Time Optimal Reachability Analysis using Swarm Verification

Zhengkui Zhang
Department of Computer Science
Aalborg University
Denmark
zhzhang@cs.aau.dk

Brian Nielsen
Department of Computer Science
Aalborg University
Denmark
bnielsen@cs.aau.dk

Kim G. Larsen
Department of Computer Science
Aalborg University
Denmark
kgl@cs.aau.dk

ABSTRACT

Time optimal reachability analysis employs model-checking to compute goal states that can be reached from an initial state with a minimal accumulated time duration. The model-checker may produce a corresponding diagnostic trace which can be interpreted as a feasible schedule for many scheduling and planning problems, response time optimization etc. We propose swarm verification to accelerate time optimal reachability using the real-time model-checker UPPAAL. In swarm verification, a large number of model checker instances execute in parallel on a computer cluster using different, typically randomized search strategies. We develop four swarm algorithms and evaluate them with four models in terms scalability, and time- and memory consumption. Three of these cooperate by exchanging costs of intermediate solutions to prune the search using a branch-and-bound approach. Our results show that swarm algorithms work much faster than sequential algorithms, and especially two using combinations of random-depth-first and breadth-first show very promising performance.

CCS Concepts

•**Theory of computation** → **Verification by model checking;** •**Computing methodologies** → *Distributed algorithms;*

Keywords

Timed automata; Time optimal reachability; UPPAAL; Swarm verification, Distributed computing

1. INTRODUCTION

Time-optimal reachability (TOR) analysis is a novel approach for solving scheduling and planning problems using model checking techniques. Given the model of a system, a model checker can automatically and exhaustively check whether this model satisfies a given specification. It may produce a diagnostic trace (a.k.a. counterexample, or witness) if the model fails to satisfy the specification. Around year 2000, researchers noticed that scheduling problems of real-time systems can be reformulated to time optimal reachability problem on timed automata [17, 1]. A diagnostic trace to a goal state offered by real-time model checkers such as UPPAAL [5] and KRONOS [11] can be interpreted as a feasible schedule because the trace carries actions of the model as well as timing information of these actions to the goal. Compared with the classical numerical method for scheduling and planning such as linear programming, dynamic programming etc, modeling describes the real-time behavior, constrains and interactions of components in a natural way. The other advantage is flexibility, because model checkers efficiently implements well known search algorithms, such as breadth-first-search (BFS), depth-first-search (DFS), random-depth-first-search (RDFS) etc, transparently of the input models. Users can therefore update topological and timing constrains to the model easily without being forced to change the underlying algorithms [12].

Related Work. Branch and Bound (B&B) is an algorithm paradigm widely applied in optimization and planning algorithms on graphs. The purpose of using B&B is to avoid enumeration the entire solution space. By a bounding function and the current best solution to the goal, B&B allows the algorithm to effectively prune parts of the solution space that guarantee do not to lead to an optimal solution [13]. Behrmann *et al.* presented a branch-and-bound minimal-cost reachability algorithm on priced timed automata (PTA) in [7, 6]. Another way to restrict and guide the state-space exploration is adding constrains on transitions [15].

As the number of components in the model increases, the size of the state-space may grow exponentially – the state-space explosion problem. One approach to push that barrier is to run a distributed model checker on a computer cluster such as distributed UPPAAL [8, 4], DIVINE [3], LTSMIN [10], etc. These tools work by partitioning the state-space and distributing the parts among distributed CPU and memory resources using message passing. Given sufficient memory and time, this paradigm can verify very large models. A different paradigm called *swarm verification* was proposed by Holzmann *et al.* in [14]. A number of SPIN instances configured with diverse search strategies and bitstate hashing run in parallel. Even though bitstate hashing may lead to false positive results, swarm can give high quality result fast even under restricted time and memory. Another approach named agent-based search was proposed by Rasmussen *et*

SAC 2016, April 04-08, 2016, Pisa, Italy

DOI: http://dx.doi.org/10.1145/2851613.2851828

al. in [18]. Various agents with diverse search patterns can put (and get) tasks to (and from) a pool, where tasks are sub-paths that lead to promising parts of the state-space.

Contribution. We design and implement swarm verification algorithms for the UPPAAL model-checker to solve the time optimal reachability problem. The large number of random searches can explore different parts of the state-space, thus finding different traces to the goal state in parallel and avoiding local optimality. Therefore this approach enable users to get optimal (or near optimal) results fast without exploring the full state-space. When time or memory are limited, this approach can still give high quality near optimal results. Because TOR typically involves the notations of cost and pruning, it may be beneficial if the instances exchange the better cost by message passing. This cooperative feature among instances would lead to more efficient pruning on all swarm instances, thus achieve less execution time the lower memory consumption. We developed four swarm algorithms based on random depth search:

P-RDFS is the basic swarm algorithm where all UPPAAL instances do RDFS independently in parallel with diverse random seeds. Local pruning (avoid exploring states that have a cost worse than the so-far best solution) within each instance is enabled. Any instance that completes first stops other peers.

S-RDFS is a cooperative version of P-RDFS where all instance exchange better costs by message passing.

S-Mix is a variant of S-RDFS where one instance runs BFS rather than RDFS.

S-Agent is an agent based version where one BFS UPPAAL instance serve as root node, while all other instances are agents running RDFS. The root takes charge of termination. Only the root starts search from the initial state. An agent requests a state as a task from the root to search from. When a certain time limit is met, the agent will ask for a new state.

RDFS is good starting point because it is effective in finding sequences of usable (not necessarily optimal) solutions in very large inexhaustible state-spaces [6]. We compare S-RDFS with P-RDFS for evaluating the benefit of sharing costs among swarm instances. The motivation for S-Mix and S-Agent is, as also found by Behrmann in [8], that in UPPAAL BFS often completes much faster than DFS/RDFS because DFS/RDFS can cause higher degree of fragmentation of the underlying symbolic state-space requiring many more symbolic states. On the other hand, BFS has an inherent drawback that it typically only reports results late when it has searched nearly all states, making it infeasible for very large state-spaces. S-Mix and S-Agent algorithms combine advantages of BFS and RDFS such that RDFS can report usable solutions fast and BFS may terminate fast.

We propose the following metrics to compare the algorithms:

Metric 1: time to find the optimal result (t_{opt}). The minimum run time for any UPPAAL instance to find the fastest trace (or schedule) to the goal. Users wish to get the optimal result fast even before an algorithm terminates.

Metric 2: time to terminate and thus prove the optimal result (t_{prov}). Users prefer an algorithm to terminate fast.

Metric 3: time to progressively improving solutions (a.k.a. *near optimal* solutions). It shows how fast results converge to the optimal as a function of runtime. In scheduling problems, the absolute optimal solution is not always required, but a sufficiently good one may suffice. Particularly when algorithms cannot terminate due to time or memory constrains, faster converge speed produces better near optimal results.

Metric 4: memory consumption when algorithms terminate normally or due to timeout. A smaller memory consumption improves scalability by enabling more parallel instances, or more available free memory to other instances.

Organization. The rest of the paper is structured as follows. Section 2 defines the timed automata and sequential TOR algorithm. Section 3 shows the swarm TOR algorithms. Section 4 compares the performance of sequential and swarm TOR algorithms using benchmark experiments on the cluster. Section 5 concludes.

2. SEQUENTIAL TIME OPTIMAL REACHABILITY

This section recalls the basic theory of timed automata and the sequential time-optimal reachability algorithm.

2.1 Timed Automata

Let $X = \{x, y, ...\}$ be a finite set of clocks. We define $\mathcal{B}(X)$ as the set of clock constraints over X generated by grammar: $g, g_1, g_2 ::= x \bowtie n \,|\, x - y \bowtie n \,|\, g_1 \wedge g_2$, where $x, y \in X$ are clocks, $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

DEFINITION 1. *A* Timed Automaton (TA) [2] *is a 6-tuple* $\mathcal{A} = (L, \ell_0, X, \Sigma, E, Inv)$ *where: L is a finite set of locations, $\ell_0 \in L$ is the initial location, X is a finite set of non-negative real-valued clocks, Σ is a finite set of actions, $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ is a finite set of edges, $Inv : L \to \mathcal{B}(X)$ sets an invariant for each location.*

DEFINITION 2. *The semantics of a timed automaton $\mathcal{A}$ is a* Timed Transition System (TTS) $S_{\mathcal{A}} = (Q, Q_0, \Sigma, \to)$ *where: $Q = \{(\ell, v) \,|\, (\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ and$v \models Inv(\ell)\}$ are states, $Q_0 = (\ell_0, 0)$ is the initial state, Σ is the finite set of actions, $\to \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$ is the transition relation defined separately for action $a \in \Sigma$ and delay $d \in \mathbb{R}_{\geq 0}$ as:*
(i) $(\ell, v) \xrightarrow{a} (\ell', v')$ if there is an edge $(\ell \xrightarrow{g,\alpha,r} \ell') \in E$ such that $v \models g$, $v' = v[r \mapsto 0]$ and $v' \models Inv(\ell')$,
(ii) $(\ell, v) \xrightarrow{d} (\ell', v + d)$ such that $v \models Inv(\ell)$ and $v + d \models Inv(\ell)$.

DEFINITION 3. *A* trace *ρ of $\mathcal{A}$ can be expressed in $S_{\mathcal{A}}$ as a sequence of alternative delay and action transitions starting from the initial state: $\rho = q_0 \xrightarrow{d_1} q'_0 \xrightarrow{a_1} q_1 \xrightarrow{d_2} q'_1 \xrightarrow{a_2} \cdots \xrightarrow{d_n} q'_{n-1} \xrightarrow{a_n} q_n \cdots$, where $a_i \in \Sigma$, $d_i \in \mathbb{R}_{\geq 0}$, q_i is state (ℓ_i, v_i), and q'_i is reached from q_i after delay d_{i+1}. State q is* reachable *if there exists a finite trace with the final state q. Let $Exec_{\mathcal{A}}$ denotes the set of traces of $\mathcal{A}$ and $Exec^f_{\mathcal{A}}$ denotes the set of finite traces.*

DEFINITION 4. *The span of a finite trace $\rho \in Exec^f_{\mathcal{A}}$ is defined as the finite sum $\Sigma_{i=1}^n d_i$. For a given state (ℓ, v), the* minimum span *of reaching the state* `MinSpan`(ℓ, v) *is the infimum of the spans of finite traces ending in (ℓ, v). For a given location ℓ, the* minimum span *of reaching the location* `MinSpan`(ℓ) *is the infimum of spans of finite traces ending in (ℓ, v) for all possible v.*

2.2 Sequential Time Optimal Reachability Algorithm

The semantics of TA will result in an infinite transition system. Real-time model checkers therefore build a finite state abstraction of the transition system. UPPAAL works by exploring a finite *symbolic reachability graph*, where the nodes are *symbolic states*. A symbolic state is a pair (ℓ, Z), where $\ell \in L$ is a location, and $Z = \{v \mid v \models g_z, g_z \in \mathcal{B}(X)\}$ is a convex set of clock valuations called *zone* [16], which is normally efficiently represented and stored in memory as *difference bound matrices* (DBM) [9].

DEFINITION 5. *The* cost function *on a symbolic state* (ℓ, Z) *is defined as* $\texttt{MinCost}(\ell, Z) = \texttt{inf}(\texttt{MinSpan}(\ell, v))$ *where* $v \in Z$. *It is the span of a finite symbolic trace ending in* (ℓ, Z).

Algorithm 1: Sequential Time Optimal Reachability

```
   PASSED ⟵ ∅, WAITING ⟵ {(ℓ0, Z0)}, COST ⟵ ∞
   ORDER ∈ {DFS, BFS, RDFS}
   Procedure Main()
1      while WAITING ≠ ∅ do
2          Search()
3      return COST
   Procedure Search()
4      select (ℓ, Z) from WAITING on ORDER
5      if (ℓ, Z) ⊨ Goal then
6          if MinCost(ℓ, Z) < COST then
7              COST ⟵ MinCost(ℓ, Z)
8      else if (ℓ, Z) ∉ PASSED and MinCost(ℓ, Z) < COST
       then
9          add (ℓ, Z) to PASSED
10         forall the (ℓ', Z') such that (ℓ, Z) ⇝ (ℓ', Z') do
11             add (ℓ', Z') to WAITING
```

The sequential TOR algorithm is shown in Algorithm 1. It is similar to standard reachability algorithm but with B&B support. WAITING and PASSED lists maintain the states waiting to be explored and states already explored respectively. Initially PASSED is empty and WAITING holds the initial state. COST records the current best result that is infinity in the beginning; and ORDER configures the search order. Until WAITING list becomes empty, procedure `Main` invokes the `Search` procedure in a loop. Inside `Search`, a state is picked out of WAITING according to ORDER. If the state is a goal state with a lower cost than the current COST at line 6, COST is updated. Meanwhile a near optimal solution to the goal is found. If the state is not goal, it is subject to symbolic state inclusion checking and B&B elimination rule at line 8. The state is pruned (discarded here) if either it is included in PASSED or its cost function is no less than the current COST. Otherwise, it is added to PASSED and its successors are added to WAITING.

3. SWARM TIME OPTIMAL REACHABILITY

This section describes the four swarm algorithms which are extended from the sequential TOR algorithm. In the basic version (P-RDFS), none swarm instances exchange costs. In the cooperative versions (S-RDFS, S-Mix and S-Agent), swarm instances will exchange the latest better costs they have found.

3.1 Basic Swarm RDFS Algorithm

Algorithm 2 shows the basic swarm RDFS TOR algorithm executed on all computing nodes. It is in fact the sequential RDFS TOR algorithm (in Algorithm 1) with different random seed run in parallel. Any instance that finishes the state-space exploration first stops the other instances (at line 3) whose `Terminate` flag will be set to true.

Algorithm 2: Basic Swarm RDFS Time Optimal Reachability (P-RDFS)

```
   PASSED ⟵ ∅, WAITING ⟵ {(ℓ0, Z0)}, COST ⟵ ∞
   ORDER ⟵ RDFS, TERMINATE ⟵ FALSE
   Procedure Main()
1      while WAITING ≠ ∅ and ¬TERMINATE do
2          Search()
3      stop other instances
4      return COST
```

3.2 Cooperative Swarm RDFS Algorithm

Algorithm 3 shows the cooperative swarm RDFS TOR algorithm. All instances do RDFS, meanwhile collaborate by exchanging better costs they have found. This algorithm resembles Algorithm 2 in the data structures PASSED, WAITING, COST, and the local search procedure `Search` at line 4. Therefore Algorithm 3 mainly highlights the additional communication supplement. A new variable ECOST maintains the external better cost received from the network and is initialized to infinity. The UPDATE message carries a new better cost. At the beginning of each iteration inside `Main`, `Ecost` compares with `Cost`. When `Ecost` is smaller, a better cost has been found by another instance, and `Cost` is updated to this. When `Cost` is smaller, the current instance found a better cost, and `Cost` is assigned to `Ecost` by procedure `Update` and broadcasted. Line 8 handles reception of a cost from the network, and updates `Ecost`.

3.3 Cooperative Swarm Mix Algorithm

Algorithm 4 shows the cooperative swarm Mix TOR algorithm. It differs from Algorithm 3 only in the search order configuration on one instance. If the process id p equals 0, that instance will do BFS rather than RDFS.

3.4 Cooperative Swarm Agent Algorithm

Algorithm 5 shows the swarm agent TOR algorithm. Its main differences from the two preceding algorithms are: (1) only the root starts search from the initial state, an agent periodically requests a new state from root to search from; (2) only the root takes charge of termination. The root invokes `MainRoot` doing BFS. Agents invoke `MainAgent` doing RDFS. The message types are extended with (1) REQUEST for agents to ask states from the root, and (2) ISSUE for root to send a state to an agent.

In `MainAgent`, `Iteration` is self-increased in every iteration, and is compared with a threshold `Limit` at line 10 to decide if this agent should request a new state from root. Because initially the `Waiting` list of an agent is empty, it

Algorithm 3: Cooperative Swarm RDFS Time Optimal Reachability (S-RDFS)

(Local Variables)
PASSED $\longleftarrow \emptyset$, WAITING $\longleftarrow \{(\ell_0, Z_0)\}$
COST $\longleftarrow \infty$, ECOST $\longleftarrow \infty$
ORDER $\longleftarrow$ RDFS, TERMINATE $\longleftarrow$ FALSE

(Message Types)
UPDATE

Procedure `Main()`
1 **while** WAITING $\neq \emptyset$ ***and*** $\neg$TERMINATE **do**
`// lines [2,3] are called atomically`
2 **if** ECOST $<$ COST **then** COST $\longleftarrow$ ECOST
3 **if** COST $<$ ECOST **then**
`Update(`COST`)`,
`Broadcast(`UPDATE, COST`)`
4 `Search()`
5 stop other instances
6 **return** COST

Procedure `Update(`NEWCOST`)` `// called atomically`
7 **if** NEWCOST $<$ ECOST **then** ECOST $\longleftarrow$ NEWCOST

(Message Processing Rules)
8 When a node receives UPDATE$\langle$NCOST$\rangle$, `Update(`NCOST`)`.

Algorithm 4: Cooperative Swarm Mix Time Optimal Reachability (S-Mix)

(Local Variables)
$$\text{ORDER} \longleftarrow \begin{cases} \text{BFS} & \text{if } p = 0, \\ \text{RDFS} & \text{otherwise.} \end{cases}$$

The rest is the same as Algorithm 3

will request a state and wait to receive the issued state at line 6. This line also ensures an agent never exit prematurely if it exhausts its `Waiting` list too fast. Finally, the message handing rules are expanded for REQUEST and ISSUE messages. Once receive a REQUEST message, the root selects an unexplored waiting state, wraps it in an ISSUE message, and sends to a agent. Once receive an ISSUE message, the agent clears its `Passed` and `Waiting` lists, extracts the issued state from the message, and inserts this state into `Waiting`.

4. EXPERIMENTS

We developed a new version of UPPAAL implementing the swarm algorithms using the MPI library. P-RDFS was relatively simple to implement because UPPAAL only needed to be modified to handle coordination of termination. S-RDFS and S-Mix required a moderate amount of modification to also exchange cost. S-Agent was most involved as it also required communication of symbolic states and careful interaction with UPPAAL memory management. We ran our experiments on a cluster with 9 computing nodes, each having 1 Tb memory (NUMA architecture) and 64 cores at the frequency of 2.3GHz (4 AMD Opteron 6376 Processors each with 16 cores), a 1 SATA Tb disk and Infiniband interconnection. Each sequential algorithm was executed 15 runs on

Algorithm 5: Cooperative Swarm Agent Time Optimal Reachability (S-Agent)

(Local Variables)
PASSED $\longleftarrow \emptyset$, COST $\longleftarrow \infty$, ECOST $\longleftarrow \infty$
$$\text{WAITING} \longleftarrow \begin{cases} \{(\ell_0, Z_0)\} & \text{if } p = root, \\ \emptyset & \text{otherwise.} \end{cases}$$
$$\text{ORDER} \longleftarrow \begin{cases} \text{BFS} & \text{if } p = root, \\ \text{RDFS} & \text{otherwise.} \end{cases}$$

(Message Types)
UPDATE, REQUEST, ISSUE

Procedure `MainRoot()`
1 **while** WAITING $\neq \emptyset$ **do**
same as lines [2-4] in Algorithm 3
2 stop all agents
3 **return** COST

Procedure `MainAgent()`
4 ITERATION $\longleftarrow 0$, LIMIT $\in \mathbb{N}$, TERMINATE $\longleftarrow$ FALSE
5 **while** $\neg$TERMINATE **do**
6 **if** WAITING $= \emptyset$ **then**
`Send(`REQUEST, Root`)`, `Recv(`ISSUE`)`
`// lines [7,8] are called atomically`
7 **if** ECOST $<$ COST **then** COST $\longleftarrow$ ECOST
8 **if** COST $<$ ECOST **then**
`Update(`COST`)`,
`Broadcast(`UPDATE, COST`)`
9 `Search()`, ITERATION $\longleftarrow$ ITERATION $+ 1$
10 **if** ITERATION $\geq$ LIMIT **then**
11 `Send(`REQUEST, Root`)`, `Recv(`ISSUE`)`,
ITERATION $\longleftarrow 0$

(Message Processing Rules)
12 When a node receives UPDATE$\langle$NCOST$\rangle$: `Update(`NCOST`)`.
13 When root receives REQUEST from agent i: select (ℓ', Z') from WAITING, `Send(`ISSUE$, (\ell', Z'), i)$.
14 When an agent receives ISSUE$\langle(\ell^*, Z^*)\rangle$ from root: WAITING $\longleftarrow \emptyset$, PASSED $\longleftarrow \emptyset$, WAITING $\longleftarrow \{(\ell^*, Z^*)\}$.

a single core. Swarm algorithms were executed 15 runs for every core setting: 2, 4, 8, 16, 32, 64, 128, 256, 512.

4.1 Models

The first three UPPAAL models are up-scaled versions of those in normal UPPAAL distribution. The fourth model is transformed from a task graph benchmark[1].

Job-Shop-6 (jb-6). Six people want to read a single piece of four-section newspaper. Each person has his own preferred reading sequence, and can spend different time on each section. When one person is reading a section, others who are also interested in it must wait. The objective is find the time optimal schedule for all six people to finish reading.

Aircraft-Landing-15 (alp-15). 15 aircrafts need to land on two runways. Each aircraft has a preferred target landing time. It can also speed up and land earlier or stay longer in the air and land later if necessary. Furthermore, aircrafts cannot land back to back on the same runway due to wake turbulence by the previous aircraft. Thus there

[1]http://www.kasahara.elec.waseda.ac.jp/schedule/stgarc_e.html

Table 1: Runtime (sec) of Job-Shop-6 and Aircraft-Landing-15

	Job-Shop-6								Aircraft-Landing-15							
#C	BFS		DFS		RDFS				BFS		DFS		RDFS			
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}			t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}		
1	108	108	7877	9899	650	2283			157	159	73	428	191	964		
#C	P-RDFS		S-RDFS		S-Mix		S-Agent		P-RDFS		S-RDFS		S-Mix		S-Agent	
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}
2	535	1996	459	1821	102	102	39	99	87	845	93	873	91	91	66	90
4	310	1730	280	1683	102	102	25	98	42	737	35	729	46	90	21	91
8	176	1720	116	1653	101	102	17	97	22	712	16	727	17	89	25	88
16	100	1591	87	1506	99	101	14	97	17	676	11	699	11	89	6	86
32	65	1488	52	1494	64	99	13	96	2	638	1	633	2	89	12	86
64	32	1452	24	1385	24	98	16	95	<1	604	<1	582	<1	88	5	85
128	4	1412	4	1367	4	98	29	95	<1	597	<1	620	<1	88	2	84
256	2	1365	<1	1341	<1	98	28	95	<1	581	<0.1	565	<0.1	88	7	84
512	<1	1337	<1	1263	<1	100	31	96	<1	585	<0.1	573	<0.1	117	5	86

are certain minimum constraints on the separation delay between aircrafts of different sizes. The objective is to find the time optimal schedule for all aircrafts to land safely.

Viking-Bridge-15 (vik-15). 15 vikings want to cross a bridge in the darkness. The bridge is damaged and can only carry two people at the same time. To find the way over the bridge the vikings need to bring a torch, but the group has only one torch to share. The 15 members of the group need different time to cross the bridge (one-way), which for simplicity is classified into four levels: 5, 10, 20 and 25 time units. The objective is to find the time optimal schedule for those 15 vikings to cross the bridge safely.

Task-Graph-88 (task-88). A robot control program has 88 computational tasks each of which has precedence constraints (predecessor tasks) among [0,3] and processing time among [1,111]. A task can start only if all its predecessor tasks complete. Now the control program is going to be assigned to four processors at the speeds of [1,1,2,4]. The objective is to determine a non-preemptive schedule that minimizes the time for all tasks to terminate.

4.2 Time to Find or Prove Optimal Result (Metric 1 & 2)

Tables 1 and 2 show for the first three models the median runtime to reach optimal cost (t_{opt} corresponding to metric 1), and runtime to prove optimal cost (t_{prov} corresponding to metric 2). We want to know how swarm algorithms scale as employing an increasing number of cores denoted by #C (#C=1 for sequential algorithms). We set a 4-hour time bound for the experiments, and use "-" to denote timeout.

Job-Shop-6. Comparing sequential algorithms, BFS is the fastest both to reach and to prove optimal cost. DFS is slowest for both. RDFS is six times slower than BFS in t_{opt}, and 21 times slower in t_{prov}.

P-RDFS and S-RDFS steadily decrease in columns t_{opt} and t_{prove} with increasing number of cores. Above 16 cores they outperform BFS in t_{opt}. Below 64 cores S-RDFS works on average 20% faster than P-RDFS due to sharing costs by messages but gradually saturates afterward. The proving time t_{prov} improves slower than t_{opt}. The reason is that swarm algorithms do not divide its work load among cores (no data parallelism) so one instance must alone complete the proof, and we cannot expect a big speed up. However, an improvement is still gained from the increase in probability that some instance finds a low cost solution fast, resulting in more effective pruning.

Compared with S-RDFS, S-Mix has better t_{opt} at lower number of cores (2 to 8), because the BFS node works fast and dominates. When more RDFS nodes join, they search deep down via different paths to the goal in parallel and report better costs more frequently. Above 16 cores, RDFS start to dominate t_{opt}. The other notable improvement is that t_{prov} drops dramatically because the BFS node completes first and terminates the job. S-Agent has the best t_{opt} from 2 to 32 cores, but superseded by other swarm algorithms afterwards. This shows distributing states to agents can speed up t_{opt} when agents do not overload the root.

Aircraft-Landing-15. Surprisingly for sequential algorithms, DFS has the fastest t_{opt}, and RDFS has the slowest t_{opt} and t_{prov}. All swarm algorithms have faster t_{opt} than the sequential algorithms above 4 cores, and S-RDFS and S-Mix are equally the best. Regarding t_{prov}, S-Mix and S-Agent are equally the best, and is even two times faster than BFS. This implies that the better costs reported by the RDFS nodes help the BFS node in pruning.

Viking-Bridge-15. This model shows a unique behavior. It is good for BFS but extremely bad for DFS/RDFS. BFS/S-Mix/S-Agent can complete within 5 minutes. But other algorithms are subject to the 4-hour timeout. As log files show this model produce solutions with a very wide cost spectrum ranging from 638195 to 220 with the decrement steps of just one. Consequently, the RDFS nodes in S-Mix and S-Agent cannot help but disturb the BFS root by reporting enormous messages containing only very fine-grained improved costs, which will activate pruning at BFS root very frequently. Therefore S-Mix and S-Agent perform worse than BFS in t_{opt} and t_{prov}.

Task-Graph-88. The table for this model is absent because the model is too large for all algorithms to complete within the 4-hour time bound. However, we show how the near optimal solutions with costs converge as a function of runtime in section 4.3.

Conclusions. (1) By comparing P-RDFS and S-RDFS, we conclude that exchanging costs can in general speed up finding and proving optimal result, especially show its usefulness at low core settings. (2) S-Mix and S-Agent can combine the benefits of BFS and RDFS. Thus, they can report

Table 2: Runtime (sec) of Viking-Bridge-15

#C	BFS		DFS		RDFS			
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}		
1	70	70	-	-	-	-		
#C	P-RDFS		S-RDFS		S-Mix		S-Agent	
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}
2	-	-	-	-	96	96	75	75
4	-	-	-	-	87	87	72	72
8	-	-	-	-	96	96	72	72
16	-	-	-	-	92	92	83	83
32	-	-	-	-	100	100	94	94
64	-	-	-	-	107	107	77	77
128	-	-	-	-	112	112	74	74
256	-	-	-	-	186	186	71	71
512	-	-	-	-	215	215	73	73

"-": denotes timeout.

results and terminate faster than other swarm algorithms. (3) Costs reported by RDFS nodes can help BFS root in pruning, but fine-grained improved costs may backfire.

4.3 Results versus Time (Metric 3)

Figures 1 and 2 show how near optimal results improve with runtime for models viking-bridge-15 and task-graph-88. We set the sample window to the first 2 minutes when runs start. For swarm algorithms, we look at the intermediate core setting of 32.

Job-Shop-6. Among sequential algorithms, BFS does not report any result until at 108 sec gives the optimal cost of 62. DFS reports no results within the sample window, whereas RDFS reports solutions with costs from 72 to 68. The swarm algorithms report useful near optimal costs immediately after start. S-Agent is the fastest in finding the optimal cost of 62 as early as 13 sec. Other algorithms are equally good at finding near optimal costs; and they all find the optimal cost faster than BFS.

Aircraft-Landing-15. Similar pattens as job-shop-6.

Viking-Bridge-15. BFS alone finds the optimal cost of 220 at 70 sec, while DFS/RDFS/P-RDFS/S-RDFS find costs far exceeding 220 in the sample window. The BFS root of S-Mix and S-Agent finds the optimal cost 30% slower than sequential BFS, then terminates all RDFS nodes. Clearly the BFS root is loaded by the RDFS nodes reporting messages containing fine-grained improved costs, indicating a point for implementation optimization.

Task-Graph-88. BFS cannot give any result within the 4-hour time limit. DFS and RDFS report some near optimal costs immediately after start. All swarm algorithms find higher quality results than the sequential algorithms, and they seem to perform equally well.

Conclusions. Swarm algorithms are generally faster than sequential algorithms at finding near optimal results; and they work equally well.

4.4 Memory Consumption (Metric 4)

Table 3 shows the average peak resident memory of a UPPAAL instance when it terminates normally (cells in roman font) or due to the 4-hour timeout (cells in *italic* font). We likewise use the core setting of 32 for swarm algorithms. The resident memory roughly reflects the number of symbolic states explored and kept in the passed list.

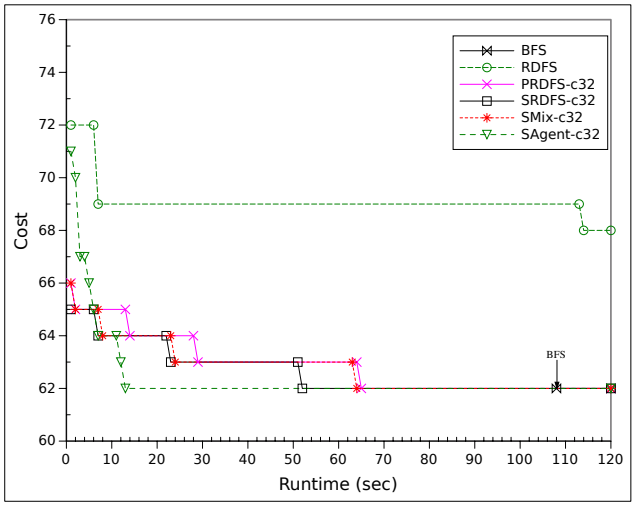

Figure 1: Cost vs. Runtime for Job-Shop-6

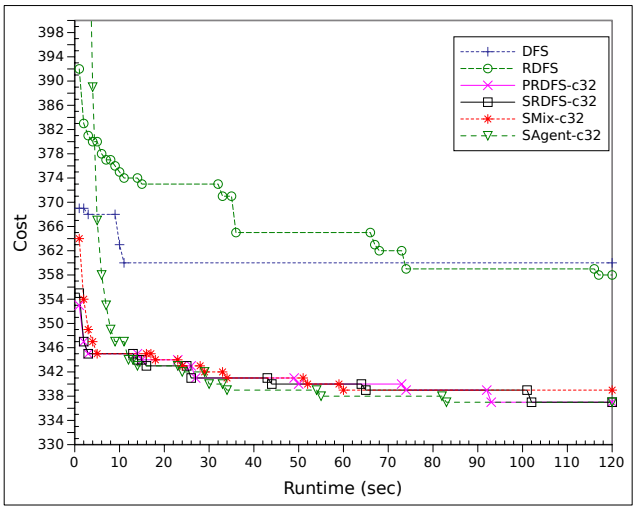

Figure 2: Cost vs. Runtime for Task-Graph-88

For all four models, S-Agent has the smallest average memory footprint overwhelmingly, because agents clear their state-spaces when receiving a new state from the BFS root. It is followed by S-Mix in the first three models because within the same amount of runtime from the start to BFS root terminates, a RDFS node typically uses less memory

Table 3: Resident Memory (MB) per UPPAAL Instance

Models	jb-6	alp-15	vik-15	task-88
BFS	152	124	408	*5068*
DFS	1754	127	*13297*	*3060*
RDFS	606	191	*12790*	*3668*
P-RDFS-c32	593	187	*13555*	*3849*
S-RDFS-c32	591	190	*15303*	*3873*
S-Mix-c32	63	51	230	*3914*
S-Agent-c32	21	13	100	*1277*

Italic font denotes termination due to timeout.

than the BFS root, thus amortizes the average. Consider the task-88 model where all algorithms stop at 4-hour limit; BFS has the largest memory consumption, around 1.4 times more.

Conclusions. S-Agent has the best average memory footprint per UPPAAL instance.

5. CONCLUSIONS

We proposed using swarm verification for time optimal reachability analysis. We developed four swarm algorithms and performed four benchmark experiments in terms of scalability, time- and memory consumption. Based on the evaluation we conclude that this approach is very promising. In particular, swarm algorithms generally find optimal (or near optimal) results much faster than sequential algorithms; S-Mix and S-Agent combines the benefits of BFS and RDFS such that they can find results and terminate fast; S-Agent has smaller memory footprint because agents do not keep the state-space; exchanging costs is beneficial for speed up at lower core settings. For the future work, we will develop the time optimal reachability algorithms that partition the state-space among the compute nodes. We would also extend swarm algorithms to more general priced time automata.

6. ACKNOWLEDGMENTS

This work has been supported by Danish National Research Foundation – Center for Foundations of Cyber-Physical Systems, a Sino-Danish research center.

7. REFERENCES

[1] Y. Abdeddaim, E. Asarin, and O. Maler. Scheduling with timed automata. *Theoretical Computer Science*, 354(2):272 – 300, 2006.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *HiBi/PDMC*, pages 4–7. IEEE, 2010.

[4] G. Behrmann. Distributed reachability analysis in timed automata. *STTT*, 7(1):19–30, 2005.

[5] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *SFM*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.

[6] G. Behrmann and A. Fehnker. Efficient guiding towards cost-optimality in uppaal. In *TACAS*, volume 2031 of *LNCS*, pages 174–188. Springer, 2001.

[7] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. W. Vaandrager. Minimum-cost reachability for priced timed automata. In *HSCC*, volume 2034 of *LNCS*, pages 147–161. Springer, 2001.

[8] G. Behrmann, T. Hune, and F. W. Vaandrager. Distributing timed model checking - how the search order matters. In *CAV*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.

[9] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003.

[10] S. Blom, J. van de Pol, and M. Weber. Ltsmin: Distributed and symbolic reachability. In *CAV*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010.

[11] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *CAV*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998.

[12] A. Fehnker. Scheduling a steel plant with timed automata. In *RTCSA*, pages 280–286. IEEE Computer Society, 1999.

[13] A. Fehnker. *Bounding and Heuristics in Forward Reachability Algorithms*. UB Nijmegen [Host], 2000.

[14] G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification. In *ASE*, pages 1–6. IEEE, 2008.

[15] T. Hune, K. G. Larsen, and P. Pettersson. Guided synthesis of control programs using uppaal. In *ICDCS Workshop*, pages E15–E22, 2000.

[16] K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *FCT*, volume 965 of *LNCS*, pages 62–88. Springer, 1995.

[17] P. Niebert, S. Tripakis, and S. Yovine. Minimum-time reachability for timed automata. In *IEEE Mediteranean Control Conference*, 2000.

[18] J. I. Rasmussen, G. Behrmann, and K. G. Larsen. Complexity in simplicity: Flexible agent-based state space exploration. In *TACAS*, pages 231–245. Springer, 2007.