Accelerating Synthesis of Timed Game using Async Parallelisation and GPU

Summary

Oliver S. Bak, Mathias W. B. Christiansen, and Oliver V. Eriksen

Department of Computer Science, Aalborg University, Denmark

The concrete semantics of a Timed Automaton (TA) and its extensions such as Timed Game Automaton (TGA) define an uncountable and infinite state space. Model checking of such automata which involves a systematic exploration of the state space thus utilising symbolic semantics, making it a task of solving simple constraint systems over real valued clock variables, typically modelled as Difference Bound Matrices (DBMs). We present methods of transforming and operating on DBMs massively in parallel on Graphical Processing Unit (GPU) enabled architectures, for the purpose of reducing verification time of model families with larger symbolic state spaces. The content of this thesis is based on our previous development of the prototype tool SMAcc for the purpose of GPU enabled co-processing symbolic reachability analysis of networks of TAs, made jointly with Marcus D. Jensen, Simas Juozapaitis and Andreas Windfeld.

We now extend the work of SMAcc with methods and techniques for the controller synthesis problem for TGAs. This verification task consists of finding discretely defined winning states through a forward exploration, and back-propagating the reachable winning information to an initial state while avoiding losing states based on the notion of uncontrollable transitions. This presents unique challenges such as dynamic GPU memory allocation, as the process of back-propagating winning information involves DBM subtractions where the size of the result set cannot be known a priori.

We have re-implemented our previous work with new methods and techniques, as well as the support for synthesis of TGA in the **GDBM** (**G**PU **DBM**) library where we see improved performance on isolated DBM operations compared to the work of SMAcc, and promising results in comparison with **UDBM** (Uppaal's DBM library). In the best case, we report a 68.2 times speedup compared to UDBM, and a 10.32 times speedup compared to SMAcc. We extend both Uppaal and Uppaal Tiga to allow for asynchronous co-processing with integrated GDBM functionality, such that Uppaal (Tiga) only operates on the discrete state space while GDBM operates on the continuous state space. Our experimental evaluation of these extensions show promising results, both being within a single order of magnitude slower than Uppaal, which in the case of reachability analysis is a huge improvement to SMAcc. We suggest points for future work that may be worth investigating if these methods are to rival Uppaal (Tiga).

Accelerating Synthesis of Timed Game using Async Parallelisation and GPU

Oliver S. Bak, Mathias W. B. Christiansen, and Oliver V. Eriksen

Department of Computer Science, Aalborg University, Denmark

Abstract. Manipulation of Difference Bound Matrices (DBMs) is essential to the symbolic verification and synthesis of varying extensions of Timed Automaton. We have previously shown the applicability of Graphical Processing Units (GPUs) for computing such DBM operations massively in parallel for symbolic reachability analysis. We now extend this work to the controller synthesis problem for Timed Game Automaton. Many non-trivial challenges arise from DBM operations involved in this verification task, such as the non-convexity of subtract operations that are part of back-propagating winning information. We discuss our techniques and approaches for achieving a high degree of parallelism and occupancy on a GPU enabled architecture. We have extended and reworked functionality from our prototype tool SMAcc to build **GDBM** – a library for **G**PU computations of **D**ifference **B**ound **M**atrices operations. We have additionally extended both Uppaal and Uppaal Tiga to allow for integration of GDBM in a co-process of discrete and continuous exploration, respectively. Experimental results show benefits of using GPUs for most DBM operations in isolation, having up to a 68.2 time speedup specifically for computing canonical forms, and a 10.32 times speedup in comparison to our previous work. This performance increase on DBM operations translates well to both reachability analysis and controller synthesis of timed games. We improve on our previous work on reachability analysis with an up to 178.72 times speedup, while additional work is needed on controller synthesis if it is to rival Uppaal Tiga.

1 Introduction

The aim of the *controller synthesis* problem is to construct a strategy for guiding a governed system towards some desired state. For timed systems, the problem can be formulated through *Timed Game Automata* – an extension of Timed Automata with the notion of controllable and uncontrollable transitions [35]. The concrete-semantics of Timed Automata based models define an uncountable and infinite state space, making them ill-equipped for the purpose of any sort of analysis where the systematic exploration of the state space is required. Tools such as Uppaal Tiga [9] typically employ *symbolic semantics* that finitely partitions the state space into convex *zones* – constraint systems over clock values and their differences – following the work of Alur & Dill [1] and their region based technique. Difference Bound Matrices (DBMs) offer a canonical representation

of the constraint system, where entries in the matrix represent bounds on given clock variables [22]. Symbolic state representations have the obvious benefit of making state space exploration feasible for timed systems. Still, the infamous *state space explosion* problem plagues all branches of the field of model checking where, in the case of timed automata based models, the number of states grows exponentially in the number of system clocks, and model components [11]. Historically, research within the field has been primarily concerned with reducing the affect of this problem in terms of both reducing the memory requirement and the runtime, e.g. [30,13,25,31,12].

In our previous work [3], we established the potential of performing DBM operations massively in parallel using GPUs. GPUs have also been used in other branches of formal verification such as LTL Model Checking, e.g. [4,6] where the state space generation is being performed on the GPU. Closer to our contribution is the work by Wijs & Osama [37], where they perform LTL Model Checking by using the GPU for both the state space generation and exploration. Statistical model checking of *Stochastic timed automata* has also previously been achieved on a GPU [2]. Still, to the best of our knowledge, this work is the first to incorporate GPU for synthesis of timed games. Many similarities can be found in the extensive research of multi-core based model checking of both timed and untimed systems, e.g. [16,5]. GPU based architectures still present unique and non-trivial challenges, in particular for the case of timed games which appear to have seen no previous efforts to be parallelised.

The contribution of this thesis is twofold: we have reworked and extended our previously established GDBM library to support the operations present in synthesis of timed game automata and, extended the state-of-the-art model checking tools Uppaal and Uppaal Tiga for asynchronous GPU enabled reachability analysis and synthesis of timed games, respectively. The remaining of this thesis is structured as follows: in Section 2 we introduce the theoretical background of this thesis such as the semantics of timed (game) automata and DBMs, Section 3 introduces the CUDA framework and related algorithmic notations of parallel concepts, Section 4 details our methods and techniques for implementing DBM operations for GPU architectures, Section 5 describes the symbolic reachability integration with Uppaal, Section 6 details the Timed Games integration with Uppaal Tiga, Section 7 details the conducted experimental evaluation, and Section 8 concludes on the results and details further possible improvements.

2 Preliminaries

We introduce the theory of timed automata, symbolic reachability analysis, and difference bound matrices as a prerequisite for the theory for synthesis of timed games.

2.1 Timed Automata

The theory of timed automata is based on the work of Alur & Dill [1]. A timed automaton is a finite state automaton extended with a finite set of real-valued

clocks. In a timed automaton, locations are labelled with an *invariant*, and transitions are labelled with a *guard*, both of which are conditions on clocks. In addition, transitions are labelled with a *clock update*, denoting a subset of clocks to be assigned a new value. A timed automaton begins execution with all clocks set to zero, and clocks increase uniformly with time while an automaton resides within a location, which it can do so long as the invariant of the node is satisfied. A transition occurs instantaneously, and all clocks in the clock update set will be assigned their new value.

Clocks. Let X be a finite set of clocks. A *clock valuation* is then a function $u : X \to \mathbb{R}_{\geq 0}$ (denoted $\mathbb{R}^X_{\geq 0}$).

Constraints. The set $B(X)$ is the set of conjunctive formulas of atomic constraints on the form $x \sim m$ or $x - y \sim n$, where $x, y \in X$, $\sim \in \{<, \leq, =, \geq, >\}$, and $m, n \in \mathbb{N}$. The elements of $B(X)$ are referred to as *clock constraints*, and are ranged over by g.

For $r \subseteq X$, $u[r]$ denotes the valuation assigning 0 for any $x \in r$. By $u + \delta$ for $\delta \in \mathbb{R}_{\geq 0}$ we denote the valuation s.t. for all $x \in X, (u + \delta)(x) = u(x) + \delta$. For $g \in B(X)$ and $u \in \mathbb{R}_{\geq 0}$, $u \vDash g$ denotes that u satisfies g, and $[\![g]\!]$ denotes the set of valuations $\{u \in \mathbb{R}^X_{\geq 0} \mid u \vDash g\}$. A *zone* D is a subset of $\mathbb{R}^X_{\geq 0}$ s.t. $[\![g]\!] = D$ for some $g \in B(X)$.

Definition 1 (Timed Automaton (TA)). *A timed automaton is a tuple* (L, l_0, A, X, E, I) *where:*

- L *is a finite set of locations,*
- $l_0 \in L$ *is the initial location,*
- A *is a set of actions,*
- X *is a set of clocks,*
- $E \subseteq L \times B(X) \times A \times 2^X \times L$ *is the set of edges,*
- $I : L \to B(X)$ *assigns invariants to locations*

When $(l, g, a, r, l') \in E$*, we write* $l \xrightarrow{g,a,r} l'$

The semantics of a TA is defined as a transition system TS, with states of the form (l, u), where l is a location in the TA and u is a clock valuation, i.e. the set of states $Q = L \times \mathbb{R}^X_{\geq 0}$. Based on this there are two types of transitions $\to$:

$$\begin{aligned} &\text{for } a \in A, (l, u) \xrightarrow{a} (l', u') \text{ if } l \xrightarrow{g,a,r} l' \in E \text{ s.t. } u \vDash g, u' = u[r] \text{ and } u' \vDash I(l') \\ &\text{for } \delta \geq 0, (l, u) \xrightarrow{\delta} (l, u') \text{ if } u' = u + \delta \text{ and } u, u' \in [\![I(l)]\!] \end{aligned} \tag{1}$$

A network of timed automata (*NTA*) is a parallel composition of timed automata such that $NTA = TA_1 \mid \cdots \mid TA_n$. Given a NTA there is a vector of locations $\bar{l}$ with an entry for each TA in the network and $\bar{l}[i]$ giving the location of the i'th TA. The location vector refers to the discrete part of the state space. The invariant $I(\bar{l})$ is the conjunction of the invariants of each location in $\bar{l}$. When a transition is possible in the i'th network we write $l_i \xrightarrow{g,a,r} l'_i$.

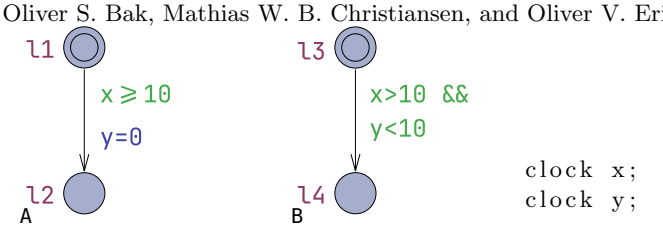


Fig. 1: Network of two timed automata A and B with the locations l1, l2, l3 and l4, and the clocks x and y

Example. An example of an NTA can be seen in Figure 1. A trace of the network could be as follows: $([\texttt{l1}, \texttt{l3}], (x = y = 0)) \xrightarrow{10} ([\texttt{l1}, \texttt{l3}], (x = y = 10)) \xrightarrow{\tau} ([\texttt{l2}, \texttt{l3}], (x = 10, y = 0)) \xrightarrow{5} ([\texttt{l2}, \texttt{l3}], (x = 15, y = 5)) \xrightarrow{\tau} ([\texttt{l2}, \texttt{l4}], (x = 15, y = 5))$. This trace shows the general behaviour of the system, but the amount of delay transitions and the delay value within them can differ in other traces.

The state space of TAs is infinite and uncountable. The analysis of TAs are thus based on the exploration of a finite *Simulation Graph* with nodes being *Symbolic States* on the form (l, D) where $l \in L$ and D is a *zone* in $\mathbb{R}^X_{\geq 0}$. The symbolic state transition $\rightarrow$ can be defined as:

$$(l, D) \xrightarrow{a} (l', D') \text{ if } l \xrightarrow{g,a,r} l' \in E \text{ and } D' = ((D \cap [\![g]\!])[r])^{\nearrow} \tag{2}$$

For $S \subseteq Q$ and an action $a \in A$, $Post_a(S) = \{(l', u') \mid \exists (l, u) \in S, (l, u) \xrightarrow{a} (l', u')\}$ defines the a-successor, while $Pred_a(S) = \{(l, u) \mid \exists (l', u') \in S, (l, u) \xrightarrow{a} (l', u')\}$ defines the a-predecessor. The timed successor and predecessor of a symbolic state is respectively defined as $S^{\nearrow} = \{(l, u + \delta) \mid (l, u) \in S \cap [\![I(l)]\!], (l, u + \delta) \in [\![I(l)]\!], \delta \in \mathbb{R}_{\geq 0}\}$ and $S^{\swarrow} = \{(l, u - \delta) \mid (l, u) \in S, \delta \in \mathbb{R}_{\geq 0}\}$.

Example. The symbolic execution of Figure 1 is as follows: $([\texttt{l1}, \texttt{l3}], (x \geq 0, y \geq 0, x = y)) \xrightarrow{\tau} ([\texttt{l2}, \texttt{l3}], (x \geq 10, y \geq 0, x - y \geq 10)) \xrightarrow{\tau} ([\texttt{l2}, \texttt{l4}], (x > 10, y \geq 0, x - y \geq 10, y < 10))$.

A run of a TA denotes a sequence of symbolic transitions in the labelled TS. $Runs((l, u), T)$ denotes the set of runs that begin in symbolic state (l, u) of the timed automaton T. We use $Runs(T)$ as a shorthand for $Runs((l_0, \mathbf{0}), T)$, where $\mathbf{0}$ indicates $\{x \mapsto 0 \mid \forall x \in X\}$. For a finite run ρ, $last(\rho)$ denotes the last state of the run.

Given a TA and a property ϕ, the on-the-fly symbolic reachability algorithm, as seen in Algorithm 1, can be used to check whether the TA satisfies ϕ. The algorithm utilises a combined *passed* and *waiting* list PW for storing both the states that have been explored and are waiting to be explored [19]. This structure is used such that a state can be checked against both the waiting- and passed list at the same time.

Algorithm 1 Abstract, symbolic reachability algorithm

```
1: W ← {(l_0, D_0^↗)}
2: PW[l_0] ← {D_0^↗}
3: while W ≠ ∅ do
4:     (l, D) ← pop(W)
5:     if (l, D) ⊢ φ then return true
6:     for all {(l', D') | (l, D) → (l', D')} do
7:         if ∀Y' ∈ PW[l'] : D' ⊈ Y' then
8:             PW[l'] ← PW[l'] ∪ {D'}
9:             W ← W ∪ (l', D')
10: return false
```

2.2 Difference Bound Matrix

The zones used for exploration in timed automata are represented as *Difference Bound Matrices (DBMs)* as presented in [22]. A DBM is a matrix D that describes the set of constraints: $\forall_{x,y}\ x - y \leq D_{x,y}$ where x and y range over X_0 i.e. describing all valid clock valuations based on a set of constraints. X_0 is a set of clocks $X_0 = X \cup x_0$ where x_0 is a reference clock that will always map to the value zero. This is done for a uniform treatment of clock constraint as $x_i - x_j \sim m$, i.e. $x_i \sim m$ is written as $x_i - x_0 \sim m$, for $\sim = \{<, \leq\}, m \in \mathbb{Z}$. All clock constraints in $B(X)$ can be written as a conjunction of constraints in this form, e.g. $x_i - x_j > 7$ is equivalent to $x_j - x_i < -7$. Under the assumption that all clock constraints in $B(X)$ include the implicit constraints on clocks $x_0 - x_i \leq 0$ and $x_i - x_0 < \infty$, a clock constraint can be viewed as a set of upper bounds on the difference between pairs of clocks, which can be represented as a DBM with the dimensions $|X_0| \times |X_0|$. We will henceforth write n as a shorthand for $|X_0|$. Each entry in the DBM is specified as a bound i.e. a value and the relation between the clocks e.g. given the constraint $x_j - x_i < -7$ the bound $(-7, <)$ would be in the row specifying x_j and the column specifying x_i.

Basic DBM Operations Bengtsson [11] and Behrmann et al. [8] describe DBM operations and transformations. These all assume that DBMs are in their canonical form – *closed under entailment* – i.e. all constraints of the DBM are as tight as possible. This is straightforwardly computed in $\mathcal{O}(n^3)$ time by an all-pairs shortest path algorithm, e.g. Floyd-Warshall [23]. Operations that transform a DBM are illustrated in Figure 2 and are listed in the following:

Empty checks whether a DBM represents the empty set i.e. having no valid clock valuations. Checking whether a DBM D is empty i.e. $[\![D]\!] = \varnothing$ is computed as $\exists_{x,y}(D_{x,y} + D_{y,x} < (0, \leq)$ in $\mathcal{O}(n^2)$ time.

Inclusion checks whether two DBMs have a subset equal relation between them. This is also referred to as one DBM being subsumed by another if they are a subset or equal. Given two DBMs D and D' checking whether $D \subseteq D'$ is computed as $\forall_{x,y}(D_{x,y} \leq D'_{x,y})$, with x and y ranging over X_0. The operation is computed in $\mathcal{O}(n^2)$ time.

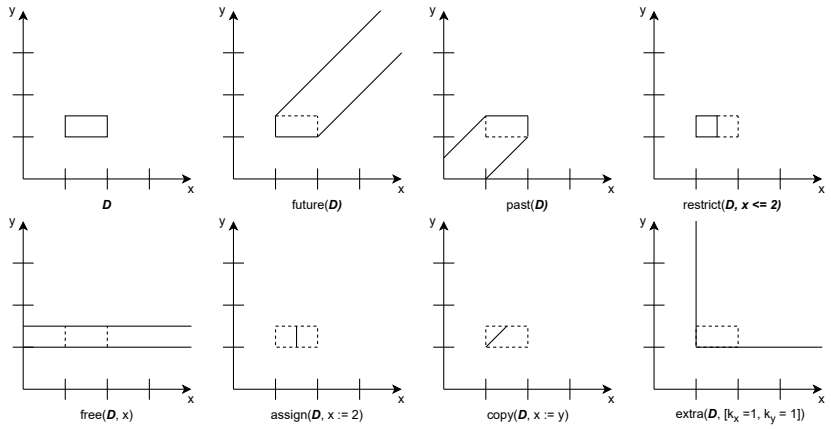


Fig. 2: Different operations applied to the same DBM D. The dashed lines indicate the original DBM and the solid lines indicate the DBM after applying an operation.

Intersection finds the intersection between two DBMs. Given two DBMs D and D' then the intersecting DBM $D'' = D \wedge D'$ is computed as $\forall_{x,y}(D''_{x,y} = \min(D_{x,y}, D'_{x,y}))$ where x and y range over X_0. The operation is computed in $\mathcal{O}(n^3)$ time.

Future expands a DBM to include the clock valuations reachable with a delay. Given a DBM D then $\forall_x(D_{x,0} \leftarrow \infty)$, with x ranging over X_0. The operation is computed in $\mathcal{O}(n)$ time.

Past expands a DBM to include the clock valuations that can reach this DBM with a delay. Given a DBM D then past is computed as $\forall_x(D_{0,x} \leftarrow 0)$, with x ranging over X_0, extra care is needed to keep D on canonical form. The operation is computed in $\mathcal{O}(n^2)$ time.

Restrict adds a new constraint to the set of constraints for a given DBM. Given a DBM D and a constraint $x - y \sim m$ then $D_{x,y} = \min((m, \sim), D_{x,y})$, with $x, y \in X_0$, extra care is needed to keep D on canonical form. The operation is computed in $\mathcal{O}(n^2)$ time.

Assign updates the DBM by assigning a clock within the DBM to a specific value. Given a DBM D, a clock x and a value v then $D_{x,0} = (v, \leq)$, $D_{0,x} = (-v, \leq)$ and $\forall_i(D_{x,i} \leftarrow (v, \leq) + D_{0,i}$ and $D_{i,x} \leftarrow D_{i,0} + (-v, \leq))$, with i ranging over X_0. The operation is computed in $\mathcal{O}(n)$ time.

Copy copies the bounds for one clock in the DBM to another. Given a DBM D and two clocks x and y then $\forall_i(D_{x,i} \leftarrow D_{y,i} \wedge D_{i,x} \leftarrow D_{i,y})$, with i ranging over $X_0 \backslash x$. Additionally, $D_{x,y}, D_{y,x} = 0$. The operation is computed in $\mathcal{O}(n)$ time.

Extrapolation is used to obtain a finite-zone graph using the maximal constant each clock is compared to in the model. The maximal constants can be found through static-analysis of the TA beforehand. Given a DBM D where

$D_{x,y} = (c_{x,y}, \sim_{x,y})$ and a set of maximal constants M then,

$$\forall_{x,y}\ (c'_{x,y}, \sim'_{x,y}) = \begin{cases} \infty & \text{if } c_{x,y} > M(x) \\ (-M(y), <) & \text{if } -c_{x,y} > M(y) \\ (c_{x,y}, \sim_{x,y}) & \text{otherwise} \end{cases} \tag{3}$$

where x and y range over X_0. This operation is computed in $\mathcal{O}(n^2)$ time, however, the canonical form of the DBM is broken afterwards. This can be expanded with a courser interpretation using diagonal extrapolation:

$$\forall_{x,y}\ (c'_{x,y}, \sim'_{x,y}) = \begin{cases} \infty & \text{if } c_{x,y} > M(x) \\ \infty & \text{if } -c_{0,x} > M(x) \\ \infty & \text{if } -c_{0,y} > M(y), i \neq 0 \\ (-M(y), <) & \text{if } -c_{x,y} > M(y), i = 0 \\ (c_{x,y}, \sim_{x,y}) & \text{otherwise} \end{cases} \tag{4}$$

LU Extrapolation works like extrapolation but uses both the maximum upper and lower constants for each clock giving a coarser interpretation. Given a DBM D where $D_{x,y} = (c_{x,y}, \sim_{x,y})$, a set of Upper constants U and a set of Lower constants L then,

$$\forall_{x,y}\ (c'_{x,y}, \sim'_{x,y}) = \begin{cases} \infty & \text{if } c_{x,y} > L(x) \\ (-U(y), <) & \text{if } -c_{x,y} > U(y) \\ (c_{x,y}, \sim_{x,y}) & \text{otherwise} \end{cases} \tag{5}$$

where x and y range over X_0. As with maximum extrapolation a coarser interpretation can be gained using diagonal LU extrapolation:

$$\forall_{x,y}\ (c'_{x,y}, \sim'_{x,y}) = \begin{cases} \infty & \text{if } c_{x,y} > L(x) \\ \infty & \text{if } -c_{0,x} > L(x) \\ \infty & \text{if } -c_{0,y} > U(y), i \neq 0 \\ (-U(y), <) & \text{if } -c_{x,y} > U(y), i = 0 \\ (c_{x,y}, \sim_{x,y}) & \text{otherwise} \end{cases} \tag{6}$$

Local Extrapolation uses the constants (either maximum or LU) for a given location instead of the global maximum. This approach is described by Behrmann et al. in [7]. The model might have an update with the maximum constant coming after the update, this would result in a location using a constant that is not relevant for that location. Instead the constants are backwards propagated through the model. A constant is not propagated over an edge if there is an update for the corresponding clock. The semantics are given just as extrapolation and LU extrapolation but with local sets of constants for each location.

The pseudo code for all the DBM operations can be found in Appendix A.

Subtract Given two DBMs D and E we may want to find valuations satisfying $D \wedge \neg E$, denoted by $D - E$ as *subtracting* E from D. This operation results in a non-convex set and can therefore not be represented by a single DBM, instead requiring a set of DBMs. The result $S = D - E$ is defined as:

$$S = D \wedge \neg(\bigwedge_{1 \leq i,j \leq n} e_{ij}) = \bigvee_{1 \leq i,j \leq n} (D \wedge \neg e_{ij}), \tag{7}$$

which is a union of D, each being restricted by a negated constraint of E, respectively. The straight forward computation of subtract following this definition is $\mathcal{O}(n^4)$ time, and the size of the result set is bounded by $n^2 - n$.

Different orderings of the constraints in E when computing the subtraction can have varying effect on the size of S. A heuristic function for ordering the constraints e_{ij} is given in [20] as $|e_{ij}| - |d_{ij}|$, sorting on the smallest values first. The heuristic value should be recomputed after constraining, as the DBM will change as a result of the operation.

Furthermore, it is suggested in [20] to use the minimal set of constraints of E obtained from *shortest-path reduction*, denoted E_m, to compute the subtraction, as the two are semantically equivalent s.t. $D - E = D - E_m$. For the purpose of self-containment, we summarise the $\mathcal{O}(n^3)$ method for computing the minimal set of constraints from [30,11] in the following section.

Shortest-Path Reduction A DBM D can be represented as a weighted, directed graph with vertices corresponding to clocks in X_0, and an edge from x to y with weight m given that $x - y \leq m$ is a constraint of D [11]. The shortest-path reduction entails removing *redundant edges*. An edge (x, y) is redundant whenever there is an alternative path from x to y whose accumulated weight is less than or equal to the weight of the edge itself. This is straight forward given the closure of the DBM, as only paths with a length of 2 needs to be considered, i.e. an edge (x, y) is redundant if there is a vertex $z(\neq x, y)$ s.t. $(x, y) \geq (x, z) + (z, y)$ [30].

Although redundant, removal of these edges is dependant on the existence of *zero-cycles* in the graph. For zero-cycle free graphs, the redundant edges can simply be removed without affecting the solution set. Otherwise, the reduction is based on a partitioning of the vertices according to zero-cycles. Two vertices are part of the same *equivalence class* whenever there is a zero-cycle that contains them both. This is a simple check given the canonical form of the DBM, as there is zero-cycle between vertices x and y, denoted $x \equiv y$, precisely when $(x, y) = -(y, x)$ [30]. Obtaining the shortest-path reduction is thus a two-step process: finding a single shortest path between equivalence classes, and removing edges within equivalence classes until only the edges forming a single zero-cycle is left. This assumes some ordering on the vertices.

Shortest-path reduction of a canonical DBM with zero cycles is illustrated in Figure 3. This example DBM contains a zero-cycle between, and thus equivalence class of, $\{x_0, x_1\}$. The vertex x_0 is the elected leader meaning it is the only vertex from its equivalence class to have edges to the remaining graph.

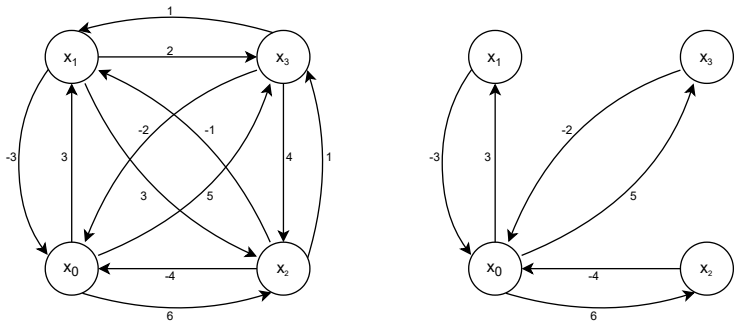


Fig. 3: Graph representation of a canonical DBM, and its shortest-path reduction. All bounds are implicitly non-strict.

Federations The set of clock constraints $B(X)$ is closed under conjunction but not disjunction, as the union of two convex sets is not guaranteed to be convex itself. A finite union of zones, called a *federation*, needs to be treated slightly differently to DBMs due to this non guarantee of convexity. Firstly, it must always be the case that no DBM is subsumed by another DBM if they are part of the same federation, i.e. for the federation F, $\forall (D, D') \in F\ (D \neq D')\ :\ D \notin D', D' \notin D$. Many of the previously described DBM operations are trivially extended to federations as simply the union of performing these operations on all DBMs in the federation, e.g. the intersection of federations $F \cap F'$ is $\bigcup_{D \in F} \bigcup_{D' \in F'} D \cap D'$. Furthermore, the termination of both reachability analysis of timed automata and timed games are determined by testing if a newly found zone is subsumed by a federation.

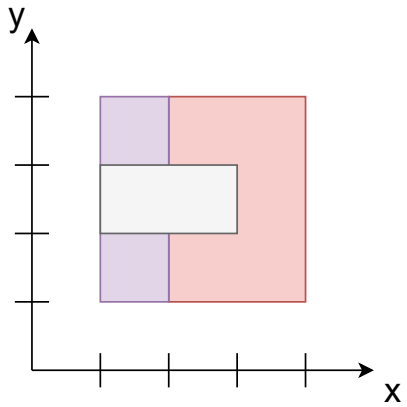


Fig. 4: The grey zone is subsumed by the entire federation (union of purple and red zones) but not by any zone individually

A DBM may only be superseded by the aggregated federation and none of its contained DBMs in isolation, as in the example of the grey zone in Figure 4 against the federation consisting of the purple and red zones. This can be detected by subtracting the DBM from the federation and checking for emptiness, i.e. for a federation $F = D_1 \cup ... \cup D_n$ and a zone D, we check for subsumption $D \subseteq F$ as $D - F = \varnothing$. The subtract operation is computationally heavy, but will guarantee the detection of subsumption. Alternatively, one may check for an *approximate* subsumption $D \subseteq F$, simply by checking sequentially on each zone in the federation, i.e. $\exists D_k \in F$ s.t. $D \subseteq D_k$. This does not guarantee the detection of subsumption, and unnecessary symbolic states may be explored as a result, but the operation is much faster.

Merging Some unions of zones *are* convex, such as the zones of Figure 4, and it would be beneficial to describe these as just a single DBM to reduce memory requirements and the number of operations. Given a federation F of n DBMs, we can attempt to merge this federation by computing the convex hull C_F of F, and check if $C_F - F = \varnothing$. If this holds, then the convex hull is equal to the federation, and we can thus replace F by C_F [17,18]. The difficulty lies in finding a subset of DBMs to attempt to merge. There are $\binom{n}{k}$ ways to attempt to merge k DBMs together, and trying this for all values of k gives a total of 2^n combinations. Because every attempt relies on the already expensive subtract operation, this becomes unfeasible. To make this more practical (but obviously less beneficial), one may limit themselves to check only pairs of DBMs A and B. Furthermore, it is beneficial to be able to cheaply discard pairs with low likelihood to be mergeable. [17] suggests (in part) the following heuristic:

$$\begin{aligned} \exists i,j.a_{ij} = b_{ij} \wedge a_{ji} = b_{ji} \\ \forall i,j.\neg(-a^+_{ij} \geq b^+_{ji} \vee -a^+_{ji} \geq b^+_{ij}) \end{aligned} \tag{8}$$

where a^+_{ij} denotes the *non-strict* constraint ij in A, e.g. $(x_i - x_j < 5)^+ = (x_i - x_j \leq 5)^+ = (x_i - x_j \leq 5)$. If this holds for a pair of DBMs, we attempt to merge them through the convex hull check.

2.3 Timed Game Automata

For the purpose of synthesis of timed games, timed automata can be extended to include the notion of controllable and uncontrollable actions [33].

Definition 2 (Timed Game Automaton (TGA)). *A timed Game Automaton (TGA) is a timed automaton with its actions A partitioned s.t. A_c and A_u define controllable and uncontrollable actions respectively.*

Given a TGA G and a set of states $W \subseteq Q$, the *controller synthesis problem* for reachability games entails finding a *strategy* f s.t. applying f to G guarantees W. The problem is similar for safety games in that it entails finding a strategy

f which avoids W in G. For a reachability game (G, W), a given finite or infinite run $\rho \in Runs(G)$ is winning if it reaches a state in W. Similarly, a safety game is losing whenever it reaches a state in W. The set of winning runs given (l, u) in G is denoted as $WinRuns((l, u), G)$. For the purpose of conciseness, we shall henceforth only detail semantics regarding reachability games, as a safety game can be transformed into a reachability game by swapping controllable and uncontrollable actions as well as goal states. A *strategy* is then a function that gives information during the game s.t. the controller knows what actions should be taken in order to win the game.

Definition 3 (Strategy). *Given a TGA G, a strategy f for G is a partial function mapping from $Runs(G)$ to $A_c \cup \{\lambda\}$ s.t. for any finite run ρ, if $f(\rho) \in A_c$ then $last(\rho) \xrightarrow{f(\rho)} (l', u')$ for some (l', u').*

The behaviour of a TGA G restricted under a strategy f is defined by the notion of *outcome* [21].

Definition 4 (Outcome). *Given a TGA G and a strategy f for G, the* outcome *$Outcome(q, f)$ of f from symbolic state q is the subset of $Runs(q, G)$ defined by:*

- $q \in Outcome(q, f)$,
- *if $\rho \in Outcome(q, f)$ then $\rho' = \rho \xrightarrow{e} q' \in Outcome(q, f)$ if $\rho' \in Runs(q, G)$ and one of the following holds:*
  1. $e \in A_u$,
  2. $e \in A_c$ *and* $e = f(\rho)$,
  3. $e \in \mathbb{R}_{\geq 0}$ *and* $\forall 0 \leq e' < e, \exists q'' \in Q$ *s.t.* $last(\rho) \xrightarrow{e'} q'' \wedge f(\rho \xrightarrow{e'} q'') = \lambda$
- *for an infinite run ρ, $\rho \in Outcome(q, f)$ if all the finite prefixes of ρ are in $Outcome(q, f)$.*

Example. A potential strategy for the Timed Game Automata in Figure 5 could be as follows:

- The controller attempts to take the delay transition `l1` $\xrightarrow{\delta=1}$ `l1` followed by `l1` → `l2`.
- The environment may force `l1` → `l3` in between. In that case
  1. `l3` → `l4`
  2. `l4` $\xrightarrow{\delta=1}$ `l4` (delay until $x = 1$)
  3. `l4` → `l2`
  4. `l2` $\xrightarrow{\delta=1}$ `l2` (delay until $x \geq 2$)
  5. `l2` → `goal`
- otherwise,
  1. `l2` $\xrightarrow{\delta=1}$ `l2` (delay until $x \geq 2$)
  2. `l2` → `goal`

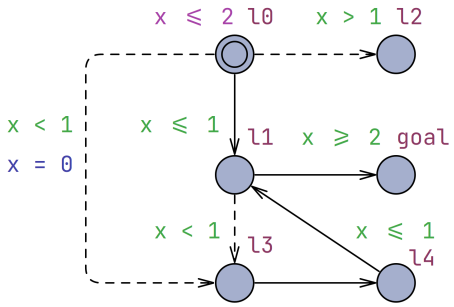


Fig. 5: Timed game automata with locations `l1`, `l2`, `l3`, `l4`, `l5` and `goal`, and the clock x

We assume that some controllable action must be taken in order to win the game, and that uncontrollable actions can only ruin the game, i.e. uncontrollable actions can not be forced to happen. As a result, a state with only uncontrollable actions will always be losing, even in the case that the game could be recovered, where the uncontrollable action is enforced as a consequence of some location invariant breaking.

A maximal run ρ is either an infinite run (disregarding infinite number of 0 delays) or a finite run ρ where either $last(\rho) \in W$ or if $\rho \xrightarrow{a}$ then $a \in A_u$, i.e. only uncontrollable actions are permissible from $last(\rho)$. Thus, a strategy f is winning from symbolic state q if all maximal runs in $Outcome(q, f)$ are in $WinRuns(q, G)$, and the state q is winning if there exists a strategy f from q in G. By $\mathcal{W}(G)$ we denote the set of winning states in G.

For timed reachability games, the computation of winning states is based on controllable predecessors of X . For this, the concept of safe, timed predecessors $Pred_t$ is needed. A state q will be in $Pred_t(X, Y)$ if we can reach $q' \in X$ from q by delaying, and we at no point in the path from q to q' reach Y. The formal definition of $Pred_t$ is[14]:

$$\begin{aligned} Pred_t(X,Y) = \{q \in Q \mid \exists \delta \in \mathbb{R}_{\geq 0} \text{ s.t. } q \xrightarrow{\delta} q', q' \in X \text{ and } Post_{[0,\delta]}(q) \subseteq \overline{Y}\} \\ \text{where } Post_{[0,\delta]}(q) = \{q' \in Q \mid \exists t \in [0,\delta] \text{ s.t. } q \xrightarrow{t} q'\} \text{ and } \overline{Y} = Q - Y \end{aligned} \tag{9}$$

If Y is a convex set, the computation of $Pred_t$ can be defined in terms of basic operations on zones[14]:

$$Pred_t(X,Y) = (X^{\swarrow} - Y^{\swarrow}) \cup ((X \cap Y^{\swarrow}) - Y)^{\swarrow} \tag{10}$$

$Pred_t$ can be be defined for unions of zones using the following distribution law[14]:

$$Pred_t(\bigcup_{D_g \in F_g} D_g, \bigcup_{D_b \in F_b} D_b) = \bigcup_{D_g \in F_g} \bigcap_{D_b \in F_b} Pred_t(D_g, D_b) \tag{11}$$

Algorithm 2 Symbolic On-The-Fly Algorithm for Timed Reachability Games

1: $Passed \leftarrow \{S_0\}$ **where** $S_0 = \{(l_0, \mathbf{0})\}^{\nearrow}$
2: $Waiting \leftarrow \{(S_0, a, S') \mid S' = Post_a(S_0)^{\nearrow}\}$
3: $Win[S_0] \leftarrow S_0 \cap (\{Goal\} \times \mathbb{R}^X_{\geq 0})$
4: $Depend[S_0] \leftarrow \varnothing$
5: **while** $((Waiting \neq \varnothing) \wedge (s_0 \notin Win[S_0]))$ **do**
6: $\quad e = (S, a, S') \leftarrow pop(Waiting)$
7: $\quad$ **if** $S' \notin Passed$ **then**
8: $\quad\quad Passed \leftarrow Passed \cup \{S'\}$
9: $\quad\quad Depend[S'] \leftarrow \{e\}$
10: $\quad\quad Win[S'] \leftarrow S' \cap (\{Goal\} \times \mathbb{R}^X_{\geq 0})$
11: $\quad\quad Waiting \leftarrow Waiting \cup \{(S', a, S'') \mid S'' = Post_a(S')^{\nearrow}\}$
12: $\quad\quad$ **if** $Win[S'] \neq \varnothing$ **then**
13: $\quad\quad\quad Waiting \leftarrow Waiting \cup \{e\}$
14: $\quad$ **else**
15: $\quad\quad Win^* \leftarrow Pred_t(Win[S] \cup \bigcup_{S \xrightarrow{c} T} Pred_c(Win[htbp]), \bigcup_{S \xrightarrow{u} T} Pred_u(T - Win[htbp])) \cap S$
16: $\quad\quad$ **if** $(Win[S] \notin Win^*)$ **then**
17: $\quad\quad\quad Waiting \leftarrow Waiting \cup Depend[S]$
18: $\quad\quad\quad Win[S] \leftarrow Win^*$
19: $\quad\quad Depend[S'] \leftarrow Depend[S'] \cup \{e\}$

Timed games can be solved on-the-fly with a symbolic extension to the algorithm proposed by [32], intertwining a forward computation of the simulation graph of the TGA with a backwards propagation of information on winning states [14]. The SOTFTR algorithm, as seen in Algorithm 2, utilises passed- and waiting-lists, containing symbolic states of the simulation graph that has already been encountered, and edges in the simulation graph that is to be explored, respectively. The set $Win[S] \subseteq S$ tracks the set of states that (at any given time) is known to be winning. $Depend[S]$ indicates the predecessors of S which must be added to the waiting-list whenever new information regarding $Win[S]$ is acquired. An edge $e = (S, a, S')$ where $S' \in passed$ is added to the set of predecessors for S' s.t. future information regarding additional winning states in S' may also be propagated back to S.

The correctness of the SOTFTR algorithm is given by Lemma 1 and Theorem 1, directly from [14].

Lemma 1. *The **while**-loop of algorithm SOTFTR has the following invariance property when running on a timed game automaton G:*

1. *For any $S \in Passed$ if $S \xrightarrow{\alpha} S'$ then either $(S, \alpha, S') \in Waiting$ or $S' \in Passed$ and $(S, \alpha, S') \in Depend[S']$*
2. *If $q \in Win[S]$ for some $S \in Passed$ then $q \in \mathcal{W}(G)$*
3. *If $q \in S - Win[S]$ for some $S \in Passed$ then either*
   - *$e \in Waiting$ for some $e = (S, \alpha, S')$ with $S' \in Passed$, or*
   - *$q \notin Pred_t[Win[S] \cup \bigcup_{S \xrightarrow{c} T} Pred_c(Win[T]), \bigcup_{S \xrightarrow{u} T} Pred_u(T - Win[T])]$.*

Theorem 1. *Upon termination of running the algorithm SOTFTR on a given timed game automaton G the following holds:*

1. *If* $q \in Win[S]$ *for some* $S \in Passed$ *then* $q \in \mathcal{W}(G)$*;*
2. *If* $Waiting = \varnothing$*,* $q \in S - Win[S]$ *for some* $S \in Passed$ *then* $q \notin \mathcal{W}(G)$

3 GP-GPU Framework CUDA

CUDA is a programming interface and extension of C and C++, developed by NVIDIA, that allows issuing and managing data-parallel computations specifically on NVIDIA *Graphical Processing Unit*s (GPUs) [34]. The CUDA interface discriminates between CPU and GPU computations, by viewing them as co-processing units. For the remainder of this thesis, we follow the precedence of referring to CPU and GPU as *host* and *device*, respectively. The device is viewed as a set of multiprocessors, each of which use a *Single Instruction, Multiple Threads* (SIMT) architecture. In this paradigm, each single sub-processor of a multiprocessor executes the same instruction at the same clock-cycle – although on different threads. The host invokes asynchronous device functions referred to as *kernels* that are executed in parallel by N threads.

3.1 Thread hierarchy

CUDA treats device threads in a hierarchy of abstraction levels. Threads can be operated on individually, but are also part of a *warp*, a collection of exactly 32 threads. Warps are further organised in equal sized *blocks* that reside on only one multiprocessor, where instructions are issued to threads using the SIMT architecture. CUDA also allows for generic groupings of threads.

The exact number of blocks and threads is determined at a kernel launch, e.g. a configuration with a grid of $(10, 3)$ blocks with $(32, 32)$ threads each, gives a total of 30 blocks, 1024 thread per block, resulting in 30720 threads. However, the maximal number of threads per block is 32×32, as a block can have at most 32 warps.

During the execution of a kernel, the instruction unit of the multiprocessor will issue a shared instruction to a given warp from the set of blocks residing on the corresponding multiprocessor per clock cycle. This is the core attribute of the SIMT architecture, achieving a high level of parallelism on the device for a computation where threads of a given warp agree on their execution path. However, it is important not to introduce a high degree of *thread divergence*, where different threads in a warp await different instructions to be issued from the instruction unit. This is due to warps being a static grouping, and that the threads of a warp will always be issued a shared instruction, such that when two different subsets of threads $a, b \in warp$ must execute different instructions, it will take additional clock cycles, i.e. a can not execute in parallel with b and must be executed sequentially.

3.2 Memory Model

Device threads may access multiple memory spaces during their execution. A thread has private local data, and each block has a shared memory space visible to all threads within it. Furthermore, all threads can access a global memory space that is visible throughout the device. As CUDA enables co-processing computations, the host memory is also worth consideration when representing the entire memory model. Figure 6 shows a high level overview of such memory model. Most importantly on the host side is the *pinned memory*, as this is directly involved in the process of data transfer between host and device.

Host memory is pageable by default, meaning infrequently accessed data can be moved to swap memory, and data residing in pageable memory can not be transferred to the device directly. Instead, page-locked memory (called pinned-memory) is utilised for transfers between host and device. Similarly, when transferring from device to host, the destination on host side also must be pinned memory. All kinds of communication between host and device – even transferring data from device to host – is managed and dictated by the host. The bandwidth of the data transfer is obviously hardware specific. The *NVIDIA RTX 3070 TI* consumer card has a bandwidth of 608.3 GB/S while the *NVIDIA A100* data centre card has a bandwidth of 1555 GB/S (these are only when utilising pinned memory).

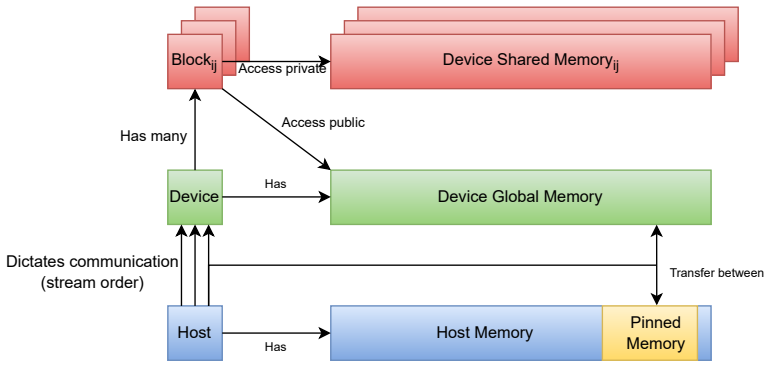


Fig. 6: The memory model, and which aspects can be interacted with by the host and device

Streams Communication between host and device occurs asynchronously through CUDA streams. Streams allow for controlling the ordering of operations (kernel invocation, transfers, etc.) within a single stream, and allows for running multiple operations in parallel, by using multiple streams. Multiple streams of operations can greatly enhance the overall throughput of a co-processing computations, especially if data transfer and communication is of major concern.

3.3 Notation

We shall in later sections utilise many CUDA related concepts when describing Algorithms and correctness. As stated previously, the host and device operate in a co-routine, with the host dictating device computations through kernel invocations. By **launch async x**, we denote the host asynchronously launching the specific kernel **x** on the device. Additional notation for CUDA related concepts is summarised in the following:

Thread Groupings CUDA operates on data with groups of threads with varying size. By T, W, B and G we denote the sets of a *singular thread*, a *warp*, a *block*, and a *grid* of threads, respectively, which operate in parallel.

Furthermore, by $\mathbf{count}(x, y)$ we denote $\lfloor \frac{|y|}{|x|} \rfloor$, e.g. $\mathbf{count}(W, B)$ for the number of warps in a block. We use $|x|$ as shorthand for $\mathbf{count}(T, x)$ for the generic group x, e.g. $|B|$ for the number of threads in a block and $|W|$ for the number of threads in a warp. In addition to the build in groupings, CUDA provides support for *custom groupings* that we denote CG_n for a custom group with n threads.

Hierarchical Identifiers All generic groupings are automatically assigned unique and incremental IDs in a hierarchical manor at kernel launch. Threads are ranked from 0 to $|W|$ in a warp and 0 to $|B|$ in a block, warps are ranked from 0 to $\mathbf{count}(W, B)$ etc. We will often utilise these unique IDs to determine what data each partition from a larger whole it should operate on. We denote by $\mathbf{rank}(x, y)$ the unique ID of x in y.

Group Partitioning Most involved is the device thread granularity that is utilised, i.e. how groups are partitioned and operated with. We denote by $\mathbf{parallel}(x, y)\ i = 0\ \mathbf{to}\ n$ generically partitioning y into $\mathbf{count}(x, y)$ groups, and loop over them in parallel s.t.

$$\begin{aligned}&\mathbf{parallel}(x, y)\ i = a\ \mathbf{to}\ b\ \ldots \\ &\equiv \\ &\mathbf{concurrently}\ \forall x \in y\ \mathbf{for}(i = \mathbf{rank}(x, y) + a;\ i < b;\ i = i + \mathbf{count}(x, y))\ \ldots\end{aligned} \tag{12}$$

For example, we write $\mathbf{parallel}(T, B)$ for partitioning a block into its singular threads, and $\mathbf{parallel}(W, B)$ for partitioning a block into its warp components. For any uneven groupings e.g. $\mathbf{parallel}(x, y)$ where $|x| \bmod |y| \neq 0$, y is partitioned into $\mathbf{count}(x, y)$ groups where the remaining idles. For example, $\mathbf{parallel}(CG_{10}, W)$ will partition a warp into 3 groups of 10 threads, with the remaining 2 threads in the warp idling ($\mathbf{count}(CG_{10}, W) = 3$).

We illustrate how this notation is used in Figure 7 with a custom group of 4 threads CG_4 operating on a shared array with 7 entries. Each value in the array

is to be squared, and the work distributed as equally as possible amongst the custom group. Using the defined notation, we would write

$$\begin{aligned} &\textbf{parallel}(T, CG_4)\ i = 0\ \textbf{to}\ 7\ \textbf{do} \\ &\quad \text{array[i]} = \text{array[i]}^2 \end{aligned} \tag{13}$$

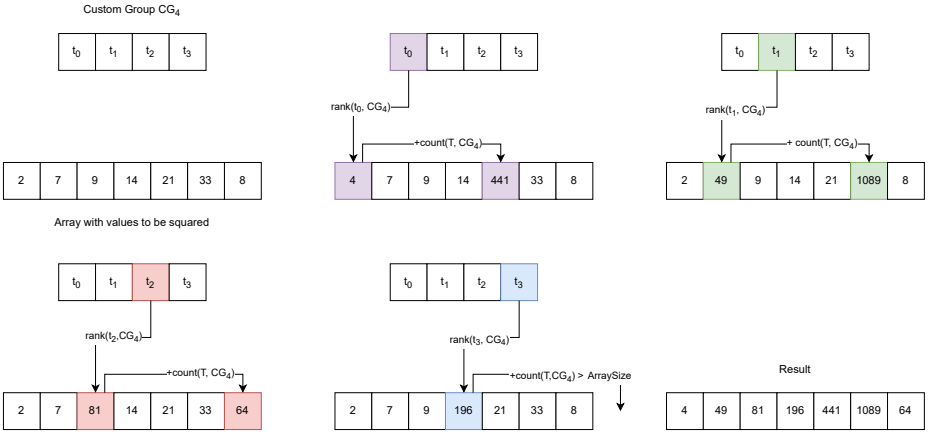


Fig. 7: Illustration of how work is distributed amongst a group of threads based on the size of the group and the rank of each thread

Communication & Consensus For some operations, some predicate must hold for all or at least one threads of the computation. We will write $g.\text{any}(p)$ denoting if the predicate p holds for any of the threads in the generic group g. Likewise, we write $g.\text{all}(p)$ denoting if the predicate p holds for all threads in the generic group g. This is only applicable for warps and custom groups that contain fewer threads than a warp i.e. 32.

4 GDBM

Our DBM library *GDBM* [3] was originally developed for the purpose of symbolic reachability analysis of timed automata. We have reworked and extended this library to now also support operations involved in controller synthesis for timed games. We will exemplify the theoretical benefit of computing these operations massively in parallel through complexity analysis. Unless stated otherwise, the derived complexity of the operations *always* assume an equal number of device threads used during the computation and DBM entries. The same is applicable when discussing partitioned operations on federations, i.e. we will assume that

the grid size (number of launched blocks) to be equal to the amount of DBMs in the federation.

4.1 Thread Mapping

The standard DBM operations from Bengtsson [11] and Behrmann et al. [8] as mentioned in Section 2.2 are in general computed using the threads of an entire thread block in GDBM. Some operations benefit from being computed on a varying level of device thread granularity – we will note this when discussing these operations in more detail. The block is utilised by mapping each thread within it to a unique subset of DBM entries. We will always require that there are fewer threads than the number of DBM entries, i.e. $|B| \leq n^2$, where n is the amount of clocks. As such, each thread will always be mapped to at least one DBM entry.

A thread is initially mapped to i, j values based on its $\mathbf{rank}(T, B)$ and the number of clocks s.t. $i = \mathbf{rank}(T, B)/n$, $j = \mathbf{rank}(T, B) \bmod n$. Each thread will then in parallel compute on its given DBM entry, dictated by the specific operation. If $|B| < n^2$, i.e. some threads are mapped to multiple entries, the values of i, j are offset in a three-step process, as seen in Algorithm 3.

Algorithm 3 Offsets the mapping of a thread to i, j values

```
1: procedure nextIndex(i, j, n, g)        ▷ g is a generic group of threads
2:     j ← j + (|g| mod n)                ▷ n is the number of clocks
3:     i ← i + (|g|/n) + (j ≥ n)
4:     j ← j − ((j ≥ n) * n)
5:     return i, j
```

While not explicitly illustrate, the computed offsets $(|g| \bmod n)$ and $(|g|/n)$ on Line 2 and 3 are *pre-computed* as both the total number of threads and the number of clocks are constant throughout the kernel execution. Many NVIDIA GPUs do not support integer division and modulo instructions, and instead relies on floating point conversions and reciprocal multiplication, which compiles to unnecessarily many instructions.

4.2 GDBM Closure

We illustrate the use of thread mapping concretely in the computation of DBM closure in Algorithm 4. Parallelisation of the all-pairs shortest path algorithm has been done before, e.g. [29,27]. Each thread evaluates its entry's bound for each step of the shortest path algorithm concurrently, and synchronises after each step, i.e. if any threads have updated their entry's bound then it is written to the main memory such that if any other thread accesses that bound, then they get the updated bound. Essentially, GDBM closure is computed sequentially, with $|B|$ entries being computed concurrently at a time, thus becoming $\mathcal{O}(n)$ as compared with the sequential $\mathcal{O}(n^3)$.

Algorithm 4 GDBM Close

```
1: procedure Floyd-Warshall(D)                          ▷ D is a DBM
2:     parallel(T, B) idx ← 0 to |B| do
3:         for k ← 0 to n do                            ▷ n is number of clocks (size of D)
4:             i, j ← getIndex(idx, n)
5:             repeat
6:                 D_{i,j} ← min(D_{i,j}, D_{i,k} + D_{k,j})
7:                 i, j ← nextIndex(i, j, n, B)
8:             until i ≥ n
9:             sync()
```

Correctness The synchronisation is pivotal to ensure correctness of the parallel closure algorithm. Our implementation synchronises all threads after each iteration of k, as to guarantee termination in n iterations (without synchronisation the algorithm will eventually compute the all-pairs shortest path but can not be guaranteed in any finite number of steps). After synchronising, if any threads have updated their entry's bound, it is written to the main memory and the updated value becomes accessible to all other threads. One may still be concerned that the order of updates occurring during any iteration of k somehow affects the correctness, e.g. an entry $D_{i,j}$ is updated depending on the value of $D_{i,k} + D_{k,j}$ such that the order in which these are updated affects the result. We argue that, as long as the k-loop occurs sequentially, the inner i, j-loops can be parallelised, as the order of i, j has no effect on the correctness. In the following, 3 identified cases for pairs of entries illustrates the correctness of GDBM closure. The cases and arguments are strongly based on our previous work [3], but has been reworded for clarity and conciseness:

No dependency indicates two DBM entries where neither is updated according to the other. As updates are only dependant on other entries where either index $= k$ for any given $k = 0 \dots n$, no dependency involves two entries $D_{i,j}$, $D_{l,m}$ where $i, j, l, m \neq k$. The order in which these are updated is obviously of no concern.

One-way dependency indicates a dependency between an entry $D_{i,j}$ and either $D_{i,k}$ or $D_{k,j}$, i.e. the two entries that are part of the update $D_{i,j} = min(D_{i,j}, D_{i,k} + D_{k,j})$. While this initially looks to be a case where the order of update matter, unfolding the supposed update of $D_{i,k}$ ($D_{k,j}$, respectively) shows that $D_{i,k} = min(D_{i,k}, D_{i,k} + D_{k,k})$ ($D_{k,j} = min(D_{k,j}, D_{k,j} + D_{k,k})$, respectively). As the diagonal of a DBM is always 0, the values of $D_{i,k}$ and $D_{k,j}$ are never updated, thus making the order irrelevant.

Two-way dependency indicates two entries in the DBM where both are dependant on the other. This is of no concern for similar reasons as the case for one-way dependency. Furthermore, for this case to even be present, the two entries must have either index $= k$, and share both column and row with the other. In other words, this dependency is only present for the case that *both* entries are $D_{k,k}$, making it obviously of no concern.

4.3 GDBM Relation

The relation operation relates to inclusion checking described in Section 2 as a two-way inclusion check between a pair of DBMs D, D', producing a relation being either *subset*, *superset*, *equal* or *different*. Computing the relation between two DBMs is one of the operations that can benefit from a higher thread granularity, i.e. computing on a warp rather than block level abstraction. This is changed from the previous version of our DBM library from [3] where an entire thread block was utilised, with the benefit of now being able to terminate early. For the sake of self-containment, we now re-state how relation was computed from our previous work (with minor modifications). While not explicitly stated (as the notation slightly differs from this thesis), the procedure still utilises a similar thread mapping as has previously been described.

Relation Computing relation between two DBMs is easily parallelisable, as the operation only requires reading values from both DBMs. Each entry is compared using a $\geq$ and $\leq$ comparison on each of their corresponding matrix entries, as seen on Line 6 and 7 in Algorithm 5, such that if all entries of D are $\leq D'$, then we can deduce a subset relation. All entries are equally distributed between all threads to maximise concurrency.

Afterwards, each thread knows the result only from the entries it has checked, and thus we need to form a consensus between threads, as shown on Line 10 and 11. Consensus synchronises all threads in a kernel, and returns *true* if *all* threads agree on the value being *true*, otherwise *false*. This mimics the $\wedge$ operation of Line 6 and 7. Computing relation in parallel gives a complexity of $\mathcal{O}(1)$.

Algorithm 5 GDBM Block Relation

```
1:  procedure Relation(D, D')                          ▷ D, D' are a DBMs
2:      parallel(T, B) idx ← 0 to |B| do
3:          i, j ← getIndex(idx, n)
4:          sub, super ← true
5:          repeat
6:              sub ← D_{i,j} ≤ D'_{i,j} ∧ sub
7:              super ← D_{i,j} ≥ D'_{i,j} ∧ super
8:              i, j ← nextIndex(i, j, n, B)
9:          until i ≥ n
10:         sub ← consensus(sub)          ▷ returns true if all threads provides true
11:         super ← consensus(super)      ▷ returns true if all threads provides true
12:         return ⟨sub, super⟩
```

Warp Relation While good results was reported from the block relation computation, we noticed a potential limitation in being incapable on terminating early in the case of contradicting relation on DBM entries, e.g. for DBMs D, D'

and there are entries such that $D_{i,j} > D'_{i,j}$ and $D_{l,m} < D'_{l,m}$, we can conclude early that the relation should be *different*.

Computing the relation on a warp, as opposed to a block, allows us to utilise the very fast intra-warp communication **warp.any()**. The procedure shown in Algorithm 6 utilises mostly the same procedure as the block relation algorithm, but allows us to check 32 entries at a time, and terminate once counter evidence is found, as oppose to reading all entries of the DBM.

Algorithm 6 GDBM Warp Relation

```
1: procedure Warp_Relation(D, D', n)                ▷ DBMs and number of clocks
2:     parallel(T, W) idx ← 0 to |W| do
3:         i, j ← getIndex(idx, n)
4:         repeat
5:             if warp.any(D_{i,j} < D'_{i,j}) then
6:                 repeat
7:                     if warp.any(D_{i,j} > D'_{i,j}) then
8:                         return different
9:                     i, j ← nextIndex(i, j, n, W)
10:                until i ≥ n
11:                return subset
12:            else if warp.any(D_{i,j} > D'_{i,j}) then
13:                repeat
14:                    if warp.any(D_{i,j} < D'_{i,j}) then
15:                        return different
16:                    i, j ← nextIndex(i, j, n, W)
17:                until i ≥ n
18:                return superset
19:            i, j ← nextIndex(i, j, n, W)
20:        until i ≥ n
21:        return equal
```

The obvious benefit of this re-implementation has been the added possibility of early termination. A more subtle benefit, is that this should provide better device occupancy, as threads have more work to do per relation computation. With block relation, each thread had fewer entries to compare, but required a synchronisation barrier to form a consensus, which significantly slows down the computation. This should also result in being able to compute many more relations with the same number of threads, under the assumption that the computation will often terminate early. Regardless, both the sequential Algorithm described in Section 2.2 and GDBMs implementation has a complexity of $\mathcal{O}(n^2)$, as it is essentially $\mathcal{O}(\frac{n^2}{\mathbf{count(T,W)}}) = \mathcal{O}(\frac{n^2}{32}) = \mathcal{O}(n^2)$, however, with a best-case complexity of $\Omega(1)$.

4.4 Intersection

Thread mapping and consensus as a concept is similarly used when computing the intersection between two DBMs. The procedure is listed in Algorithm 7. Consensus is reached (in principle) on the block shared variable *changed* that indicates whether the constraints in D have changed. If not, there is no need for closing the DBM, as it would already be on canonical form. The DBMs do not intersect if D becomes *empty*. As the operation may need to recompute the canonical form of the DBM, the complexity of the operation is similarly $\mathcal{O}(n)$. We utilise an optimisation not depicted in the algorithm, by continuously keeping track of the DBM entries $D_{i,j}$ that are changed. We can thus compute the canonical form based only on these changed entries, i.e. pre-selecting values of k (based on the changed entries) rather than looping over all values of k from $0 \dots n$.

Algorithm 7 GDBM Intersection

```
 1: procedure Intersection(D, D')                                  ▷ DBMs
 2:     block shared changed ← false
 3:     parallel(T, B) idx ← 0 to |B| do
 4:         i, j ← getIndex
 5:         repeat
 6:             if D'_{i,j} < D_{i,j} then
 7:                 D_{i,j} ← D'_{i,j}
 8:                 changed ← true
 9:             i, j ← nextIndex(i, j, n, B)
10:         until i ≥ n
11:     sync()
12:     if changed then
13:         Close(D)
14:     return D
```

The procedure can trivially be modified to only compute whether or not the DBMs have an intersection in $\mathcal{O}(1)$ time by disregarding the close operation on Line 13, without actually computing the intersection. We will later use this as a subroutine in larger and more expensive operations.

4.5 Restrict

While the restrict operation follow the tendency of utilising a full thread block, the main body of the computation has no need for any thread specific data, as it only relies on the bound passed as parameter. The computation can be seen in Algorithm 8. It checks whether the bound to be added will produce an empty DBM on Line 3. If not, and if the bound is tighter than what the DBM already holds, we update the bound. The DBM must be closed as a consequence of updating the bound, as the restrict break the canonicity otherwise. It is only

in restoring the canonical form subroutine that the device threads are properly utilised in parallel.

Algorithm 8 GDBM Restrict

```
1: procedure Restrict(D, n, (x − y ~ m))
2:     parallel(T, B) idx ← 0 to |B| do
3:         if D_{y,x} ≤ ¬(~ m) then
4:             D = ∅
5:         else if (~ m) < D_{x,y} then
6:             sync()
7:             D_{x,y} = (~ m)
8:             Closeij(D, n, x, y, idx)
9:         return D
10: procedure Closeij(D, n, x, y, idx)
11:     i, j ← getIndex(thread, n)
12:     repeat
13:         D_{i,j} ← min(D_{i,j}, D_{i,x} + D_{x,y} + D_{y,j})
14:         i, j ← nextIndex(i, j, n, B)
15:     until i ≥ n
16:     return D
```

Restoring the canonical form of the DBM, after updating a constraint utilises a special form of the all-pairs shortest path algorithm. The DBM is already assumed to be in canonical form before the constraint was updated. The shortest path has been shortened between the entries of the new bound denoted $D_{a,b}$, so we only need to recompute potential shortest path on the form $D_{i,j} = min(D_{i,j}, D_{i,a} + D_{a,b} + D_{b,j})$. This is essentially similar to computing all-pairs shortest path with a pre-selected k, omitting the otherwise needed loop. As this is the case, the complexity of the entire restrict procedure becomes $\mathcal{O}(1)$.

4.6 Extrapolation

GDBM has support for the four types of extrapolation named in Section 2.2. The approach follows the thread mapping established earlier and utilises an entire thread block for the computation. For the sake of conciseness, we only show the procedure for LU^{+}_{extra}, as it is the most involved while the others are similar. It can be seen in Algorithm 9.

Extrapolation unfortunately requires 2 synchronisation points, as the results found in the *repeat-until* loop from Line 4 is required in updating based on upper bounds from Line 11, that is again required before recomputing the canonical form. While this limits the speed of computation in practice, the complexity of the operation is still $\mathcal{O}(n)$ through the need of recomputing the canonical form of the DBM, while the remaining of extrapolation is $\mathcal{O}(1)$. The call to *CloseLU* is a specialised computation of the canonical form on only clocks that have a corresponding lower or upper bound from the input parameter.

Algorithm 9 GDBM LU_Extra$^+$

```
1: procedure LU_Extra+(D, [L_0,···,L_n], [U_0,···,U_n])   ▷ DBM, lower and upper bounds
2:     parallel(T, B) idx ← 0 to |B| do
3:         i, j ← getIndex(idx, n)
4:         repeat
5:             if i ≠ j ∧ i > 0 then
6:                 if D_{i,j} > L_i ∨ D_{0,i} < −L_i ∨ D_{0,j} < −U_j then
7:                     D_{i,j} ← ∞
8:             i, j ← nextIndex(i, j, n, B)
9:         until i ≥ n
10:        sync()
11:        if idx < n then
12:            if D_{0,idx} < −U_{idx} then
13:                if U_{idx} ≥ 0 then
14:                    D_{0,idx} ← −U_{idx}
15:                else
16:                    D_{0,idx} ← (0, ≤))
17:        sync()
18:    CloseLU(D, [L_0,···,L_n], [U_0,···,U_n])
```

GDBM makes no discrimination whether to extrapolate as a global or location-based zone abstraction [7], but only provides the functionality to compute this given the bounds. The intended extrapolation is determined by the symbolic verification algorithm that will interact with GDBM.

4.7 Shortest-Path Reduction

Computing the shortest-path reduction in GDBM differs slightly from the procedure established in Section 2.2 and [11,30]. As opposed to searching for redundant edges, they are initially assumed to be redundant and we search for evidence of the contrary.

The procedure is listen in Algorithm 10. The computation occurs on a thread block that is further partitioned into custom groups of size n (CG_n) to find equivalence classes (zero cycles) on Line 5. Threads in a custom group shares the set P that is used to contain *all* nodes in an equivalence class, while the entire thread block shares the set Eq used to keep track of the leaders (lowest node in P according to assumed ordering) of each equivalence class. Each custom group is assigned a node i in the graph that it will search for zero cycles from. Each thread in the custom group is further assigned to every other node j in the graph, and will evaluate concurrently if there is a zero cycles between i and j (Line 6) and add it to the set F. If the equivalence class contains multiple nodes, it indicates a zero cycles between the nodes in P. Edges in a zero cycles are connected in an incremental manner in the resulting DBM D_m on Line 11. We use Next(P, i) as a shorthand for the smallest value $i' \in P, i < i'$. In the case

that no $i < i'$ is found, i' becomes the smallest value $i' \in P$, in order to complete the loop.

Synchronisation is needed on Line 12 to guarantee all equivalence classes has been found, and that their leaders have been elected. The straightforward procedure *Reduce* is then computed across all threads in the block, where only the leaders of equivalence classes are considered when finding edges that should be part of D_m in a similar approach to [30].

Algorithm 10 GDBM Shortest-Path Reduction

```
 1: procedure Mingraph(D)
 2:     block shared Eq ← ∅, D_m ← ∅
 3:     parallel(CG_n, B) i ← 0 to n do
 4:         group shared P ← {i}
 5:         parallel(T, CG_n) j ← 0 to n do
 6:             if D_{i,j} + D_{j,i} = 0 then
 7:                 P ← P ∪ {j}
 8:         if i is smallest in P then
 9:             Eq ← Eq ∪ {i}
10:         if |P| > 1 then
11:             connect i to Next(P, i) in D_m          ▷ Assumes loop around
12:     sync()
13:     Reduce(D, Eq, D_m)
14:     sync()
15:     return D_m
16: procedure Reduce(D, Eq, D_m)
17:     parallel(T, B) i, j ∈ Eq do
18:         if ∀k ∈ Eq, k ≠ i, j : D_{i,k} + D_{k,j} > D_{i,j} then
19:             Connect i → j in D_m
```

Interestingly, the reduction algorithm described in [11] utilises more optimised nested loops (i from 0 to n, j from $i+1$ to n) than our implementation (i from 0 to n, j from 0 to n) for finding equivalence classes. A sequential computation can easily keep track of nodes to be elected leaders of their equivalent classes as part of the loops, while a parallel computation can not – e.g. if we partition generic groups to nodes i and have them search from i **to** n for equivalence classes, there is much added overhead in communication and reaching consensus regarding equivalence classes and finding the leader of these, as only the exact group that is assigned the correct leader will initially find this as part of the zero cycle. The essence of this is that a sequential computation has the benefit of being able to skip certain comparisons, e.g. if for $i = 0, j = 2$ it is found that (i, j) is an equivalence class, there is no need to check for the synonymous case of $i = 2, j = 0$. Utilising the worse structure in our implementation allows for better parallelability – partitioning the block into n groups of n threads results in all computations occurring concurrently, with the only added overhead

in atomic insertions into sets P and Eq and the synchronisation point. With perfect partitioning, each thread will have $\mathcal{O}(1)$ work. The *Reduce* procedure is sequentially $\mathcal{O}(|Eq|^3)$ but is parallelisable in a similar manner to computing closure, thus becoming $\mathcal{O}(|Eq|)$.

4.8 Federation Operations

GDBM is developed to primarily operate on federations – represented in memory as a linked list of DBMs – rather than singular DBMs, in order to achieve a higher degree of device occupancy and thus concurrency. The intended interaction with symbolic verification algorithms is to operate on *every* zone associated with a discrete state at once. To achieve this, federations can be operated on with varying degrees of device thread granularity, i.e. varying size and abstraction in the grouping of threads (grid, block, warp), depending on the specific operation.

We utilise the IDs assigned to generic groups to associate them with a unique DBM in the federation, and compute the DBM operation concurrently. As an example, Algorithm 11 shows how the canonical form of all DBMs in a federation are computed concurrently across a number of blocks. Based on the group IDs, this is trivially extended to all other DBM operations with varying generic groupings.

Under the assumption that there are z DBMs in F, and that there are an equal number of blocks in the grid, as there are DBMs in the federation, the complexity of this operation is $\mathcal{O}(z+n)$, as each block must traverse at most z elements through the linked list to find their corresponding element, and compute the closure. Going forward, we will omit the traversal of the linked list as part of the complexity analysis, as this in practice is only a minor concern of the algorithms.

Algorithm 11 GDBM Federation Closure

```
1: procedure Fed_Close(F)                                   ▷ Federation
2:     parallel(B, G) idx ← 0···|G| do
3:         D ← F[idx]
4:         Floyd-Warshall(D)
```

Representing federations as linked lists may introduce problems common in concurrent computations in regards to insertion and deletion of new nodes in the linked list, as illustrated in 8. If two threads concurrently wants to remove nodes B and C from the same linked list, the first thread, will attempt to connect node A to node C (thus removing B) while the other thread will attempt to connect node B to D (thus removing C), which can result in the linked list being left disconnected. Our general approach is to handle this through lazy synchronisation, e.g. marking nodes as *logically* removed on-the-fly, only *physically* removing them from the linked list as part of a final cleanup after the operation has concluded, as proposed in [26, Chapter 9].

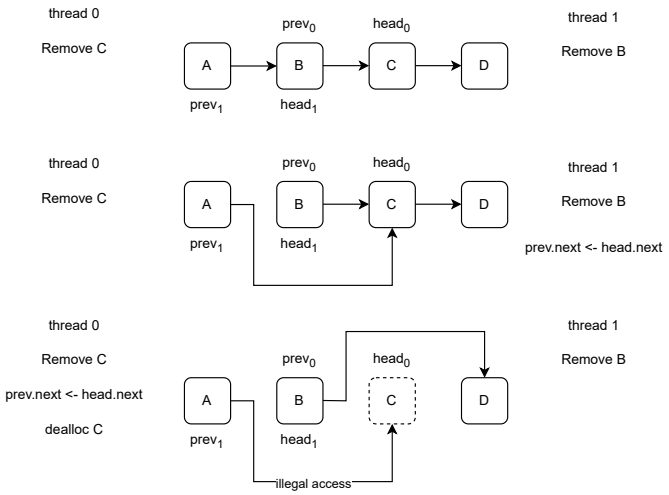


Fig. 8: Threads 0 and 1 working on the same linked list, each attempting to remove different nodes, resulting in an illegal memory access and a disconnected list

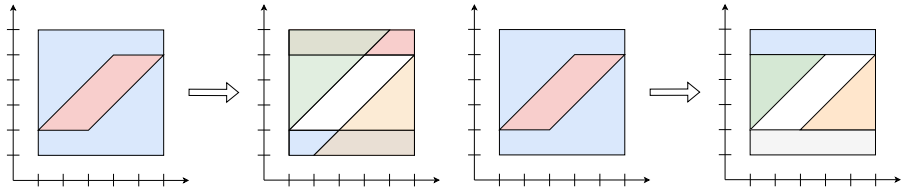
For the sake of comparison, GDBM also support naive federation operations with no partitioning of DBMs, i.e. computing the parallel DBM operations sequentially on a single block for all DBMs in the federation. The anomaly to this is federation intersection, that only implements this naive approach.

4.9 DBM Subtract

Multiple challenges arise from subtraction due to the non-convexity of the result of the operation. In a sequential computation, the primary concern is the rapid growth of the federation, while GDBM suffers additional adversity in not being able to predetermine the size of the result.

Dynamic Allocation While the size of the set resulting from subtraction is bounded by $n^2 - n$ DBMs, assuming this a priori quickly becomes too large to fit in memory. This is problematic as while CUDA does provide a built-in dynamic device allocator, it is well known to perform poorly, almost certainly forfeiting any speed-ups gained from computing the operation massively in parallel. Dynamic device allocation is an active research topic, e.g. [24,36]. We instead opt for circumventing this issue entirely, by preemptively creating a device memory pool of DBMs. The memory pool is held as a linked list of DBMs, with a global device pointer to the head of the list. Dynamically allocating a new DBM thus becomes a simple process of moving the global pointer as to disconnect the head

from the rest of the list. The drawback of doing this in parallel is the necessity of a locking mechanism, in our case handled through a simple spinlock scheme with a tiny critical section (compiles down to < 10 instructions). Furthermore, this introduce some overhead on startup in having to preemptively allocate the memory and connect the nodes in a linked list.



(a) Non-disjoint DBM subtract (b) Disjoint DBM subtract

Fig. 9: Illustration of different subtract variations, subtracting the inner (red) zone from the outer (blue) zone

Algorithm 12 GDBM DBM Subtraction

```
1: procedure Disjoint_Subtract(D, E)                                   ▷ DBMs
2:     block shared fed ← ∅
3:     for (i − j ~ m) ∈ E_m do        ▷ From reduced DBM, order based on heuristic
4:         if i ≠ j ∧ D_{i,j} < E_{i,j} then
5:             if D_{j,i} < ¬E_{i,j} then        ▷ D becomes empty, add remaining to fed
6:                 fed ← fed ∪ D
7:                 break
8:             block shared D′ ← mempool.next()          ▷ next dbm in mempool
9:             D′ ← D                                                  ▷ Copy D
10:            D′ ← Restrict(D′, ¬E_{i,j})
11:            D ← Restrict(D, E_{i,j})
12:            fed ← fed ∪ D′
13:    return fed
14: procedure Nondisjoint_Subtract(D, E)                               ▷ DBMs
15:    block shared fed ← ∅
16:    if D ∩ E = ∅ then
17:        return fed ∪ D
18:    parallel(W, B) i, j ∈ E_m do
19:        if D_{i,j} < E_{i,j} then
20:            warp shared D' ← mempool.next()
21:            D' ← D
22:            D' ← Restrict_warp(D′, ¬E_{i,j})      ▷ restrict for warp instead of block
23:            fed ← fed ∪ D′
24:    return fed
```

Disjoint Subtract GDBM implements multiple variations of the subtract computation, such as using reduced DBMs and keeping the resulting federation disjoint. It is hypothesised in [20] that it may be beneficial for the overarching verification algorithm to keep DBMs in the result set disjoint. An example of the discrepancy between in keeping DBMs disjoint is illustrated in Figures 9a and 9b.

Algorithm 12 lists the disjoint computation using reduced form (using all constraints is a trivial altercation). Unfortunately, the disjointedness requires an almost sequential computation. The only added parallelism stems from the **Restrict** subroutine from Line 10 and selecting the next constraint according to the heuristic function (sequentially is $\mathcal{O}(n^2)$, can be done in constant time in parallel). The ordering mentioned on Line 3 is the efficient heuristic from [20], i.e. the smallest value of $|E_{i,j}| - |D_{i,j}|$ in each iteration of the loop.

Non-disjoint Subtract Disregarding the disjointedness leads to much more potential for parallelism, as each restrict no longer requires the previous to be computed. Algorithm 12 lists the procedure for computing this form of subtraction using the reduced DBM form (again only trivial altercation is needed to use all constraints). Each constraint can now be partitioned amongst warps in a block instead of looping through this sequentially on a block. Computing on warp rather than block level necessitates a change in the call to **Restrict** on Line 22 – it is a trivial altercation to the procedure from using a block to using a warp. Only the locking scheme associated with using the memorypool on Line 20 and atomically inserting D' into the result set on Line 23 limits the parallelism.

While disregarding the disjointedness brings more potential for parallelism, the size of the result may become larger than it would otherwise. This stems from the disjoint algorithm being able to terminate earlier, i.e. when the selected constraint according to the heuristic would produce an empty DBM. If our assumption in regards to complexity analysis is consistent, and we thus assume that $\mathbf{count}(W, B) = n^2$ (a warp per constraint in E_m), the complexity of the non-disjoint subtraction is $\mathcal{O}(1)$. This assumption only holds in exceedingly rare cases in practice, and the derived complexity is a great exaggeration that poorly reflect the true nature of the operation. The number of constraints each warp must go through still scales by n^2. Similarly, computing restrict on a warp breaks the assumption of having a thread per DBM entry. The more conservative analysis thus deems the operation to be $\mathcal{O}(n^4)$. However, even with a theoretical worse complexity than the disjoint DBM subtract operation, the non-disjoint operation has more benefit from being parallelisable to a greater extend.

Naive Subtract For the sake of being able to better gauge the effect of different subtract strategies, GDBM supports a naive subtract method as well, where no heuristic function is used for the ordering of constraints, the full DBMs are used as opposed to reduced ones, and the resulting set is kept disjoint.

Federation Subtract Subtracting federations is, for the most part, a straight forward extension of subtracting DBMs. Similarly to how federation closure is computed in Algorithm 11, the federation that is being restricted is partitioned into singular DBMs across many blocks in the grid. For subtracting $F - F'$, each $D \in F$ is partitioned across blocks and each block computes $D - D'$ **for all** $D' \in F'$. This procedure is listen in Algorithm 13. Each block must continuously manage a partial result set (P and R on Line 7-8) until the partial set is added to the globally shared result on Line 9.

Algorithm 13 GDBM Federation Subtract

```
1:  procedure Federation_Subtract(F, F')                    ▷ Federations
2:      global shared result ← ∅
3:      parallel(B, G) idx ← 0 to |G| do
4:          block shared R ← {F[idx]}, P ← ∅
5:          for E ∈ F' do
6:              for D' ∈ R do
7:                  P ← P ∪ Subtract(D', E)
8:              R ← P, P ← ∅
9:          result ← result ∪ R                  ▷ Concurrent union across blocks
10:     return result
```

Concurrent Union Implementation wise, adding to the global result on Line 9 of Algorithm 13 is much more involved than what is indicated in the algorithm, as the federation invariant must be enforced, i.e. no DBM in the federation is subsumed by another.

The intuition is a continuous union across blocks until the entire result has been reached, as illustrated in Figure 10. Specifically, each block will try to write to some global memory space. If this is empty, the block will simply write its federation F in its place. Otherwise, the block will load the partial result into local block memory (thus removing it from global), combine it with its own local F (which involves removing subsumed DBMs) and re-attempt to write to global memory. This more detailed procedure is similarly illustrated in the lower half of Figure 10. A block will finish its part of this procedure whenever it writes to the global memory space and that space contains an empty set. The concurrency and thus device occupancy of this procedure diminishes over time as a consequence.

The removal of redundant DBMs whenever two federations are combined is computed concurrently on a single block. Given two federations F, F' that we want to union, we partition DBMs $D \in F$ across warps in the block, and compute the relation between D and all $D' \in F'$, marking D as redundant whenever $D \subseteq D'$ (and D' whenever $D' \subseteq D$). When a DBM is found to be subsumed by another, it is only *marked* redundant and only removed when all relations have been computed. Removal of DBMs on-the-fly would introduce conflicts similar to the *ABA* problem that plagues concurrent computing.

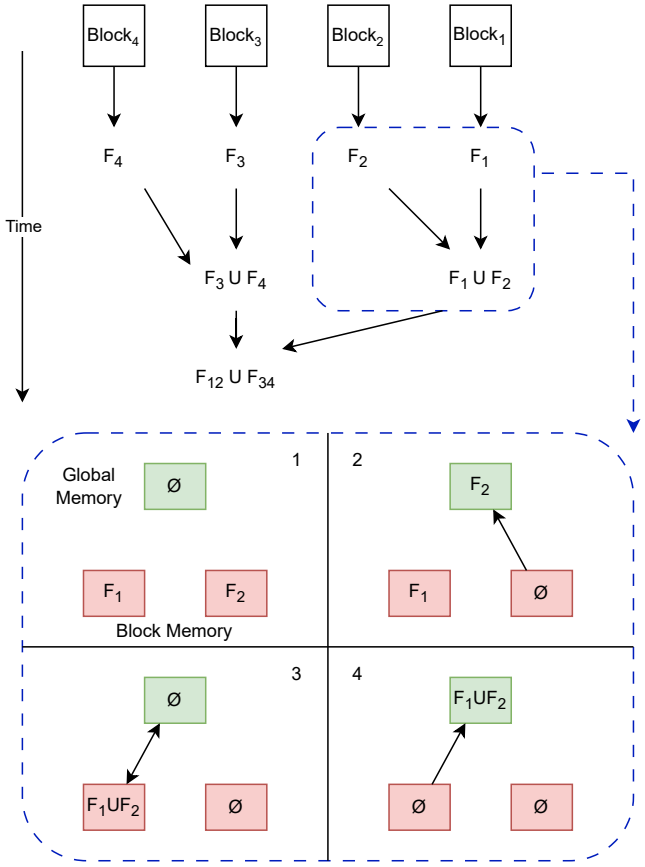


Fig. 10: Concurrent union computed across blocks. Each block begins with its own locally stored federation denoted F_i that are continuously combined. Partial results are combined through access to global memory, as illustrated in the box with dashed blue lines in the lower half.

4.10 Merging Convex Sets

Reducing a federation through merging of convex unions follows the methodology described in Section 2.2 closely with little added parallelism. All pairs of DBMs in the federation are sequentially looped through on a single block when the merge is attempted, while the heuristic *IsMergable* and the subsequent subtractions and convex union on Line 4 in Algorithm 14 are computed concurrently. We utilise CG_2 (custom groups of 2 threads) for computing the heuristic, such that both i, j and j, i entries are computed concurrently with shared communication between the threads in the group. The heuristic checks for at least $n-2$ compatible opposite constraints on Line 15 (this is not mentioned in [17,18] but is in accordance with UPPAAL's DBM library UDBM). Additionally, we check that non-strict DBMs are not intersecting on Line 17, and if either DBM is subsumed by the other. Subsumptions would obviously not be present initially due to the federation invariant, but may be introduced as a consequence of earlier merges.

Algorithm 14 GDBM Merge

```
 1: procedure Merge_Reduce(F)
 2:     for all (D, D') ∈ F do
 3:         if IsMergable(D, D') then                          ▷ Heuristic
 4:             if ((D ⊔ D') − D) − D' = ∅ then
 5:                 Remove D and D' from F
 6:                 F := F ∪ {D ⊔ D'}
 7: procedure IsMergable(D, D')
 8:     block shared sup, sub ← true
 9:     block shared count ← 0
10:     parallel(CG_2, B) idx ← 0 to count(CG_2, B) do
11:         i, j ← getLowerTriangularIndex(idx, n)
12:         repeat
13:             if rank(T, CG_2) = 0 then
14:                 i, j ← j, i
15:             if CG_2.all(D_{i,j} = D'_{i,j}) ∧ rank(T, CG_2) = 0 then
16:                 count + +
17:             if CG_2.any(−D^+_{i,j} ≥ D'_{j,i}^+) then
18:                 intersects ← false
19:             sub ← (D_{i,j} ≤ D'_{i,j}) ∧ sub
20:             sup ← (D_{i,j} ≥ D'_{i,j}) ∧ sup
21:             i, j ← nextTriangularIndex(i, j, n, count(CG_2, B))
22:         until i ≥ n
23:     sync()
24:     return sub ∨ sup ∨ (intersects ∧ (count ≥ n − 2))
```

A final note is that the check $((D \sqcup D') - D) - D' = \varnothing$ is computed in place, i.e. with no additional memory usage. We may rewrite the check as $((D \sqcup D') - D) \subseteq$

D', where each resulting restriction (from subtract) can be compared against D' without being stored. For a more thorough merge, using more resources, the check can be rewritten as $(((D \sqcup D') - D) - D') \subseteq F$ when $D, D' \in F$.

4.11 Predt

Due to the nature of the operation, computing $Pred_t$ is a very expensive procedure. Parallelising the operation is conceptually rather straight forward and similar to a sequential implementation. The procedure is listed in Algorithm 15. The call to $Pred_t$ on Line 5 is shorthand for Equation 10, consisting of previously described DBM operations (e.g. subtract and past). The partial result computed in parallel on each block is added to a global result set on Line 6 in similar way to federation subtract.

Algorithm 15 GDBM Pred_t

```
1: procedure Pred_t(F_g, F_b)                                  ▷ Federations
2:     global shared result ← ∅
3:     parallel(B, G) idx ← 0 to |F_g| do
4:         D_g ← {F[idx]}
5:         P ← ⋂_{D_b ∈ F_b} Pred_t(D_g, D_b)        ▷ Concurrent intersection across blocks
6:         result ← result ∪ P                          ▷ Concurrent union across blocks
7:     return result
```

The intersection on Line 5 *may* be computed across multiple blocks, limited by the size of the federation F_g. The approach is similar to concurrent union across blocks described in Section 4.9, but using multiple addresses (one for each DBM in F_g) in global memory as opposed to just one. When all intersections have been written to the global memory space, we take the union of all intersections. Each DBM in F_g must have an associated address that is known a priori, as the implemented dynamic allocation through a memory pool only supports allocating DBMs. We denote the version of pred_t that partitions only $D_g \in F_g$ as *multi* $pred_t$ and the version that partitions both $D_g \in F_g$ and $D_b \in F_b$ as *super* $pred_t$.

4.12 DBM Hash Table

For the purpose of reducing the memory footprint of GDBM, we have implemented a DBM hash table such that identical DBMs can be shared rather than needlessly duplicated. We base this on similar structures found in Uppaal's DBM library *UDBM* [19], while closely following the state of the art work by *WarpCore* [28]. The DBM table is modelled as a *Struct of Arrays* in global device memory, consisting of *keys* (a hashed DBM), *references* (reference counting) and *values* (pointer to the DBM), as can be seen in Figure 11. The table is structured in such a way that the same index is correlated in all 3 arrays, i.e. $keys[x]$

corresponds to $references[x]$ and $values[x]$. Reference counting is introduced as a consequence of sharing DBMs, in order to determine when the DBM can be deallocated and removed from the table.

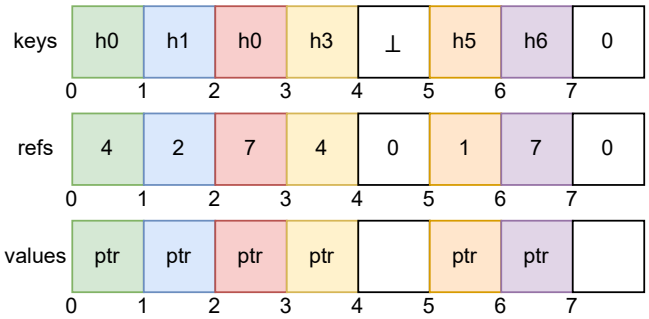


Fig. 11: DBM table consisting of keys (h, $\perp$, and 0 indicating a hash value, tombstone value, and empty slot), references, and values (pointers)

Collisions We utilise an *Open Addressing* strategy for dealing with hash collisions – that is, when attempting to insert a DBM D' whose key k' collides with another key k already placed in the table, if for the corresponding DBMs $D \neq D'$, we must compute a new index where k' is inserted. Using Open Addressing, this is resolved by a *probing strategy* that searches through candidate positions until an unused slot is found. WarpCore has developed the *Cooperative Probing Strategy* for device hash tables that utilises warps during the probe. This strategy is based on a *Linear Probing Strategy* for threads within a warp – threads in a warp has a fixed offset based on their id in the warp – and a *Double Hashing Strategy* for indexing the search of the entire warp. For example, if we are looking for an empty slot in the table, the warp is initially indexed through double hashing, and each thread in the warp is linearly offset from each other. If no empty slot is found, the double hashing strategy determines the next slot from which to continue the search.

We modify this probing strategy slightly, such that we also index warps based on linear probing – essentially offsetting the search by the size of a warp. Not utilising a double hashing strategy makes our probing much more sensitive to clustering but should have much improved cache performance. The strategy is illustrated in Figure 12.

Insertion, Deletion & Retrieval Insertion of a key-value pair into the storage structure is done on a warp level abstraction, where the outer linear probing scheme is used to determine the initial index of the warp. Each thread in a warp is offset from the initial probing index for its warp, and will check whether its

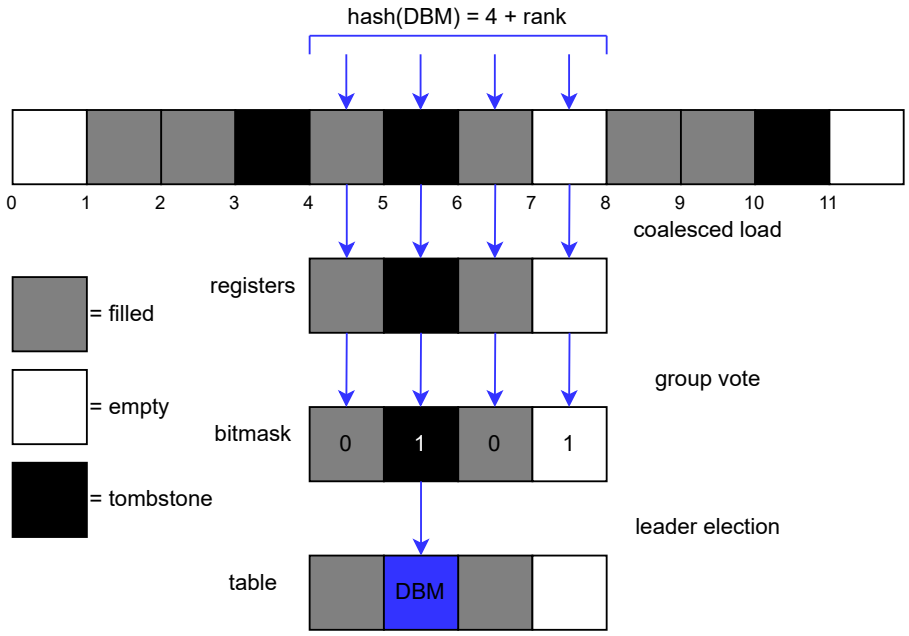


Fig. 12: The modified Cooperative Probing Scheme. Based on Figure 2 from [28]

assigned key slot is a potential candidate, determined by the slot being currently empty or holding a Tombstone value. This is communicated to the rest of the warp through intra-warp communication. If no potential candidate is found, the entire process is restarted. Otherwise, the smallest candidate slot index is selected. An atomic Compare And Swap operation is attempted on the selected candidate slot with the key-value pair. This may fail due to successful insertion by another thread in the meantime, and in that case, we select the next lowest potential candidate slot index and retry from there. If successful, the key and value are inserted and the associated reference count is incremented. As such, either a pointer to the newly inserted DBM, or a pointer to an already inserted identical DBM, is returned to the caller.

Deleting entries is accomplished by decrementing the reference count to the value. If the reference count reaches zero, a tombstone value overwrites the key slot and the DBM is deallocated.

The maximum size of the DBM table is limited to 2^{28} entries, for purpose that will become clearer when discussing the *federation table*. Normally, similar storage structures are implemented as an expandable table. As ours is fixed and statically allocated, GDBM will terminate entirely in the (albeit unlikely) scenario that $tableSize + 1$ entries are required. Each entry takes up 16 bytes, allowing for the table to require up to 4 GB of memory.

4.13 Federation Table

While federations are held as linked list during most operations, this become infeasible for operations with insertion and deletion of DBMs in the federation, due to the infamous ABA problem in concurrent computations. As such, federations are stored in a multi map structure consisting of *keys*, *dbmIdx* and *flags* as seen in Figure 13. Keys are acquired from a mapping of discrete states to natural numbers $f : L \to \mathbb{N}$ in an incremental manner, and placed in the table according to a hash value of the keys. The keys are placed in the federation according to the hashed value rather than the natural number in order to encourage more locality of federations, i.e. diminishing the chance that entries belonging to different federations being interleaved in the table. A federation is thus described by all entries that share a key. In practice, this structure acts as the PW list of Algorithm 1.

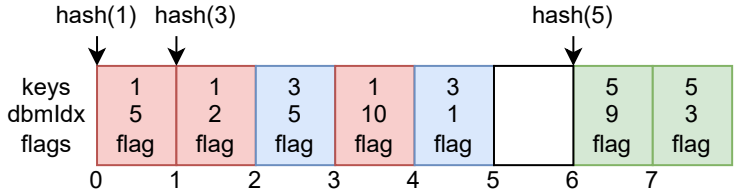


Fig. 13: Fed table consisting of keys, dbmIdx (for indexing in the DBM table), and flags

The dbmIdx is stored as a 32bit integer value, corresponding to the index of the zone in the DBM hash table. As the maximum size of the DBM hash table is limited to 2^{28}, we utilise the remaining 4 bits for flags, indicating the following:

- Is waiting
- Is passed
- Is losing (timed games)
- Is winning (timed games)

By limiting the size of the DBM table, and thus bitcram the flag indicators into the value field instead of using additional memory, we can contain each entry in the Fed table to exactly 8 bytes, which allows us to fully utilise atomic operations, making the table entirely lock-free and wait-free. Otherwise, the federation table is similar to the previously described DBM hash table as it utilises the same probing scheme, i.e. searching from an initial candidate entry according to the hashed value until the desired entry or an empty entry is found. Similarly, when removing entries from the federation table, a tombstone value is left in its place.

5 Asynchronous Co-Processing Reachability Engine

UPPAAL and UPPAAL Tiga have been extended to allow for integration of GDBM functionality and support for an asynchronous host-device verification of timed automata and synthesis for timed reachability games. This section will exclusively detail the integration into UPPAAL, i.e. symbolic reachability, as a prerequisite for discussing UPPAAL Tiga, due to reachability similarly being part of the synthesis for timed games. For a fairer comparison in later experiments, we have also implemented an asynchronous host exclusive symbolic reachability algorithms that we also describe briefly.

5.1 Host-Device

The asynchronous workflow between host and device is essentially modelled as a producer-consumer pattern, such that the host is responsible for discrete exploration and queuing jobs associated with continuous exploration, i.e. DBM operations, for the device. The respective host and device parts of the overall verification is shown in Algorithm 16 and 17. The verification is entirely dictated by the host, by asynchronously launching kernels on the device, and receiving data back into the *waiting* set, that stores discrete states (locations l) that are yet to be explored by the host. Once explored (through discrete transitions), the reachability kernel is asynchronously launched with source and successor locations, that the device uses to *grab* associated waiting DBMs from the combined passed-waiting (PW) list. The device computes new continuous states associated with discrete successor states, updates its PW list, and sends results back to the hosts waiting set.

Algorithm 16 Asynchronous symbolic reachability – Host

1: **procedure** $reachability_{host}(l_0, D_0, \phi)$
2: $waiting \leftarrow \{l_0\}$
3: Compute $(l_0, D_0)^{\nearrow}$ on device and save resulting DBM as D'
4: Insert (D', w) into $PW[l_0]$ on the device
5: **while** $\neg terminate$ **do**
6: **while** $waiting \neq \varnothing$ **do**
7: $l \leftarrow pop(waiting)$
8: **if** $l \vdash \phi$ **then return** *true*
9: $succ \leftarrow \{(l', g, r) \mid l \xrightarrow{g,r} l'\}$
10: **launch async** $forward_{device}(l, succ)$
11: **return** *false*

DBMs are exclusively stored on the device, as to limit the data that needs to be transferred back and forth. The device only needs discrete states in order to grab DBMs and their associated flags, as described in Section 4.13. More specifically, the device procedure *Grab* finds all waiting DBMs – in practice creating a

federation – for a given discrete state. The flags of these DBMs are then changed from *waiting (w)* to *passed (p)* to avoid unnecessary future re-computations. The device also utilises the *update* procedure to handle subsumption checking against all other DBMs in the PW list for a given discrete state. If the new DBM is subsumed by another it returns *false*, indicating it should not be explored further. DBMs from the PW list that are subsumed by the new DBM are removed. If the new DBM is not subsumed by any in the PW list, it is added and the discrete state parts are send to the hosts waiting set.

The host holds a user specified number of CUDA streams – dubbed *workers* – for communicating with the device. The work scheduled within a CUDA stream is totally ordered, while we desire a concurrent and asynchronous reachability analysis (and later synthesis of timed games), thus the need for multiple workers. The number of workers is limited by each requiring its own device memory space.

The algorithm runs until some termination criteria (denoted *terminate)* between the host and device is met. For this asynchronous workflow, it is important that neither the host nor the device terminate as soon as their respective waiting set is empty, as the other might still be in the midst of performing forward exploration. Termination is only fulfilled when both the host and device have no more work i.e. there are no more states to be explored, which is guaranteed due to the finite state space achieved through the use of extrapolation.

Algorithm 17 Asynchronous symbolic reachability – Device

```
1: procedure forward_device(l, succ)
2:     F_W ← grab(l)
3:     for all {(l', D') | (l', g, r) ∈ succ ∧ D ∈ F_W ∧ D' ← ((D ∩ [[g]])[r])^↗} do
4:         if D' ≠ ∅ ∧ update(l', D', {p, w}, w) then
5:             send(l', waiting)
6: procedure grab(l)
7:     F_W ← ∅
8:     for all (D, w) ∈ PW[l] do
9:         F_W ← F_W ∪ {D}
10:        PW[l] ← (PW[l]\(D, w)) ∪ (D, p)
11:    return F_W
12: procedure update(l, D, flags, f_n)
13:    for all (D', f) ∈ PW[l] ∧ f ∈ flags do
14:        if D ⊆ D' then
15:            return false
16:        else if D' ⊆ D then
17:            PW[l] ← PW[l]\{(D', f)}
18:        PW[l] ← PW[l] ∪ {(D, f_n)}
19:        return true
```

5.2 Exploiting Concurrency

The continuous state computation on the device is described in Algorithm 17 in a sequential manner for the purpose of simplicity and clarity. In practice, the computation proceeds concurrently similarly to how federation operations are described in Section 4.8. Instead of launching the forward procedure once on a single block, the host keeps track of how many DBMs are waiting for each state and launches the kernel with $(|waitingDBMs \in Source| \times |Successors|)$ blocks. Each launched block grabs a single unique DBM, and is only tasked with the forward exploration of a single successor state and a single DBM. As such, we compute many DBM operations in parallel on Line 3, while the computation of each of these DBM operations is also parallelised.

During the *update* procedure, each block attempts to check their newly computed DBM against the PW list for the successor state, finding whether it is subsumed by any DBM already in the PW list. In practice, we first check if said DBM is subsumed by another newly computed DBM for the same successor, and if not, then compare it to the PW list. If the newly computed DBM supersedes an element of the PW list, we remove it using lazy synchronisation, by unmarking its passed and waiting flags. Insertion into the PW list is only done once *every* block has finished their subsumption check, and removed their DBM or unmarked ones in the PW list that is superseded by it. If it gets added to the PW list, the successor state is send back to the hosts waiting set, as it should get further explored. Unmarked entries are also removed from the PW table.

5.3 Work Dependencies

The concurrent exploration may result in work being queued that should be mutually exclusive. The timed automata (with clocks, guards, and invariants omitted for simplicity) illustrated in Figure 14 can have undefined or incorrect behaviour if explored concurrently. For example, if `l1` and `l2` are forwardly explored concurrently, the transition `l1` → `l3` may be evaluated first, beginning the subsumption check and removing symbolic states on `l3`. Symbolic states reachable from `l1` and `l3` may overlap, such that a scenario may arise where the transition `l2` → `l3` is evaluated after symbolic states have been removed by the previous subsumption check, but before writing the zones that supersedes these, leading to undefined behaviour between the two processes.

Our solution to this is utilising CUDA stream events for a linearisation scheme for critical sections, similar in nature to a dependency graph. One can conceptualise this as a locking scheme, as this guarantees exclusive access to critical sections, although with the benefit of not using any system resources, as an agreed upon ordering of tasks emerges through the linearisation. The many workers (CUDA streams) that is utilised during reachability is used to asynchronously communicate between host and device. Tasks queued from a single worker is totally ordered, but utilising multiple workers gives no guarantee of the execution order between them. A stream event is completed when all preceding tasks from the worker has been completed. The critical section that is

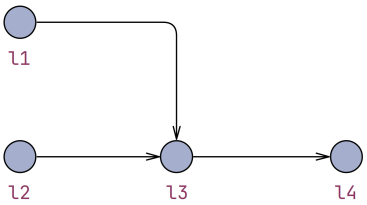


Fig. 14: A timed automaton with converging location on l3. Clocks, guards and invariants are omitted for simplicity

locked is always a discrete state and *all* of its successors (both for forward exploration in reachability analysis and later for back-propagation in synthesis of timed games). For the timed automata in Figure 14, we will only lock l1 and l3 when exploring l1 → l3, while locking all of l2, l3, l4 and l5 in Figure 15 when forwardly exploring l2.

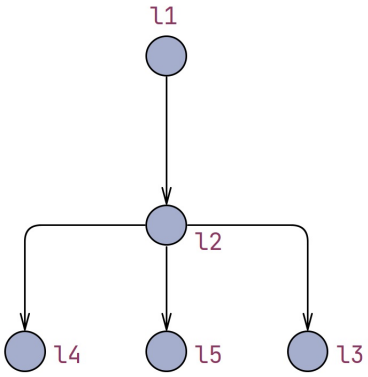


Fig. 15: A timed automata. Clocks, guards and invariants are omitted for simplicity

5.4 Exploration Order

The exploration order in the algorithm can be determined by how work is extracted from the waiting list. The waiting list is implemented as a queue giving an exploration order of breadth first search (BFS), but since the device works in parallel there is no guarantee of strict BFS. Using no guarantee of order the device will do its computations in parallel and adding to the hosts waiting list in whatever order it finishes its work. Having no order gives way for differing run times of the algorithm between different runs as some explorations of the state space might lead to exploring fewer states. Strict BFS can be implemented

by always adding work to the waiting list in the same order that it is retrieved from the device. This can lead to less concurrency as it has to wait for the strict ordering, but it will make the runs of the algorithm more uniform as it is guaranteed to explore the states in the same order, and will resemble the exploration order of UPPAAL.

Some care must be taken as to limit unnecessary duplication of forward exploration. For a timed automaton with a converging location, e.g. in Figure 14 with locations `l1`, `l2`, `l3` and `l4`. When both `l1` and `l2` explore forwardly, i.e. the transitions `l1` → `l3` and `l2` → `l3`, they would needlessly queue the exploration of `l3` twice when only once would be necessary. Discrete states are thus locked from having exploration tasks duplicated in the waiting set, which is then unlocked as soon as the forward exploration has begun.

5.5 Host Version

The algorithm has also been implemented as a multi-core host version. It still uses an asynchronous producer-consumer workflow with a single CPU core handling the discrete exploration in the same manner as Algorithm 16. Work entailing continuous exploration is added to a queue of jobs for workers (implemented as a lockfree queue from *boost*), as opposed to launching device kernels. The continuous exploration (DBM operations) is computed with a user specified amount of CPU cores (number of workers) similarly to the workflow of the host-device implementation. The successor locations that should be explored further are similarly send back to the waiting list of the producer. The amount of cores used for continuous exploration is obviously limited by the specific CPU, e.g. a CPU with 16 cores can use up to 15 cores for computing DBM operations concurrently.

6 Asynchronous Co-Processing Timed Games Engine

Extending UPPAAL Tiga to allow for integration of GDBM functionality is undoubtedly more convoluted than extending standard UPPAAL, while the guiding principles remain the same. The asynchronous workflow still employ a producer-consumer pattern with the host responsible for discrete exploration and queuing DBM operations for the device. As a quick note, in Section 2.3, we assumed semantically that uncontrollable actions can not be forced to occur as a consequence of a broken location invariant, i.e. that uncontrollable actions can only ruin the game. However, as was reported by [15], the UPPAAL Tiga [9] tool does in fact force an uncontrollable transition whenever an invariant is violated. As we develop an extension of UPPAAL Tiga, and because we want to verify correct results, implementation wise, we will do the same.

The host and device part of the overall synthesis is portrayed in Algorithm 18 and 19, respectively. The host and device must continuously communicate the *direction* of any given state, i.e. whether the state should be forwardly explored or propagated back to its predecessor, depending on the presence of winning information. Winning information is determined through the property ϕ, which

Algorithm 18 Asynchronous Timed Reachability Games – Host

```
1: procedure TimedGames_host(l_0, D_0, φ)
2:     if l_0 ⊨ φ then
3:         return true
4:     waiting ← {(l_0, forward)}
5:     Depend[l_0] ← ∅
6:     flag ← waiting
7:     Compute (l_0, D_0)^↗ on device and save resulting DBM as D'
8:     On device PW[l_0] ← (D', flag)
9:     while ¬terminate do
10:        while waiting ≠ ∅ ∧ ¬terminate do
11:            (l, direction) ← pop(waiting)
12:            succ ← {(l', g, r, win) | l --g,r--> l', win ← l' ⊨ φ}
13:            if direction = forward then
14:                launch async forward_device(l, succ)
15:            else if direction = backward then
16:                launch async backward_device(l, succ, Depend[l])
17:    return ∃(D, win) ∈ PW[l_0]
```

is assumed to only regard discrete information (e.g. a location is marked *Goal* without any timing constraints in ϕ). The direction of a state determines the operations, thus the kernel, that is asynchronously launched on the device. The forward kernel is especially similar to that of the UPPAAL extension, only differing in updating dependencies (Algorithm 19 Line 5), and re-queuing states in the case of winning information (Algorithm 19 Line 8). The procedures *grab* and *update* are the same procedures which are used in the GDBM enabled symbolic reachability algorithm. The backwards kernel initially seems complex, but for the most part only entails gathering respectively winning and losing zones from the PW list associated with different locations (Algorithm 19 Lines 10-12), and subsequently computing $Pred_t$. If location l turns out to be winning as a result from this procedure, all its predecessors must compute backward, as they may end up being winning as well (Algorithm 19 Line 16). The *terminate* criteria in the **while**-loops is extended to $\exists((D, win) \in PW[l_0]) \vee (waiting = \varnothing \wedge device\ idles)$, as to guarantee that the algorithm terminates in the case that winning information has reached the initial state, or both host and device are left with no more work to be done.

Correctness Under the assumption that our forward and backward kernels are correct, our asynchronous implementation is correct as it satisfies both Lemma 1 and Theorem 1 that similarly gives the correctness of the SOTFTR algorithm, mainly differing in the additional case of forwards and backwards kernels concurrently being computed on the device. The inner-most **while**-loop in Algorithm 18 Line 10 has a similar invariance to that of Lemma 1:

Algorithm 19 Asynchronous Timed Games – Device

```
1: procedure forward_device(l, succ)
2:     F_W ← grab(l)
3:     for all {(l', D', f) | (l', g, r, f) ∈ succ ∧ D ∈ F_W ∧ D' ← ((D ∩ [[g]])[r])↗} do
4:         if D' ≠ ∅ ∧ update(l', D', {p, w}, w) then
5:             send(l, Depend[l'])
6:             send((l', forward), waiting)
7:             if f = win then
8:                 send((l, backward), waiting)
9: procedure backward_device(l, succ, depends)
10:    F_win ← grabFlag(l, win)
11:    F'_win ← ⋃_{l_c ∈ succ} {Pred_c(D) | D ∈ grabFlag(l_c, win)}
12:    F'_loss ← ⋃_{l_u ∈ succ} {Pred_u(D) | D ∈ (grabFlag(l_u, ¬win) − grabFlag(l_u, win))}
13:    Win* ← Pred_t(F_win ∪ F'_win, F_loss) ∩ PW[l]
14:    if ∃{D | D ∈ win* ∧ update(l, D, {win}, win)} then
15:        for all prev ∈ depends do
16:            send((prev, backward), waiting)
17: procedure grabFlag(l, t)
18:    return {D | (D, f) ∈ PW[l] ∧ f = t}
```

1. For any $l \in PW$ if $l \xrightarrow{\alpha} l'$, either
   - $l' \in waiting$
   - $l' \in PW$ and $l \in Depend[l']$ or
   - a $forward_{device}(l', succ)$ kernel is currently being computed.
2. If $(D, win) \in PW[l]$ for some $l \in PW$, then $(l, D) \in \mathcal{W}(G)$ for the timed game automaton G.
3. If $D \in grabFlag(l, \neg win)$ for some $l \in PW$, either
   - $l' \in PW$ with $l \in Depend[l']$ and $l \in waiting$
   - a $backward_{device}$ kernel is currently being computed or
   - $D \notin Win^*$, i.e. (l, D) is not currently winning based on information on its successors.

Theorem 1 is also fulfilled, as the termination criteria is either when l_0 is found to be winning, or when neither the host nor device has any more work to do. Additionally, there is no concern of race conditions in regards to queuing and performing tasks. Whenever a task of back-propagation is queued either

1. No other back-propagation task is queued for that discrete state, in which case no issue arise
2. The discrete state is already queued for back-propagation, in which case we don't re-queue it, as the potentially new winnings zones are already inserted into the PW list for that state
3. A back-propagation task for that discrete state is currently being computed, in which case we queue it anew

6.1 Exploiting Concurrency

The forward exploration for timed games utilises concurrency similar to that described for reachability analysis in Section 5.2, i.e. by launching a block for each zone associated with the discrete state, and computing the forward exploration across these multiple blocks. Back-propagation can exploit concurrency with a similar methodology, as a block is launched for each discrete successor that computes that states' Pred_c and Pred_u. Once all blocks have completed their computations and their results gathered, we similarly compute Pred_t concurrently, i.e. we launch a block for each *good* DBM.

6.2 Exploration Order

The exploration order in the algorithm is well documented to have large ramifications for the synthesis time for any given model [10]. As with the exploration order for the extension of Uppaal, no guarantee can be given in terms of computation order, i.e. a task a can be launched before task b, but task b can complete before task a. A similar scheme which was utilised for reachability, can be used here to enforce a strict ordering of task, and enforce a similar ordering to Uppaal Tiga.

A lot of care is taken to minimise unnecessary duplication of forward exploration and back-propagation, through the use of directed locks on discrete states. Consider the case illustrated in Figure 16 with location `l1`, `l2`, `l3`, `l4`, with `l2` and `l4` containing winning information. Consider the ordering of:

1. exploring `l1` finding `l2` and `l3` with `l2` to be winning (thus queuing to back-propogate on `l1`),
2. exploring `l3` finding `l4` to be winning, thus queuing to back-propagate on `l3`,
3. finding `l3` to be winning as a result of back-propagation, thus queuing to back-propagate on `l1`.

In this computation order, `l1` would be scheduled to back-propagate information from its successors twice, which would obviously be detrimental as only one would be necessary. In the case that a state is both scheduled for back-propagation and forward exploration, we will always prioritise forward exploration, as this may lead to more winning information being discovered. Similarly, if both a location and its successor is marked for back-propagation, we will always prioritise the successor first, as this, once again, may lead to discovering more winning information, e.g. if both `l3` and `l4` in Figure 16 is marked for back-propagation. However there is a potential downside to this approach if e.g. after finding out that `l2` is winning this could be back-propagated to the initial state, we could then terminate early, but as Uppaal uses the approach of prioritising successors we do the same.

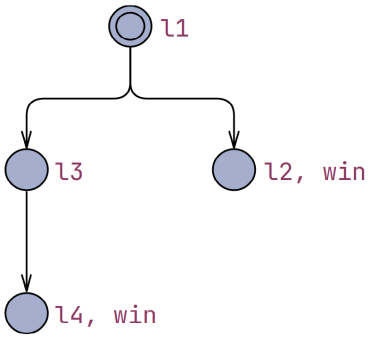


Fig. 16: Timed game automata with locations `l1`, `l2`, `l3`, `l4`, with `l2` and `l4` containing winning information. Clocks, guards and invariants have been omitted for simplicity

7 Experiments

Our experiments section documents the applicability of GPUs for computing DBM operations massively in parallel. We shall initially experiment on DBM operations in isolation, with comparisons to both our own multi-core CPU implementation (that distributes the operations over multiple cores, with no added parallelism within each computation), and against Uppaal's DBM library *UDBM*. We additionally conduct experiments on our asynchronous host-device implementations of symbolic reachability analysis and synthesis of timed games. These are compared against our own asynchronous multi-core CPU implementation and Uppaal, and against Uppaal Tiga, respectively.

The CPU and GPU which the experiments are conducted on can be found in Table 1. Reachability and Timed Games experiments are conducted on a number of case study models, an overview of which can be seen in Table 2 and 6, respectively.

	Name	Cores	Clock speed	Memory size	Memory bandwidth	Compute capability	Release year
GPU	NVIDIA RTX 3070 TI	48	1575 MHz	8 GB	608.3 GB/s	8.6	2021
CPU	Intel i7-11700KF	8	3.60 GHz	32 GB	-	-	2021

Table 1: The CPU and GPU used for the experiments

7.1 DBM/Federation Operations

While we have already reported on isolated DBM experiments in [3], the extension of, and subsequent re-implementations in GDBM, warrants conducting similar experiments once more. A specific DBM or federation operation is applied to a large set of DBMs or federations with varying federation size and DBM dimensions. DBMs and federations are generated by UDBM and GDBM in canonical form, except for experiments involving the closure operation.

Both libraries are used for generating data for these experiments, as the resulting DBMs differ in their properties stemming from the generating methods. GDBM initialises the first row to 0 and first column to random values within the range of 2 to 30 (all bounds are randomly strict or non-strict), and closes the DBM. This guarantees that all DBMs have root in the origin, i.e. that they all intersect to some extend, ensuring that we avoid trivial or degenerate cases in experiments for intersect, subtract and pred_t. UDBM uses a more sophisticated method for generating DBMs. Both the first row and column are initialised with random values within a range of 20 centred around 10, followed by closing the DBM. Based on randomness, some clocks in the DBM may be deactivated and diagonal constraints may be tightened in different ways. DBMs generated through this method are more scattered through the continuous state space, giving more variety in experiments such that they more accurately reflect DBMs found during reachability analysis or synthesis of timed games. Importantly, all configurations are run on DBMs generated from the same seed, such that all configurations uses the same DBMs for e.g. 16 dimensions.

The general structure of these experiments is comparing the performance of UDBM with both host and device computations of GDBM on 250000 DBMs with varying dimensions. For host computations, we run experiments with both a single CPU thread and 16 CPU threads, while device experiments are run with varying batch sizes for partitioning DBMs, ranging from 1 to 1024. In addition, the device experiments does *not* include overhead time in transferring DBMs, as this is also not part of the reachability analysis or synthesis of timed games. When applicable, we will compare different device methods for computing the same DBM operation, e.g. different subtract methods (disjoint vs. non-disjoint) and using either blocks or warps for computing the relation between DBMs. We conduct experiments on the following DBM and federation operations:

- DBM Operations
  - Close
  - Intersection
  - ExtrapolationLU
  - DBM Subtract
  - Shortest-Path Reduction
- Federation Operations
  - Approximate Relation
  - Federation Subtract
  - Pred_t

For the experiments whenever Host or Device is followed by a number in parentheses it specifies the amount of threads or the batch size, respectively, which were used in the experiments. As an example Host (1) and Device (1) refers to performing the experiments with 1 thread and a batch size of 1, respectively.

Close The performance of the different experiment configurations for computing the canonical form can be seen in Figure 17. Notably, we see that device computations with a batch size of 1 outperform UDBM and single threaded host computations on DBMs larger than 16 dimensions. Unsurprisingly, using larger batches for device computations generally results in better performance, although with diminishing returns going from a batch size of 256 to 1024. Comparing device computations with a batch size of 1024 to UDBM, we see a 68.2 times speedup, while comparing the same device configuration with the multi-threaded host computation shows a speedup of 8.8 times on 64 dimensions. While this may initially seem worse than the results we reported in our previous work [3], where we saw a 76 times speedup on close with 64 dimensions, the hardware utilised in those experiments are significantly more powerful than what is used in these experiments. Comparing the methods on the same hardware, we measure varying speedup between 1.35 times on 64 dimensions to 10.32 times on 32 dimensions in comparison to our previous work.

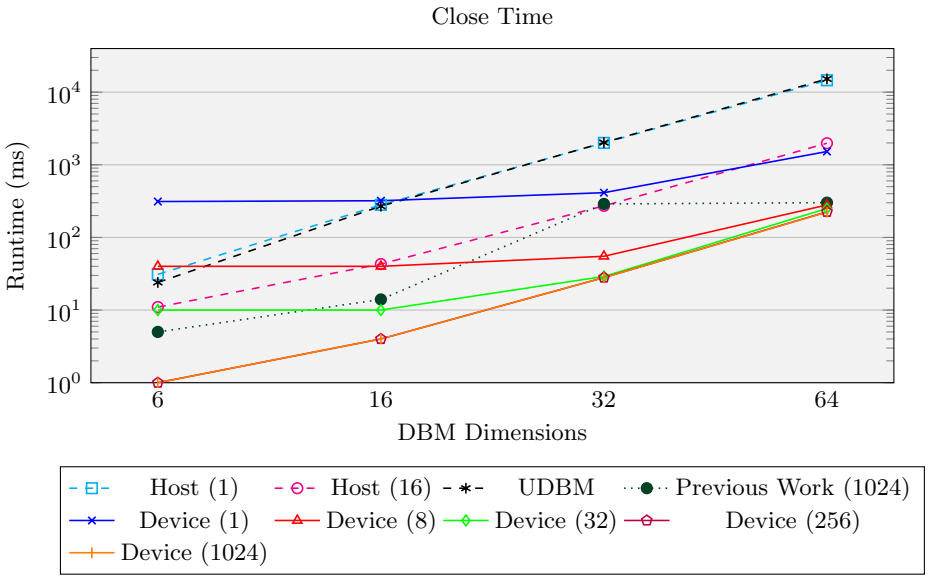


Fig. 17: Plot of computation time of different configurations computing the canonical form of 250000 DBMs for different dimensions. The y-axis in scaled logarithmic.

Intersection Figure 18 shows the results of the experiments on the intersection operation. While these experiments are still conducted with 250000 DBMs, the operation takes 2 DBMs as input, thus the operation is performed 125000 times. The results are somewhat similar to those observed from the closure experiments, as device with a single a batch size of 1 is slowest on 6 dimensions, but on 64 dimensions outperforms UDBM, single-threaded- and multi-threaded host configurations. The device configurations utilising a batch size of 1 to 32 for their computations initially seem almost static in execution time across DBM dimensions from 6 to 32. Given that this phenomenon does not persist in any of the other configurations of the same dimensions, we believe that this likely reflects the device not being sufficiently saturated on those experiments. The gained speedup between UDBM and GDBM ranges from 21 times on 6 dimensions to 52.9 times on 64 dimensions.

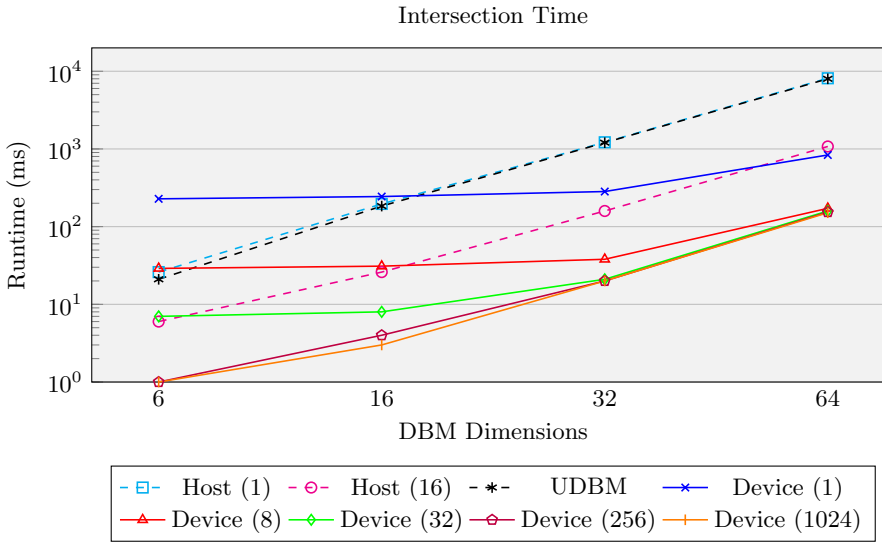


Fig. 18: Plot of execution time of different configurations computing the intersection of 125000 pairs of DBMs for different dimensions. The y-axis in scaled logarithmic.

Shortest-Path Reduction The results illustrated in Figure 19 somewhat surprisingly show great benefit in using concurrency for computing the shortest-path reduction of DBMs. We mentioned in Section 4.7 that our concurrent implementation utilises less optimal looping structures than the sequential computation for the purpose of more parallelism. It is unclear how much of the performance gain can be credited to this optimisation strategy. More likely is it that the performance gains stem from the underlying *Reduce* procedure that sequentially is

an $\mathcal{O}(n^3)$ whereas in our implementation is an $\mathcal{O}(n)$ procedure run many times in parallel. The subroutine essentially benefit from parallelism similarly to the *Close* operation, with additional advantage in not having to synchronise for each iteration of the $\mathcal{O}(n)$ loop. Device computations with a batch size of 1024 have a 63.2 times speedup to UDBM on 64 dimensions and a 9.5 times speedup to the multi-threaded host configuration.

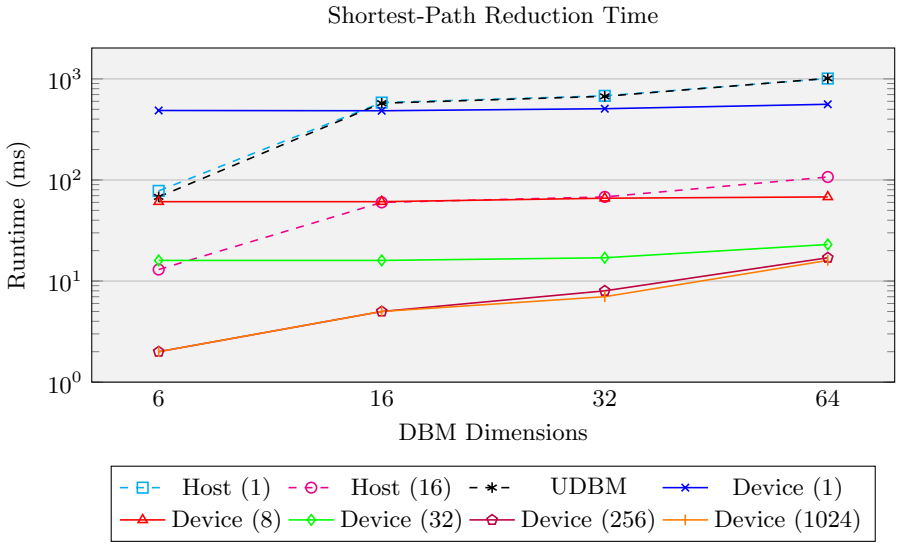


Fig. 19: Plot of execution time of different configurations computing the Shortest-Path Reduction of 250000 DBMs for different dimensions. The y-axis in scaled logarithmic.

ExtrapolationLU All extrapolation experiments are performed with lower and upper bounds of $(\leq, 5)$ and $(\leq, 15)$ respectively. The implementation of extrapolation utilises the specialised *CloseLU* for re-establishing the canonical form of the DBMs. This subroutine is similarly utilised by UDBM, thus making the experiments a better indication of its performance during reachability analysis or synthesis of timed games. The result of these experiments are seen in Figure 20, reinforcing the benefit of device computations. Using either a batch size of 1024 or 256 seems to make no difference, while both outperforms configurations of a batch size of 32 and 8 initially with a 37 times speedup, but have diminishing returns with only a 1.25 times speedup on 64 dimensions. This phenomenon occurs across multiple experiments and seems to stem from better device occupancy with more blocks and lower dimensions. We hypothesise that this is a result of the SIMT architecture – with lower dimensions there are fewer threads within a device block, thus making more blocks able to compute concurrently. As

such, the fewer clocks in the DBMs, the more we rely on the concurrent number of blocks in order to achieve high device occupancy.

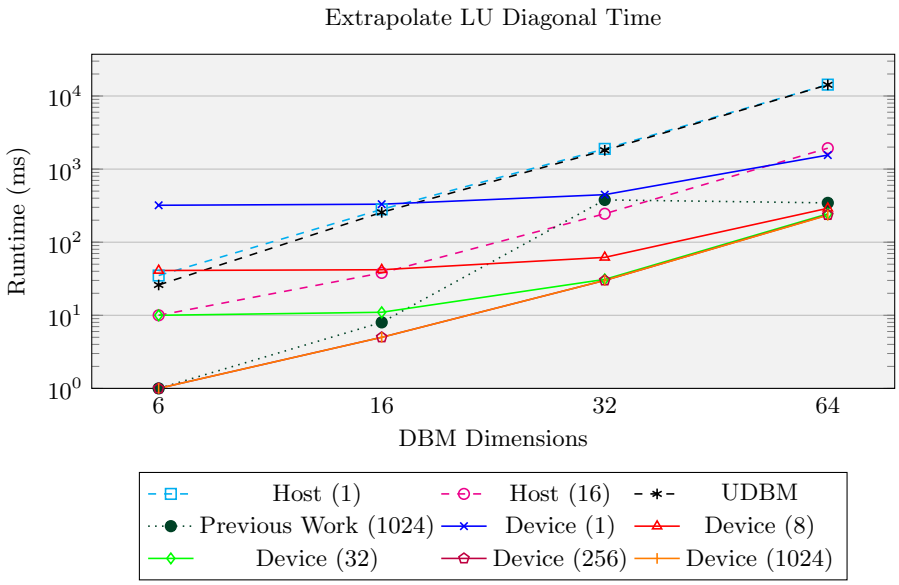


Fig. 20: Plot of execution time of different configurations extrapolating 250000 DBMs for different dimensions. The y-axis in scaled logarithmic.

DBM Subtraction DBM subtraction experiments are conducted with the intend of comparing the disjoint, non-disjoint and naive methods. We anticipated that the non-disjoint methods would perform the best in isolation, but may lead to worse performance when used for the synthesis of timed games. The results of these experiments can be seen in Figure 21, where it is clear that the multi-threaded Host (16) configuration performs the best, which may indicate that the device is poorly occupied during these experiments. While not initially clear only from Figure 21, we surprisingly found the disjoint method to outperform the non-disjoint one on all configurations. The difference is initially subtle on fewer dimensions and batches, having a speedup of 1.06 times between Device D (1) (361ms) and Device ND (1) (381ms) on 6 dimensions. The difference becomes slightly more pronounced, measuring a speedup of 1.5 times between Device D (1024) (3366ms) and Device ND (1024) (5024ms) on 64 dimensions.

Notably, subtract seems to be unique in that it does not scale much with an increase in batch size, rather sometimes having larger batch sizes being beat out by smaller batch sizes. We hypothesise that this is due to the varying degrees in which DBMs intersect and thus having varying degrees of splitting. The

larger batch sizes have to wait before all DBMs in a batch have finished computing before another batch can be launched, creating a bottleneck on the slowest computation in the batch.

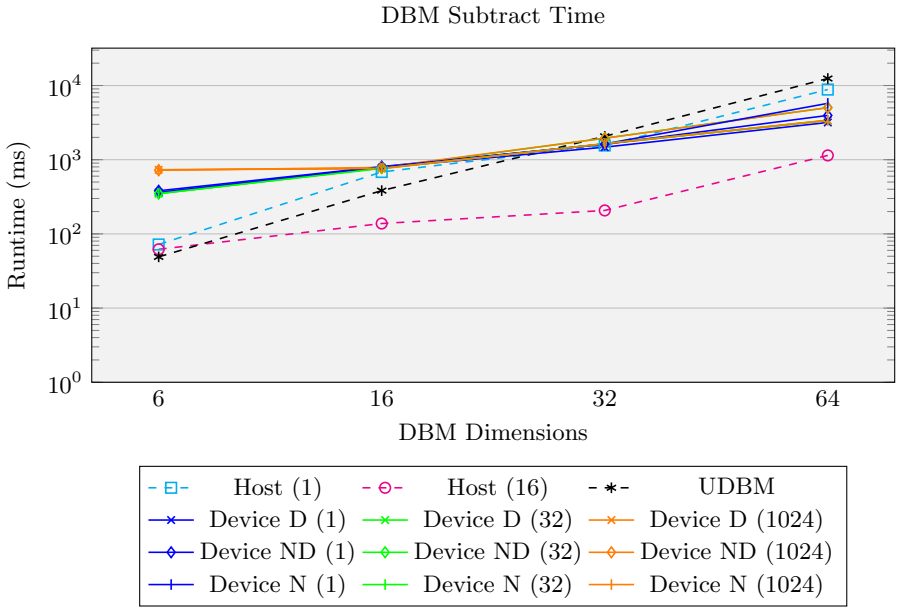


Fig. 21: Plot of execution time of different configurations computing subtraction of 125000 pairs of DBMs for different dimensions. D = Disjoint, ND = Not Disjoint, N = Naive. The y-axis in scaled logarithmic.

Approximate Relation The experiments regarding the approximate relation operation are conducted between 250000 DBMs and a federation of 100 DBMs to more closely mimic the subsumption check during reachability analysis and synthesis of timed games. GDBM implements computing the relation either across a block or a warp, both of which are tested in this experiment, the results of which can be seen in Figure 22. It seems to be entirely detrimental to only use a batch size of 1 for these computations, as those configurations were consistently outperformed by both single threaded Host and UDBM on all DBM dimensions. Even still, the warp version of the computations often outperforms its counterpart on the same batch size, which we credit to the ability to terminate early that is similarly found in sequential versions. This is not surprising, as the early termination was the incentive for the warp implementation, as was discussed in detail in Section 4.3. This early termination criteria may also result in almost static performance across different DBM dimensions for the warp configurations. On 64 dimensions, we measure a 13.7 times speedup between UDBM and the

warp computation with a batch size of 1024, but only a 1.4 times speedup in comparison to the multi-threaded host configuration. We additionally observe a speedup between all versions with and without warp computations on 64 dimensions, ranging from 3.7 times on a batch size of 1 to the largest difference of 22.3 times speedup on a batch size of 1024. On DBMs with only 6 dimensions, we measured virtually no difference between block and warp computations. This is unsurprising, as blocks are always launched with sizes of increments of 32 threads, meaning that both configurations are computed with exactly 32 device threads.

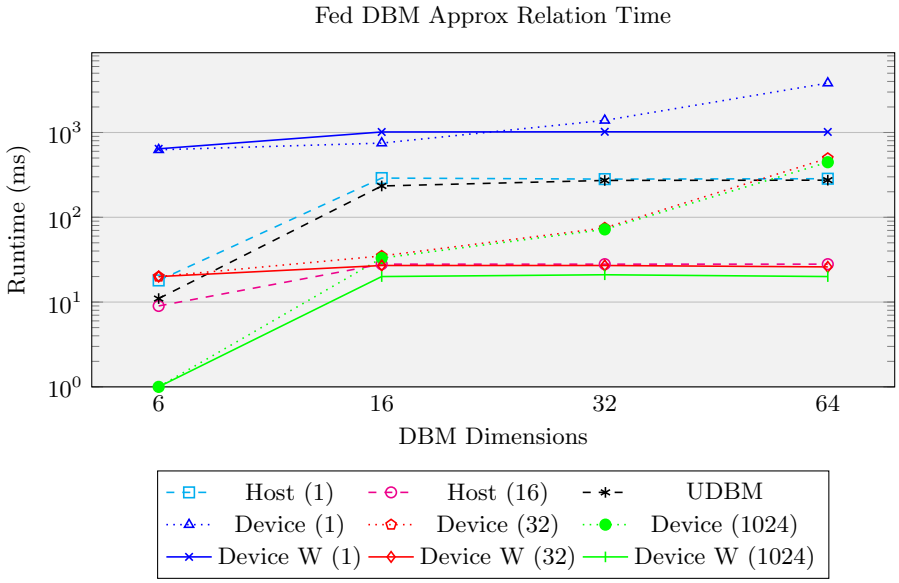


Fig. 22: Plot of execution time of different configurations computing the relation of 250000 DBMs against a federation of 100 DBMs for different dimensions. W signifies warp version of the operations. The y-axis in scaled logarithmic.

7.2 Fed-Fed Subtract

With the experiments regarding subtraction of federations, we primarily want to establish whether it in isolation is beneficial to disregard the disjointedness of the result set, in addition to the effect of partitioning DBMs across multiple blocks. The results of these experiments are visualised in Figure 23, where it is immediately noticeable how little deviation there is between disjoint and non-disjoint configurations. The disjoint configurations surprisingly outperform the non-disjoint ones on all but 6 dimensions, Non-Multi having a speedup of 1.17 when comparing Multi D (32) (1098ms) to Multi ND (32) (1284ms). We had

anticipated that ND configurations would be faster in isolation, but that it would bring some degree of penalty when used during reachability analysis. The Multi configuration also had a slight positive effect, measuring a speedup of 1.26 when comparing Device D (32) (1377ms) to Device multi D (32) (1091ms). On 6 dimensions, all device configurations are outperformed by Host (1), Host (16) and UDBM. The largest difference on this number of dimensions is measured between Device Multi N (32) (34ms) and Host (16) (1ms).

It intuitively looks rather strange that UDBM (292ms) outperforms Host (16) (756ms) on 16 dimensions. The explanation for these results is a sub-optimal host implementation in that it will have to re-compute the shortest-path reduction multiple times. Implementation wise, the shortest-path reduction only computes a mask for DBMs that indicates non-redundant constraints, instead of altering the DBMs directly. Unlike UDBM and the device implementation, the host implementation does not cache this mask. When computing $F - F'$ where $D_1, D_2, D_3 \in F$ and $D' \in F'$, the shortest path reduction of D' is computed for each of $D_1 - D'$, $D_2 - D'$ and $D_3 - D'$.

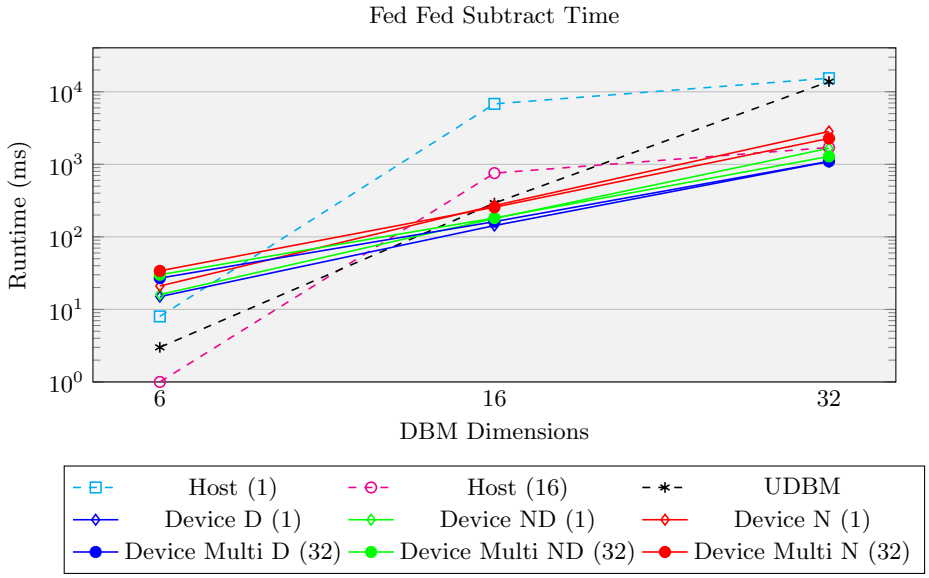


Fig. 23: Plot of execution time of different configurations computing Fed-Fed subtraction of 512 pairs of federations with a size of 3 DBMs on different dimensions. D = Disjoint, ND = Not Disjoint, N = Naive. The y-axis in scaled logarithmic.

Predt We experimentally compare our different methods for computing Pred_t. To reiterate, the different device methods involve either the entire computation on a single block, partitioning the DBMs in the "good" federation across many blocks (denoted "Multi"), and partitioning DBMs in both the "good" and "bad" federations across many blocks (denoted "Super"). The results of these experiments are seen in Figure 24. Each experiment is run with 1024 federations of size 3, generated using the GDBM method, resulting in many splits, due to all good and bad intersecting at origin. These experiments therefore test the worst case of pred_t, which causes the most splits. While not initially obvious from Figure 24, we generally see the super configuration outperforming both multi and the sequential methods on DBMs with more dimensions. For example, we measured a 1.95 times speedup between super and the sequential methods with a batch of 1 and 32 dimensions, while showing a 1.44 times slowdown on 6 dimensions and a batch size of 32 (57ms for sequential vs. 87ms for dimension). On 32 dimensions and a batch size of 32 we see a speedup of 1.05 between Device (27386 ms) and multi (25928 ms) and additional speedup of 1.09 between multi and super (23649 ms).

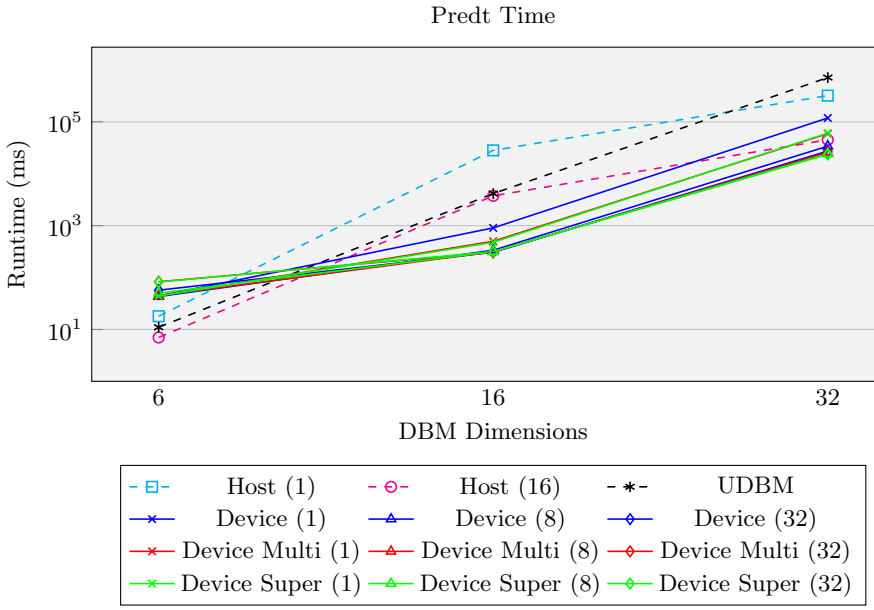


Fig. 24: Plot of execution time of different configurations computing pred_t of 512 pairs of federations with a size of 3 DBMs on different dimensions. The y-axis in scaled logarithmic.

These results are somewhat surprising, as we had anticipated a larger speedup from both the multi and super configurations. We may conclude that the methods utilising more concurrency are limited by the small federation size, and that these experiments poorly reflect the speedup that could be acquired from using these. However, this would be disingenuous, as our empirical experiments resulting in out of memory errors on the device with larger federation sizes. Still, we see a speedup of 11.77 times when comparing UDBM (713857 ms) to Device Super with a batch size of 1 (60619 ms) on 32 dimensions. The super configuration with a batch size of 32 only achieves a 1.91 times speedup when compared to the multi threaded host version on 32 dimensions (45203 ms).

7.3 Reachability

We have similarly already conducted and reported on symbolic reachability analysis experiments in [3] with our prototype tool SMAcc. We then noted how the co-processing implementation introduced many performance hampering elements such as locking mechanisms, and that Uppaal may benefit from a more optimised implementation from years of fine-tuning, that were not present in SMAcc. Having extended Uppaal with interweaved GDBM functionality similarly warrants re-conducting those experiments. The case study models and their type of property can be seen in Table 2. We compare the device enabled versions on a varying number of workers, and with both the strict and non-strict search orders that we mentioned in Section 5.4 to Uppaal. Additionally, we compare both versions with the host version of our algorithm, utilising both 1 and 15 workers. Our achieved results can be seen in Table 3.

Model	Property	Exhaustive	#Clocks	#Components
FischerMutex-N9	`A[]`	Yes	10	9
FischerMutex-N12	`A[]`	Yes	13	12
Milner-N100d4	`E<>`	No	202	101
TrainMutex-N9	`A[]`	Yes	10	10
Vikings-N16	`E<>`	No	18	17

Table 2: The different models, their type of property, whether the property is exhaustive, and the amount of clocks and components in the models.

Notably, many of the device configuration are now within a single order of magnitude of the performance of Uppaal, which is a large improvement on the results obtained from SMAcc [3]. For example, on the FischerMutex-N9 model we previously reported a performance of 7 minutes, meaning we now have a 178.72 times speedup. On Milner we now see a speedup of 4.91 times with the old runtime of 44 seconds. The Milner experiments might not seem as impressive, but a point worth considering is the difference between search orders. Our previous work, used a loosely defined BFS with work stealing queues, which

had the effect of only approximating BFS (more loosely than current non-strict version) rather than strictly following BFS, and would therefore terminate faster once the property was satisfied.

Model	Workers	Device	Device Strict	Host	Host Strict	UPPAAL
FischerMutex-N9	1	8.36	8.34	31.69	31.99	0.83
	64/15	2.35	2.38	7.63	7.67	-
	256	2.38	2.40	-	-	-
	1024	2.49	2.40	-	-	-
FischerMutex-N12	1	302.50	303.80	TO	TO	51.66
	64/15	87.23	88.09	TO	TO	-
	256	86.34	86.59	-	-	-
	1024	89.77	89.53	-	-	-
Milner-N100d4	1	25.09	24.89	14.23	14.27	8.00
	64/15	9.50	9.22	3.02	3.61	-
	256	10.04	8.96	-	-	-
	1024	10.24	9.47	-	-	-
TrainMutex-N9	1	369.35	368.03	24.5	22.46	20.91
	64/15	87.77	87.05	38.41	37.54	-
	256	87.87	87.14	-	-	-
	1024	91.60	92.03	-	-	-
Vikings-N16	1	358.37	357.57	22.38	22.21	25.82
	64/15	89.33	89.88	34.45	32.70	-
	256	89.10	88.76	-	-	-
	1024	93.04	92.92	-	-	-

Table 3: Run times for reachability experiments. Host and Device refers to the asynchronous Host and Host-Device implementations, respectively, and Strict refers to running it with the strict exploration order. The columns with 64/15 workers indicate that the Device and Host were run with 64 and 15 workers, respectively. TO indicates termination due to timeout after 10 minutes. Runtimes are measured in seconds.

Comparing the run-times using strict to non-strict does not show any significant changes. Increasing the amount of workers from 1 to 64, enabling more concurrency, gives a performance boost on all models, however increasing this count further does not seem to produce significantly better outcomes. This might be due to there being too few states in the waiting list on the host, and the GPU can therefore not utilise the extra workers since there is no work for them to compute. Additionally, the overhead associated with launching kernels likely introduces a bottleneck in the form of under utilising the device. Profiling work shows 35% of the runtime on FischerMutex-N9, and 22% on TrainMutex-N9, to be consumed by kernel invocation. The work dependency system is much more lightweight in this iteration of the tool, taking up only 11% and 5% of the

runtime on FischerMutex-N9 and TrainMutex-N9 respectively. This is a huge improvement to our previous work, where locking took up 30% of the runtime.

Our host exclusive implementation outperforms the device enabled implementation on all models except for the two FischerMutex models where it either is a bit slower or not able to perform the reachability analysis within the 10 minute timeout limit. On compute heavy models, such as the Milner model, increasing the number of workers greatly decreases the runtime, as the 15 workers version outperforms even Uppaal. This is likely due to the model requiring more computations per task, due to the number of clocks in the system. Through profiling work with gprof we found the communication overhead through boost's lockfree queue to be the major bottleneck, having around 48% of the runtime being contained within said queue. Additionally, we had anticipated increasing the number of workers on the host implementation to always be beneficial, but surprisingly this was not the case on the Vikings and TrainMutex models. We hypothesise that this is due to the number of unique DBMs encountered during the reachability analysis to be exceedingly low for these models, limiting the tasks being performed by each worker, thus further pronouncing the bottle-necking effect of the communication overhead.

Hash Table Experiments We have documented statistics of both the DBM and federation hash tables that were described in Sections 4.12 and 4.13 at the *end* of the reachability analysis, to better gauge their significance. Both the DBM table and federation table were initialised with 2^{26} entries, resulting in them occupying 1GB and 0.5GB of device memory, respectively.

Model	#DBMs	#Unique DBMs	Memory Saved	#Collisions	Max Offset
FischerMutex-N12	2681780	24565	2108mb	171	1
FischerMutex-N9	81035	2296	39mb	0	0
Milner-N100d4	24391	12330	1955mb	1	1
TrainMutex-N9	6541957	10	3244mb	0	0
Vikings-N16	6160214	17	8882mb	0	0

Table 4: DBM hash table statistics collected at the *end* of the reachability analysis of various models. All results are using 1024 workers and strict ordering.

Table 4 contains these statistics for the DBM hash table, documenting the total number of stored DBMs, how many of these are unique, how much memory the table has saved as opposed to storing every DBM, how many hash collisions occurred during the reachability analysis, and the maximum offset of a DBM compared to where the hash function would index it at. It is immediately noticeable that collisions and thus offsets are exceedingly rare. Unsurprisingly, collisions are more likely to occur the more unique DBMs are stored, which is

reflected in the FischerMutex-N12 and Milner models. Even having 171 hash collision, the maximum offset of a stored DBM from the run of FischerMutex-N12 is still only 1. The benefit of our DBM hash table is indisputable from a glance at the column of memory saved. Remarkably, only 17 and 10 unique DBMs was encountered in the Vikings and TrainMutex models, respectively, saving 8GB of memory in the most extreme case. As with the utilised system only having 8GB of device memory, we would have been unable to conduct experiments on the Vikings-N16 model without the use of the DBM hash table. An additional, albeit undocumented, benefit of the DBM hash table is its effect on subsumption checking. As DBMs are shared, we can establish DBM equality simply through pointer equality.

Model	#Feds	Avg. Offset	Max Offset	Avg. Span	Max Span	#Tombstones
FischerMutex-N12	2681780	3.13	27	4.52	38	7391761
FischerMutex-N9	81035	1.89	11	2.90	13	152746
Milner-N100d4	24385	0.02	2	1.02	3	582
TrainMutex-N9	6541957	0.06	7	1.15	10	0
Vikings-N16	6160214	0.05	7	1.14	10	0

Table 5: Federation hash table statistics collected at the *end* of the reachability analysis of various models. All results are using 1024 workers and strict ordering. All models had a max fed size of ≤ 2.

Statistics from the federation table are seen in Table 5. Most importantly, we see that the maximum offset found throughout all reachability analysis experiments were 27 and the maximum span, which denotes the distance between two empty slots, were at most 38. Using the double linear probing scheme that utilises warps, this means that we only in very rare cases exclusively on the FischerMutex-N12 model could potentially require additional probing for insertion and subsumption checks. This is somewhat surprising, as we had anticipated harsher penalties as the table would be exceedingly more occupied. Even when so many entries simply store tombstone values, we see only minimal penalty. We found the average *and* maximum sizes for all models to be exactly 1, with the exception of the Milner model with a maximum federation size of 2. This is only indicative of gathering data at the end of the reachability analysis, as the number of tombstone value reflects federations having been larger throughout the analysis. What we can conclude from these statistics is simple that the federation table works as intended and is not the bottleneck we anticipated it to be.

7.4 Timed Games

We conduct experiments on the synthesis of timed games with a similar methodology as with reachability analysis, while not being able to compare it against any previous work. An overview of the case study models and their properties to synthesise a controller for is listed in Table 6 with corresponding experimental result in Table 7. Notably the models contain fewer clocks and components than in the reachability experiments, as larger models scale worse since Uppaal Tiga does not use local or LU extrapolation, leading to larger models timing out after 10 minutes on all configurations.

Model	Property	#Clocks	#Components
Juggler-N2	`control: A[]`	5	4
Juggler-N3	`control: A[]`	6	5
Prodcell-3-cont	`control: A[]`	5	4
Prodcell-3-uncont	`control: A[]`	5	4
TrainGame-N4	`control: A[]`	6	4

Table 6: The different models, their type of property, and the amount of clocks and components in the models.

We measure performance of the asynchronous device implementation that is often on par with, but never faster than, Uppaal Tiga. We have already established UDBM to be faster than GDBM on the subtract and pred_t operation on few dimensions, which likely contributes majorly to this. Still, it is somewhat surprising that the device enabled implementations are not *far* slower than Uppaal Tiga, as we had anticipated them to be.

The effect of using either a strict or non-strict search order for timed games is much the same as the one established for reachability analysis, while still being somewhat inconsistent. For example, we see that the strict ordering outperforms non-strict on the Juggler-N3 model regardless of the number of workers, while this is not the case on the Juggler-N2 model. The results seems to generally favour the non-strict ordering, disregarding the Juggler-N3 model and TrainGame with few workers.

The performance effect – or lack thereof – with the growth of number of workers is somewhat surprising, as we had anticipated larger penalty on such small models. We mentioned in Section 6.2 how detrimental duplicate work may be scheduled, e.g. if a location `l1` and its successor `l2` are both queued for back-propagation it would be unnecessary to compute it for `l1` before `l2`. Eliminating such duplication is at the moment only based on work that has already completed, i.e. jobs currently being computed are not taking into account. We had anticipated a detrimental effect on the runtime from a combination of work duplication growing with the number of workers as well as additional overhead, however the results of our experiments does not suggest this to be the case.

Model	Workers	Device	Device Strict	Uppaal Tiga
Juggler-N2	1	0.56	0.47	0.16
	64	0.33	0.31	-
	256	0.31	0.32	-
	1024	0.33	0.43	-
Juggler-N3	1	31.06	30.23	2.67
	64	26.00	13.27	-
	256	71.43	23.33	-
	1024	41.10	13.84	-
Prodcell-3-cont	1	5.34	5.20	0.15
	64	2.57	2.37	-
	256	2.18	2.25	-
	1024	2.41	2.44	-
Prodcell-3-uncont	1	0.94	0.92	0.16
	64	0.66	0.66	-
	256	0.67	0.69	-
	1024	0.76	0.72	-
TrainGame-N4	1	3.25	3.25	0.64
	64	1.80	1.97	-
	256	1.92	1.83	-
	1024	1.85	1.89	-

Table 7: Run times for Tiga experiments. Device refers to the asynchronous Host-Device implementation, and Strict refers to running it with the strict exploration order. Run-times are measured in seconds.

8 Conclusion

In this work, we show the applicability and advantages of utilising Graphical Processing Units for operations on Difference Bound Matrices in the context of both symbolic reachability analysis and controller synthesis of timed games. We elaborated and extended our work from the prototype tool SMAcc [3] and presented the **GDBM** library with novel techniques and methods for dealing with DBM operations massively in parallel, specifically for enabling the synthesis of Timed Game Automata. We have additionally implemented extensions of Uppaal and Uppaal Tiga for interweaving asynchronous calls to GDBM, such that the continuous and discrete state space explorations can be computed concurrently between Uppaal and GDBM.

In the final part of our work, we empirically investigated the performance of GDBM, both in isolation, in the context of symbolic reachability analysis, and in controller synthesis of timed games. While we have achieved great results on many operations in isolation, our implementation of subtraction and consequently pred_t are somewhat lacklustre, especially on DBMs with fewer dimensions. This is especially unfortunate as these are fundamental to the contribution of this thesis. Of the other DBM and federation operations, we still

outperform even the multi-threaded host computation, and have now improved on our previous work from SMAcc [3] with up to 10.32 times.

In comparison to Uppaal and Uppaal Tiga, we are within a single order of magnitude slower. In the case of synthesis of timed games, this is very encouraging, as we had anticipated larger penalties from the underperforming subtract and pred_t operations. However compared to SMAcc we see up to a 178.72 times speedup.

In addition, we documented statistics of our implemented DBM and federation hash tables, showing large memory savings and benefits. As our isolated DBM experiments show subtraction and pred_t to perform poorly on DBMs with fewer dimensions, we know this to have an effect on the timed games experiments. As opposed to reachability experiments, we never outperform Uppaal Tiga but are still surprisingly close to its performance.

In general, we have experimentally showed promising potential for both device enabled reachability analysis and synthesis of timed games, however more work is needed to make this a viable option. Controller synthesis is primarily due for future work in that it is severely limited by the sub-optimal subtract and pred_t operations.

8.1 Future Work

Our experimental evaluation of GDBM shows great potential in isolation, while the performance of the asynchronous extensions of Uppaal and Uppaal Tiga are somewhat lacklustre and in no way realises this potential. Future work entails devising methods for making the use of GPUs viable for formal verification. The current implementations seem to be entirely limited by the GPU not being saturated nearly enough, i.e. an imbalance of communication overhead to actual GPU computations. To counteract the communication overhead, an obvious point worth exploring in future work is to launch kernels for *multiple* discrete states in the waiting list at once, rather than just one at a time. Additionally, the possibility of sending successors of successors is also worth considering. More involved would be developing methods for doing the formal verification and controller synthesis exclusively on the device akin to the work of GPUExplore [37], thus entirely eliminating the present communication overhead.

The implementation of the federation table is also worth revising. While the statistics gathered as part of our experimental evaluation showed no signs of performance penalty, we are still concerned about the scalability of the federation table, as empirical experiments showed clear performance drop-offs as the number of entries grew.

While we have documented promising experimental results that hint at the potential benefit of device computations for formal verification and controller synthesis, it is still clear that more concurrency, better implementations and different techniques are needed for this to be a competitive alternative to Uppaal and Uppaal Tiga. Given that the DBM subtract operation is fundamental to the contribution of this thesis, it is unfortunate that so little performance gain and deviation was obtained across the various described methods. Future works

should involve investigating alternative methods for computing this operation, as our experiments suggest that to be a major limitation for the pred_t operation and thus the entire synthesis of timed games. For subtraction of federations, it is likely worth exploring the use of dynamic parallelism for partitioning of the subtrahend, as well as purely using lazy removal during concurrent union across blocks. However, better performance of DBM subtraction is of more pressing concern, as it of course is involved in subtraction of federations and similar lacklustre results were obtained from those experiments. While the subtract operation is likely the major limiting factor of the pred_t operation, it would be obvious to revisit the computation of intersection of federations, as this is trivially parallelisable and were only omitted from the contribution of this thesis due to time limitations.

We were for similar reasons unable to further explore potential optimisations and variations of merging convex unions. The performance and benefit of merging has unfortunately been left undocumented, but empirical experiments showed its use following the pred_t operation to be hugely advantageous, even though the sub-routine has seen little optimisation in our implementation. It is likely possible to use concurrency as part of many of the subroutines of merging pairs of DBMs. Likewise, it may be possible to use concurrency to more efficiently attempt merging multiple DBMs as mentioned in [17,18], rather than limiting to only pairs.

The extension of Uppaal Tiga has unfortunately seen few optimisations, as a consequence of this thesis' time limitation. In addition to better performance of the underlying DBM and federation operations, it is possible to add more parallelism to the synthesis of timed games, especially during back-propagation, and through grouping of multiple discrete states together. Our extension assumes only discretely defined winning states (i.e. a location named goal). Extending the work to supporting liveness properties, involving fixed-point computations on the dependency graphs using GPUs, is also left for future work.

Acknowledgements. The authors want to thank Marius Mikučionis for his consultation regarding implementation into Uppaal and Uppaal Tiga, as well as Kim Guldstrand Larsen and Thomas Møller Grosen for their guidance.

9 Bibliographical Note

The content presented in this thesis is largely a continuation of our 9^{th} semester project made in collaboration with Marcus D. Jensen, Simas Juozapaitis and Andreas Windfeld. For the purpose of self-containment, this thesis include sections that are directly from this jointly made work. The **correctness** sub-section of 4.2 and the **Relation** sub-section of 4.3 are partly from this joint work but has been reworded. Section 2.1 is based on a section from previous work, but updated to fit the current domain. Section 2.2 is partly from the previous work, novel to this thesis are the sub-sections of **Subtract**, **Shortest-Path Reduction** and **Merging**. Section 3 is similarly partly from previous work, with only the sub-section 3.3 being entirely novel to this thesis, while the remainder of the section has been based on our 9^{th} semester project. DBM operations found in Appendix A were also part of our joint work, with new extensions from Algorithm 30 and onward. The content of this thesis not mentioned here is unique to this work.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, Apr. 1994.
2. O. S. Bak, M. W. B. Christiansen, O. V. Eriksen, M. D. Jensen, S. Juozapaitis, and A. Windfeld. Smacc: A gpu accelerated statistical model checker for stochastic systems. 2022.
3. O. S. Bak, M. W. B. Christiansen, O. V. Eriksen, M. D. Jensen, S. Juozapaitis, and A. Windfeld. Multicore cpu and gpu acceleration of symbolic model checking. 2024.
4. J. Barnat, L. Brim, and M. Češka. Divine-cuda-a tool for gpu accelerated ltl model checking. *arXiv preprint arXiv:0912.2555*, 2009.
5. J. Barnat, L. Brim, and P. Ročkai. Scalable multi-core ltl model-checking. In *Model Checking Software: 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007. Proceedings 14*, pages 187–203. Springer, 2007.
6. E. Bartocci, R. DeFrancisco, and S. A. Smolka. Towards a gpgpu-parallel spin model checker. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, pages 87–96, 2014.
7. G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen. Static guard analysis in timed automata verification. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 254–270, Berlin, Heidelberg, 2003. Springer-Verlag.
8. G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *STTT*, 8(3):204–215, 2006.
9. G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *Computer Aided Verification: 19th International Conference*, volume 4590 of *LNCS*, pages 121–125. Springer, 2007.
10. G. Behrmann, K. G. Larsen, O. Moller, A. David, P. Pettersson, and W. Yi. Uppaal-present and future. In *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No. 01CH37228)*, volume 3, pages 2881–2886. IEEE, 2001.

11. J. Bengtsson. *Clocks, dbms and states in timed systems.* Acta Universitatis Upsaliensis, 2002.
12. F. Boenneland, P. Jensen, K. Larsen, M. Muniz, and J. Srba. Start pruning when time gets urgent: Partial order reduction for timed systems. In *CAV'18*, volume 10981 of *LNCS*, pages 527–546. Springer-Verlag, 2018.
13. F. M. Bønneland, P. G. Jensen, K. G. Larsen, M. Muñiz, and J. Srba. Partial order reduction for reachability games. In *30th International Conference on Concurrency Theory (CONCUR 2019)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2019.
14. F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Concurrency Theory: 16th International Conference*, volume 3653 of *LNCS*, pages 66–80. Springer, 2005.
15. M. B. Dahlsen-Jensen, B. Fievet, L. Petrucci, and J. van de Pol. On-the-fly algorithm for reachability in parametric timed games (extended version), 2024.
16. A. E. Dalsgaard, A. Laarman, K. G. Larsen, M. C. Olesen, and J. Van De Pol. Multi-core reachability for timed automata. In *Formal Modeling and Analysis of Timed Systems: 10th International Conference, FORMATS 2012, London, UK, September 18-20, 2012. Proceedings 10*, pages 91–106. Springer, 2012.
17. A. David. Merging dbms efficiently–extended abstract.
18. A. David. Merging dbms efficiently. In *17th Nordic Workshop on Programming Theory*, pages 54–56. DIKU University of Copenhagen, 2005.
19. A. David, G. Behrmann, K. G. Larsen, and W. Yi. A tool architecture for the next generation of uppaal. In *Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers*, pages 352–366. Springer, 2003.
20. A. David, J. Håkansson, K. G. Larsen, and P. Pettersson. Model checking timed automata with priorities using dbm subtraction. In *Formal Modeling and Analysis of Timed Systems: 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006. Proceedings 4*, pages 128–142. Springer, 2006.
21. L. De Alfaro, T. A. Henzinger, and R. Majumdar. Symbolic algorithms for infinite-state games. In *CONCUR 2001—Concurrency Theory: 12th International Conference Aalborg, Denmark, August 20–25, 2001 Proceedings 12*, pages 536–550. Springer, 2001.
22. D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems: International Workshop*, volume 407 of *LNCS*, pages 197–212. Springer, 1990.
23. R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345–345, 1962.
24. I. Gelado and M. Garland. Throughput-oriented gpu memory allocation. In *Proceedings of the 24th symposium on principles and practice of parallel programming*, pages 27–37, 2019.
25. M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager. Adding symmetry reduction to uppaal. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems*, pages 46–59, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
26. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.
27. J. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. In S. Sahni, editor, *Proceedings of the International Conference on Parallel Processing*,

pages 713–716. Pennsylvania State Univ Press, Dec. 1987. Proc Int Conf Parallel Process 1987 ; Conference date: 17-08-1987 Through 21-08-1987.
28. D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt. Warpcore: A library for fast hash tables on gpus. In *2020 IEEE 27th international conference on high performance computing, data, and analytics (HiPC)*, pages 11–20. IEEE, 2020.
29. V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, 1991.
30. K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proceedings Real-Time Systems Symposium*, pages 14–24. IEEE, 1997.
31. K. G. Larsen, M. Mikučionis, M. Muñiz, and J. Srba. Urgent partial order reduction for extended timed automata. In D. V. Hung and O. Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 179–195, Cham, 2020. Springer International Publishing.
32. X. Liu and S. A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *Automata, Languages and Programming: 25th International Colloquium*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998.
33. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *12th Annual Symposium on Theoretical Aspects of Computer Science*, volume 900 of *LNCS*, pages 229–242. Springer, 1995.
34. NVIDIA. Cuda c++ programming guide, 2023. Last accessed 30 November 2023. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.
35. A. Pnueli, E. Asarin, O. Maler, and J. Sifakis. Controller synthesis for timed automata. In *System Structure and Control*. Citeseer, Elsevier, 1998.
36. M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, 2012.
37. A. Wijs and M. Osama. Gpuexplore 3.0: Gpu accelerated state space exploration for concurrent systems with data. In *International Symposium on Model Checking Software*, pages 188–197. Springer, 2023.

A DBM operations

Algorithm 20 Close

```
1: procedure Close(D)                                   ▷ DBM D
2:     for k := 0 to n do                               ▷ Number of clocks n
3:         for i := 0 to n do
4:             for j := 0 to n do
5:                 D_{i,j} = min(D_{i,j}, D_{i,k} + D_{k,j})
```

Algorithm 21 Is_empty

```
1: procedure is_empty(D)                                ▷ DBM D
2:     for i := 0 to n do                               ▷ number of clocks n
3:         for j := i to n do
4:             w := D_{i,j} + D_{j,i}
5:             if w < (0, ≤) then
6:                 return true
7:     return false
```

Algorithm 22 Inclusion

```
1: procedure relation(D, D')                            ▷ DBMs D, D'
2:     sub := true, super := true
3:     for i := 0 to n do
4:         for j := 0 to n do
5:             sub = sub ∧ (D_{i,j} ≤ D'_{i,j})
6:             super = super ∧ (D_{i,j} ≥ D'_{i,j})
7:     return ⟨sub, super⟩
```

Algorithm 23 Future

```
1: procedure future(D)                                  ▷ DBM D
2:     for i := 1 to n do
3:         D_{i,0} = ∞
```

Algorithm 24 Past

```
1: procedure past(D)                                          ▷ DBM D
2:     for i := 1 to n do
3:         D_{0,i} = (0, ≤)
4:         for j := 1 to n do
5:             if D_{j,i} < D_{0,i} then
6:                 D_{0,i} = D_{j,i}
```

Algorithm 25 Restrict

```
1: procedure restrict(D, (x − y ~ m))                         ▷ DBM D, bound
2:     if D_{y,x} + (m, ~) < 0 then
3:         D_{0,0} = (−1, ≤)
4:     else if (m, ~) < D_{x,y} then
5:         D_{x,y} = (m, ~)
6:         for i := 0 to n do
7:             for j := 0 to n do
8:                 if D_{i,x} + D_{x,j} < D_{i,j} then
9:                     D_{i,j} = D_{i,x} + D_{x,j}
10:                if D_{i,y} + D_{y,j} < D_{i,j} then
11:                    D_{i,j} = D_{i,y} + D_{y,j}
```

Algorithm 26 Assign

```
1: procedure assign(D, x = m)                                 ▷ DBM D, clock x, new value m
2:     D_{x,0} = (m, ≤)
3:     D_{0,x} = (−m, ≤)
4:     for i := 0 to n do
5:         D_{x,i} = (m, ≤) + D_{0,i}
6:         D_{i,x} = D_{i,0} + (−m, ≤)
```

Algorithm 27 Copy

```
1: procedure copy(D, x = y)                                   ▷ DBM D, clocks x and y
2:     for i := 0 to n do
3:         if i ≠ x then
4:             D_{x,i} = D_{y,i}
5:             D_{i,x} = D_{i,y}
6:     D_{x,y} = (0, ≤)
7:     D_{y,x} = (0, ≤)
```

Algorithm 28 Shift

```
1: procedure shift(D, x = x + m)                    ▷ DBM D, clock x, value m
2:     for i := 0 to n do
3:         if i ≠ x then
4:             D_{x,i} = D_{x,i} + (m, ≤)
5:             D_{i,x} = D_{i,x} + (−m, ≤)
6:     D_{x,0} = max(D_{x,0}, (0, ≤))
7:     D_{0,x} = min(D_{0,x}, (0, ≤))
```

Algorithm 29 LU_Extra$^+$

```
1: procedure LU_Extra+(D, [L_0, ..., L_n], [U_0, ..., U_n])   ▷ DBM, lower, upper bounds
2:     for i := 0 to n do
3:         for j := 0 to n do
4:             if i == j then continue
5:             if D_{i,j} > L_i then
6:                 D_{i,j} = ∞
7:             else if −D_{0,i} > L_i then
8:                 D_{i,j} = ∞
9:             else if −D_{0,j} > U_j and i ≠ 0 then
10:                D_{i,j} = ∞
11:            else if −D_{i,j} > U_j and i == 0 then
12:                D_{i,j} = (−U_j, <)
13:            if i = 0 and D_{i,j} > (0, ≤) then
14:                D_{i,j} = (0, ≤)
15:            if j = 0 and D_{i,j} < (0, ≤) then
16:                D_{i,j} = (0, ≤)
17:    Close(D)
```

Algorithm 30 Disjoint Subtract

```
1: procedure disjoint_subtract(D, E)                ▷ DBMs D, E
2:     Compute E_m
3:     S = false, R = D
4:     for e_{ij} ∈ E_m, i ≠ j do
5:         S = S ∨ (R ∧ ¬e_{ij})
6:         R = R ∧ e_{ij}
7:     return S
```

Algorithm 31 Reduction of Zero-Cycle Free Graph G with n nodes

```
1: procedure Reduce_zero_cycle_free(G, n)          ▷ Graph G, number of nodes n
2:     for i := 1 to n do
3:         for j := 1 to n do
4:             for k := 1 to n do
5:                 if G_ik + G_kj ≤ G_ij then
6:                     Mark edge i → j as redundant
7:     Remove all edges marked as redundant.
```

Algorithm 32 Reduction of Negative-Cycle Free Graph G with n nodes

```
1: procedure reduce_negative_cycle_free(G, n)      ▷ Graph G, number of nodes n
2:     for i := 1 to n do
3:         if Node_i is not in partition then
4:             Eq_i = ∅
5:             for j := 1 to n do
6:                 if G_ij + G_ji = 0 then
7:                     Eq_i := Eq_i ∪ {Node_i}
8:     Let G' be a graph without nodes.
9:     for each Eq_i do
10:        Pick one representative Node_i ∈ Eq_i
11:        AddNode_i to G'
12:        Connect Node_i to all nodes in G' using weights from G.
13:    Reduce G'
14:    for each Eq_i do
15:        Add one zero cycle containing all nodes in Eq_i to G'
```

Algorithm 33 2-way merging of DBMs in Federation F

```
1: procedure 2way_merge(F)                         ▷ Federation F
2:     for each (D, D') ∈ F) do
3:         if H(D, D') then                        ▷ Heuristic function
4:             if ((D ⊔ D') − D) − D' = ∅ then
5:                 F := F ∪ {D ⊔ D'}
```