Extensions

- Counterexample reconstruction: store witness abstract matches; backtrack on failure.
- On-demand zone expansion: lazily build successors only when needed.
- Partitioned relation: per-location buckets to reduce scan cost.
- LU-extrapolation refinement: integrate lower bounds L for more precision.

7 Future Work and Research Directions

7.1 Symbolic Partial Order Reductions

Combine τ-closure with stubborn set or ample set selection using clock dependency analysis to prune interleavings.

7.2 Parametric Timed Bisimulation

Generalize guards with parameters; integrate parametric DBM (PDBM) constraints; solve via SMT-backed refinement.

7.3 Compositional Reasoning

Introduce interface zones per component, compose via synchronized product with assume/guarantee contracts.

7.4 On-the-fly Abstraction

Derive LU-bounds dynamically from frontier zones; feed back tightened bounds to extrapolation to curb blow-up.

7.5 Counterexample-Guided Refinement

If relation collapses to empty, extract mismatching pair path to refine partition or introduce clock splitting.

7.6 Parallel Exploration

Sharding of waiting list by location hash; lock-free insertion into state map with per-bucket spinlocks.

7.7 Probabilistic Extensions

Annotate transitions with rates/probabilities; lift RTWBS to weak probabilistic bisimulation using coupling arguments.