Time and Cost Optimization of Cyber-Physical Systems by Distributed Reachability Analysis

Zhang, Zhengkui

DOI (link to publication from Publisher):
10.5278/vbn.phd.engsci.00085

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

TIME AND COST OPTIMIZATION OF CYBER-PHYSICAL SYSTEMS BY DISTRIBUTED REACHABILITY ANALYSIS

BY
ZHENGKUI ZHANG

DISSERTATION SUBMITTED 2016

AALBORG UNIVERSITY
DENMARK

Time and Cost Optimization of Cyber-Physical Systems by Distributed Reachability Analysis

Ph.D. Dissertation

Zhengkui Zhang

Dissertation submitted December 20, 2016

Dissertation submitted: December 20, 2016

PhD supervisor: Prof. Kim Guldstand Larsen
Aalborg University, Denmark

Assistant PhD supervisor: Assoc. Prof. Brian Nilsen
Aalborg University, Denmark

PhD committee: Associate Professor Josva Kleist (chairman)
Department of Computer Science
Aalborg University, Denmark

Senior researcher, Radu Mateescu
Inria Grenoble - Rhône-Alpes / CONVECS
Inovallée, France

Associate Professor, Bernhard Klaus Aichernig
Institute of Software Technology
Graz University of Technology, Austria

PhD Series: Faculty of Engineering and Science, Aalborg University

ISSN (online): 2246-1248
ISBN (online): 978-87-7112-852-9

Published by:
Aalborg University Press
Skjernvej 4A, 2nd floor
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Rosendahls, 2017

Abstract

There is no doubt that real-time systems are ubiquitous in our information society. They are the brains of many critical systems, and becoming more pervasive with the rise of cyber-physical systems and industry 4.0. Incorrect designs of those critical systems can result in a significant loss of money for re-design and fixing, or more seriously in catastrophe for system safety. The challenge summons a new development methodology that should be more rigorous and precise. Therefore "model-based systems engineering" based on a series of automated model-based formal methods has been proposed to facilitate design, analysis, optimization, verification and validation activities.

Network of timed automata is an elegant framework for modeling real-time behaviors, constraints and interactions between concurrent components of a real-time reactive system in a natural way. Timed automata have been widely used as the input models for real-time model checking. Time optimal reachability analysis is a novel use of timed automata for solving scheduling and planning problems in a static environment. However it also inherits state-space explosion from model checking. The thesis develops distributed algorithms for time optimal reachability to efficiently accelerate finding the optimal results and battle the state-space explosion.

An extended form of time optimal scheduling is multi-objective scheduling regarding a variety of quantitative objectives. The thesis extends timed automata with discrete prices on transitions, and develops algorithms for Pareto optimal reachability analysis to compute a set of schedules that are in Pareto optimum. Engineers can choose a schedule that is close to their preference and optimized for all objectives in the best balance.

Stochastic hybrid automata and timed game automata are extended formalisms of timed automata that are able to model more complexities about a real-time reactive system as well as handling unpredictable environments. They are the input models for statistical model checking and controller synthesis separately. The thesis integrates controller synthesis and (statistical) model checking, such that users can conveniently verify a synthesized strategy for additional correctness properties by model checking, and evaluate the performance aspects of the strategy by statistical model checking.

Resumé

Real-tidssystemer er allestedsværende i vores moderne informationssamfund, hvor de forestår den centrale styring af mange kritiske systemer. De har stigende udbredelse i takt med introduktionen af cyber-fysiske systemer og industri 4.0. Forkert design af sådanne kritiske systemer kan resultere i betydelige økonomisk omkostninger til re-design og fejl korrektion, eller værre, i manglende system sikkerhed. Denne udfordring kræver nye udviklingsmetoder, der er mere omhyggelige og præcise. Forskere har foreslået "model-based system engineering", som baserer sig på en række af automatiserede model-baserede formelle metoder til at støtte design, analyse, optimering, verifikation og validering.

Netværk af tidsautomater er en elegant og naturlig formalisme til modellering af real-tidsadfærd, krav, og interaktioner imellem samtidige systemkomponenter. Ligeledes understøttes tidsautomater modeller af værktøjer til model-check analyse. Tidsoptimal opnåelighedsanalyse er en ny anvendelse af tidsautomater til løsning af schedulerings- og planlægningsproblemer i statiske omgivelser. Denne teknik arver dog problemet med eksplosion i størrelsen på modellens tilstandsrum fra model-check. Denne afhandling udvikler nye distribuerede algoritmer til tidsoptimal opnåelighedsanalyse for at accelerere beregningen af optimale resultater, og for at reducere tilstandsrums-eksplosions problemet.

Afhandlingen adresserer endvidere en udvidet form for tidsoptimal planlægning, hvor multiple kvantitative mål indgår. I afhandlingen udvides tidsautomater med multiple diskrete priser på transitioner, og der udvikles optimeringsalgoritmer, der beregner mængden af Pareto optimale planer. Ingeniører kan derefter vælge den plan, der er bedst i forhold til deres præferencer, og som giver den bedste balance iblandt de indgående mål.

Stokastiske hybride automater og timed game automater er udvidelser til tidsautomater, der håndterer komplekse reaktive real-tidssystemer og uforudsigelige omgivelser. De anvendes som input til værktøjer til henholdsvis statistisk model-check og kontrol syntese. Denne afhandling integrerer kontrol syntese og (statistisk) model-check således, at brugere bekvemt kan verificere en synteseret strategi for yderligere korrekthedsegenskaber ved hjælp

af model-check, og kan evaluere dens performance ved hjælp af statistisk model-check.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Prof. Kim Guldstand Larsen for providing me the opportunity to learn and discover the area of formal methods on real-time systems, for the continuous support of my PhD study and related research, for his patience, motivation, guidance and immense knowledge.

The same earnest appreciation also goes to my co-supervisor Assoc. Prof. Brian Nielsen for his encouragement got me out of the slump, for his patient and enlightening guidance helped me in all the time of research and writing of this thesis. His office was always open for a discussion during my research or writing. I could not have imagined having a better advisor than Prof. Larsen and Assoc. Prof. Nielsen for my PhD study.

My sincere thanks also go to Alexandre David, who was my former co-supervisor and is now working in USA, for his technical instructions during the first half of my PhD study. He continued to give me consulting supports during the second half of my study.

Besides my advisors, I am grateful for my co-authors Prof. Holger Hermanns, Gilles Nies, Marvin Stenger and Huixing Fang. We had happy collaboration experiences together. They also reviewed the draft papers and gave me valuable comments.

I also appreciate the happy experiences with other colleagues at the CISS group and the administrative staffs at the department: Marius Mikucionis, Assoc. Prof. Radu Mardare, Assoc. Prof. Jiri Srba, Giorgio and Giovanni Bacci, Danny Bøgsted Poulsen, Andreas Engelbredt Dalsgaard, Rikke W. Uhrenholt, Susanne T. Larsen, Helle Schroll and Helle Westmark.

Last but not the least, I would like to thank my family: my parents and my wife for supporting me spiritually throughout the research, thesis writing and my life in general.

Zhengkui Zhang
Aalborg, Denmark
December 20, 2016

Contents

Thesis Details

Thesis Title:	Time and Cost Optimization of Cyber-Physical Systems by Distributed Reachability Analysis
PhD Student:	Zhengkui Zhang
Supervisors	Prof. Kim Guldstand Larsen, Aalborg University Assoc. Prof. Brian Nilsen, Aalborg University

The thesis is organized in two parts. Part I is an overview of this PhD work. Part II describes important design and implementation details. Part III consists of the following four papers.

[A] *Time Optimal Reachability using Swarm Verification*. Zhengkui Zhang, Brian Nielsen, and Kim G. Larsen. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, pages 1634–1640, ACM 2016.

[B] *Distributed Algorithms for Time Optimal Reachability Analysis*. Zhengkui Zhang, Brian Nielsen, and Kim G. Larsen. In Proceedings of Formal Modeling and Analysis of Timed Systems - 14th International Conference FORMATS 2016, volume 9884 of Lecture Notes in Computer Science, pages 157–173, Springer 2016.

[C] *Pareto Optimal Reachability Analysis for Simple Priced Timed Automata*. Zhengkui Zhang, Brian Nielsen, Kim G. Larsen, Gilles Nies, Marvin Stenger and Holger Hermanns. Under submission.

[D] *Verification and Performance Evaluation of Timed Game Strategies*. Alexandre David, Huixing Fang, Kim G. Larsen and Zhengkui Zhang. In Proceedings of Formal Modeling and Analysis of Timed Systems - 12th International Conference, FORMATS 2014, volume 8711 of Lecture Notes in Computer Science, pages 100–114, Springer 2014.

Part I

Overview

Overview

1 Introduction

1.1 Model-Based Engineering

Real-time systems are ubiquitous in our information society. Particularly, they are the brains of many critical systems such as vehicles, avionics, robotics, traffic control, financial transactions, industrial applications, and telecommunication infrastructures etc. The correct behavior of these systems depends not only on the logic order of the performed events but also on their timing [AILS07]. For instance, the precise real-time temperature control is critical for chemistry plants, the high-speed rail network must have accurate real-time monitoring and control over the vast simultaneously fast moving trains, and the upcoming 5G mobile networks standard imposes extremely high real-time responsive requirement on the base stations and switches.

Industry is often haunted with the challenge that incorrect designs of safety critical systems made on early stages can result either in catastrophic consequences for system safety or in a significant loss of money for re-design and fixing or both [DRA15]. *Model-based systems engineering* (MBSE) is based on a series of automated model-based formal methods to facilitate design, verification, validation, quality assurance, performance evaluation or optimization during the whole system development life cycle [HB13]. These methods are supported by techniques that include but are not limited to *design space exploration*, *model checking*, *runtime verification*, *model-based testing*, *statistical model checking*, *controller synthesis* etc. Effective use of these supporting techniques has been proposed to assist industry to improve automation, productivity, quality, management, and time-to-market.

Design space exploration discovers and evaluates different design alternatives of system construction and configuration, then selects the optimal one for a particular application according to the given requirements [DRA15]. Making sure a system is bug free is of vital importance. Several methods are available to ensure reliability and safety of systems including: testing, model checking, runtime verification and controller synthesis. Testing shows the

presence of bugs very effectively on the real implementation. Model-based testing [Rus05] is a technology for automated test case generation from models. Model checking [Cla08] is a formal verification method that guarantees a verified system model always satisfies the formalized correctness and safety properties. Runtime verification [BLS11] is a lightweight verification technique complementing model checking and testing. It avoids the complexity of – but has less coverage than – model checking. It is also applicable to black box systems and allows executing user-defined code. Controller synthesis [MPS95] automatically computes a control strategy for a game model, and hence may relieve the design complexity of reactive systems. The synthesized strategy is guaranteed to supervise the prototyped system under control to achieve a certain control objective while handling environment uncertainties in the model. Finally, statistical model checking [You05] uses statistical inference on simulated runs of a stochastic model to make probability estimation or hypothesis testing of properties. Engineers can predict reliability or performance aspects of the system behaviors at the model level before the implementation phase.

1.2 Scheduling & Planning

Scheduling and planning is an important optimization activity in embedded systems engineering, production, logistics etc. In a nutshell, scheduling and planning is about choosing and ordering a set of tasks in chronological order to achieve a certain goal, while resolving resource constrains between those tasks. A *schedule* is a table or plan comprising the starting time and access to resources for each allocated task. The scheduling problem is a very wide and active research area. It may be classified into basically two methodologies [AAM06, GNT16]: *domain-specific* or *domain-independent*.

Domain specific methods are tailor-made algorithms with the domain knowledge in mind to solve specific applications [GNT16] like flight navigation planning, logistic planning, or industrial process planning etc.

Many generic numeric optimization paradigms from operation research have been proposed to support solving domain-specific scheduling problems such as linear programming [Sch86], dynamic programming [DL77], genetic algorithm [Gol89], simulated annealing [KGV83] etc. But engineers still need to code domain specific constrains, heuristics, or transition relations between tasks into the optimization algorithm. The solution remains ad hoc, lacking a unified scheduling discipline. Particularly, this approach is not the most natural one for expressing some complex real-life scheduling situations [AAM06].

Domain-independent (also called *model-based*) methods describe a problem as a generic state transition model, then apply generic graph search algorithms in that model, providing more freedom and reusability [GNT16]. Therefore engineers can focus on modeling the scheduling problem itself.

Certain characteristics of the scheduling problem should be considered in order to choose the most appropriate modeling framework.

- **Time and concurrency.** Time is modeled explicitly to reason about durations, deadlines, time constraints, synchronizations, or handling concurrent activities. A temporal model such as the timed automata [AD94] is appropriate. A model-checking based planner reduces the scheduling problem on a temporal model into a time optimal reachability problem. A pre-planned static scheduler is obtained.
- **Uncertainty.** The model has non-deterministic events due to a dynamic environment, which cannot be anticipated by a scheduler. A game model with two players – controller and environment – is needed. A model based game solver reduces the scheduling problem into a reachability game problem. The generated strategy is dynamic, which determines the next move regarding the current situation of the game. The optimality may refer to minimizing the best or worst execution time.
- **Cost.** Tasks are typically associated with diverse quantitative properties expressing costs (due to the resource consumption, risk, weariness etc). Cost can be a discrete value on a transition, or proportional to the elapsing time of a task at a state with a given incremental rate. Cost optimal scheduling refers to finding a schedule that has the minimum accumulated costs for a given quantitative objective. If there are multiple quantitative objectives, it is a problem of multi-objective scheduling. There may not always exist any single solution that will simultaneously optimize all objectives if a subset of them are conflicting.
- **Stochasticity.** The model has stochastic behaviors due to the environment. A probabilistic model such as stochastic extension of timed automata, Markov decision processes [Bel57] etc can be chosen. The scheduling problem reduces to searching for a schedule that may minimize the expected execution time or cost according to the expected behaviors of the probabilistic model.

In this thesis, we mostly target at the time and cost optimal scheduling and planning problems for a fixed amount of tasks modeled by timed automata and simple priced timed automata. The environment is static (no uncertainty). The behaviors of components in the model are both action deterministic and time deterministic. Meanwhile the generated finite schedule has a full controllability over the components. However, we have also made contributions forward supporting controller synthesis problems modeled by timed game automata.

1.3 UPPAAL Tool Suite

UPPAAL [BDL04] is a mature and well-known model checker for real-time sys-

tems modeled as networks of *timed automata* [AD94]. It was originally developed in a collaboration between Uppsala University in Sweden and Aalborg University in Denmark since the mid-1990s. It has been successfully applied to solve many academic and industrial case studies, and used as an educational tool in classrooms of formal methods world wide. Nowadays, UPPAAL consists of a family of tools aiming at solving a wider range of MBSE problems using techniques that are inspired by, or derived from, model checking. Three of these techniques which are covered in this thesis are: *time optimal reachability analysis*, statistical model checking, and controller synthesis.

Time optimal reachability (TOR) analysis aims at finding time optimal schedulers for classical scheduling problems in the static environment. Around the year 2000 [AAM06, NTY00], researchers proposed that scheduling problems could be reformulated to the reachability problems modeled by timed automata. They were solved by finding traces with minimum span. Actually, this problem can also be regarded as a *one-player game*, where the scheduler has a full controllability over the tasks. TOR allows modeling of real-time behaviors, constrains and interactions of components in a natural way, as well as flexible choices of efficiently implemented search algorithms inside model checkers. Standard UPPAAL has an option for time optimal reachability.

Statistical Model Checking (SMC) [You05] refers to a series of simulation-based techniques that monitor the sample runs of a model, and use statistical algorithms to estimate the probability of the model to satisfy a certain property with a given level of confidence. It can handle *stochastic hybrid systems* which are either too complex or undecidable for classical model checking. A hybrid system refers to a dynamic system that exhibits continuous and discrete (or even stochastic) behaviors, which can be modeled by *hybrid automata* [DDL$^+$12], where clocks have different rates (even potentially negative) in different locations. UPPAAL-SMC [DLL$^+$11b] is a statistical model checking extension of standard UPPAAL, typically used for reliability and performance analysis of a stochastic hybrid model.

Since 1990s, controller synthesis (SYN) has become a topic of growing interest as a method for automatic design of reactive systems. It can automatically synthesize a controller for the reactive system to interact with its environment, and to achieve a given control objective. The timed control problem can be modeled as a *two-player game* by *timed game automata*, where the actions taken by the controller (*Player 1*) are *controllable*, whereas the actions taken by the environment (*Player 2*) are *uncontrollable* and maybe considered adversarial. No matter how the environment behaves, the controller will take the right controllable actions at the right time in order to reach its goal [DFLZ14]. UPPAAL-TIGA is an efficient solver for controller synthesis of timed games with reachability or safety control objectives [BCD$^+$07].

One common challenge for model-checking and controller synthesis is the notorious *state-space explosion* problem that the size of the state-space grows

exponentially with the number of concurrent components in the model. Generating and storing the state space consumes a lot of computing resources in terms of CPU time and memory [ZNL16b]. One way to mitigate the state-space explosion and accelerate the exploration process is using parallel and distributed computing on a computer cluster. The linear improvement in the computing power does not overcome the exponential growth problem, but pushes the barrier up to allow analyzing larger or industrial sized models.

1.4 Research Objectives and Contributions

Figure 1 shows our research objectives and contributions (in light yellow boxes) in connection with the existing related work (in gray boxes). As mentioned in the previous section, various techniques derived from model checking have been proposed to solve a wide range of MBSE problems. Algorithms and UPPAAL implementations exist for model checking (MC), time-optimal reachability (TOR), controller synthesis (SYN), and statistical model checking (SMC). Two of them already have distributed versions: distributed statistical model checking (D-SMC) developed in 2012 [BDL+12], and distributed model checking (D-MC) [Beh05] developed in 2005. Unfortunately, the code base of D-MC was not available for this thesis work. As a summary, the objectives and contributions of this PhD work are three folded as follows.

1. **Objective.** We want to find methods for efficiently accelerating time optimal reachability analysis and battling state-space explosion using a computer cluster, for measuring and comparing performance of the developed algorithms, and for possible optimizations to the algorithms.

   Contribution. After surveying distributed model checking techniques, we developed the distributed TOR algorithms for UPPAAL (denoted by

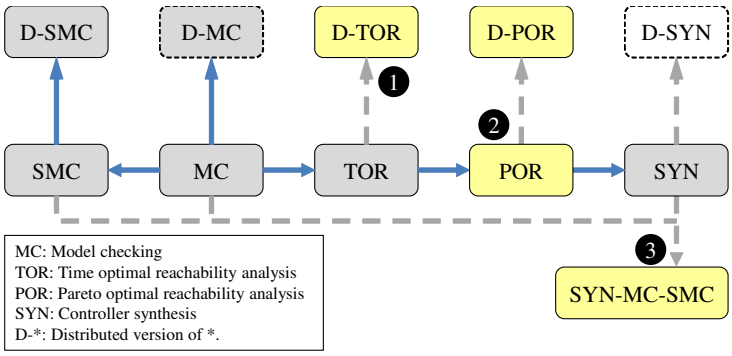

Fig. 1: Overview of Research Objectives and Contributions

tag ❶) in two directions: swarm verification and distributed state-space exploration. The work was published in papers A and B separately. In addition, the product distributed computing module are reusable to recover D-MC and support development of D-POR and D-SYN.

2. **Objective.** We want to solve multi-objective scheduling and planning problems. This entails extending the timed automata formalism properly to support multiple costs, and developing efficient algorithms for multi-objective optimization based on the costs.

   Contribution. We proposed simple priced automata that allow discrete costs on transtions. We developed and implemented Pareto optimal reachability (POR) algorithms (denoted by tag ❷) for UPPAAL. Objectives are functions on cost variables or clock variables in the model. This work is in paper C.

3. **Objective.** We want to verify and evaluate the closed system of the synthesized strategy and the timed game model. This entails extending MC and SMC algorithms to include the strategy as an extra player to supervise the timed game model during (statistical) model checking.

   Contribution. We published two methods to exploit the synthesized strategy in paper D. In the 1^{st} method, we translated the strategy into a controller automaton which is incorporated with the timed game automata into a closed system for MC and SMC. In the 2^{nd} method, we integrate controller synthesis, model checking and statistical model checking into one tool (SYN-MC-SMC) to facilitate (statistical) model check the synthesized strategy (denoted by tag ❸).

1.5 Thesis Structure

The thesis is organized in three parts. Part I is an overview of this PhD work including three chapters. The remaining three chapters in Part I are structured as follows. Chapter 2 describes the fundamental theories of real-time (statistical) model checking, time optimal reachability, controller synthesis, and the related work of distributed model checking. Chapter 3 elaborates our three research contributions listed previously. Chapter 4 makes conclusions and discusses the future work. Part II has detailed explanations on the design and implementation of the distributed time optimal reachability analysis. It also serves as a technical reference for maintenance and further development. Part III includes four papers as follows.

Paper A. *Time Optimal Reachability using Swarm Verification*. Zhengkui Zhang, Brian Nielsen, and Kim G. Larsen. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, pages 1634–1640, ACM 2016.

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.

Contribution: From an overall idea proposed by his supervisors Brian Nielsen and Kim Guldstand Larsen, Zhengkui Zhang co-developed the distributed swarm algorithms. He designed their tool integration and implementation, and implemented them into a distributed swarm version of Uppaal. He performed benchmark experiments and made the analysis. Zhengkui Zhang is the main author of this paper, wrote the complete draft and most revisions.

Paper B. *Distributed Algorithms for Time Optimal Reachability Analysis*. Zhengkui Zhang, Brian Nielsen, and Kim G. Larsen. In Proceedings of Formal Modeling and Analysis of Timed Systems - 14th International Conference FORMATS 2016, volume 9884 of Lecture Notes in Computer Science, pages 157–173, Springer 2016.

Abstract: Time optimal reachability analysis is a novel model based technique for solving scheduling and planning problems. After modeling them as reachability problems using timed automata, a real-time model checker can compute the fastest trace to the goal states which constitutes a time optimal schedule. We propose distributed computing to accelerate time optimal reachability analysis. We develop five distributed state exploration algorithms, implement them in Uppaal enabling it to exploit the compute resources of a dedicated model-checking cluster. We experimentally evaluate the implemented algorithms with four models in terms of their ability to compute near- or proven-optimal solutions, their scalability, time and memory consumption and communication overhead. Our results show that distributed algorithms work much faster than sequential algorithms and have good speedup in general.

Contribution: Inspired by other work in the area, Zhengkui Zhang de-

veloped distributed time optimal algorithms, designed their integration into UPPAAL, and implemented them resulting in the distributed memory version of UPPAAL. He performed benchmark experiments and made the analysis. Zhengkui Zhang is the main author of this paper, wrote the complete draft and most revisions.

Paper C. *Pareto Optimal Reachability Analysis for Simple Priced Timed Automata*. Zhengkui Zhang, Brian Nielsen, Kim G. Larsen, Gilles Nies, Marvin Stenger and Holger Hermanns. Under submission.

Abstract: We propose Pareto optimal reachability analysis to solve multi-objective scheduling and planing problems using real-time model checking techniques. Not only the makespan of a schedule, but also other objectives involving quantities like performance, energy, risk, cost etc, can be optimized simultaneously in balance. We develop the Pareto optimal reachability algorithm for UPPAAL to explore the state-space and compute the goal states on which all objectives will reach a Pareto optimum. After that diagnostic traces are generated from the initial state to the goal states, and Pareto optimal schedules are obtainable from those traces. We demonstrate the usefulness of this new feature using two case studies.

Contribution: Zhengkui Zhang co-developed the Pareto optimal reachability algorithms with his supervisor Kim Guldstand Larsen. He extended the query language, and proposed several extensions to the basic algorithm. He implemented them in UPPAAL. He performed the case studies, including generalizing them with multiple objectives. Zhengkui Zhang is the main author of this paper, wrote most of the body for the draft and most revisions.

Gilles Nies, Marvin Stenger and Prof. Hermanns are the collaborators at Saarland University in Germany. They provided the material for the second case study of the paper. They also reviewed the paper, and gave valuable comments. Gilles Nies was a PhD student at Saarland University. He provided the tool to parse output traces from UPPAAL into visualizable schedules for the second case study. Marvin Stenger was a Master student at Saarland University. He built the UPPAAL-CORA model which was adapted into the UPPAAL model for the second case study.

Paper D. *Verification and Performance Evaluation of Timed Game Strategies*. Alexandre David, Huixing Fang, Kim G. Larsen and Zhengkui Zhang. In Proceedings of Formal Modeling and Analysis of Timed Systems - 12th International Conference, FORMATS 2014, volume 8711 of Lecture Notes in Computer Science, pages 100–114, Springer 2014.

Abstract: Controller synthesis techniques, based on timed games, derive strategies to ensure a given control objective, e.g., time-bounded reachability. Model checking verifies correctness properties of systems. Statistical model

checking can be used to analyze performance aspects of systems, e.g., energy consumption. In this work, we propose to combine these three techniques. In particular, given a strategy synthesized for a timed game and a given control objective, we want to make a deeper examination of the consequences of adopting this strategy. Firstly, we want to apply model checking to the timed game under the synthesized strategy in order to verify additional correctness properties. Secondly, we want to apply statistical model checking to evaluate various performance aspects of the synthesized strategy. For this, the underlying timed game is extended with relevant price and stochastic information. We first explain the principle of translating a strategy produced by UPPAAL-TIGA into a timed automaton, thus enabling the deeper examination. However, our main contribution is a new extension of UPPAAL that automatically synthesizes a strategy of a timed game for a given control objective, then verifies and evaluates this strategy with respect to additional properties. We demonstrate the usefulness of this new branch of UPPAAL using two case-studies.

Contribution: Zhengkui Zhang co-authored this paper with his supervisors Alexandre David and Kim Guldstand Larsen. He contributed to the theory and co-developed the tool CONTROL-SMC for evaluation of strategies using statistical model-checking. This work is now part of UPPAAL-STATEGO. He performed the case studies and analysis. Zhengkui Zhang wrote the most part of this paper and the most revisions.

Huixing Fang was a visiting PhD student for three months from East Normal University in China. He cooperated with Zhengkui Zhang in building models at early phase of this paper during his stay at Aalborg University. Fang also helped in reviewing the draft paper and giving comments.

2 Background

2.1 Real-Time Model Checking

The origin of model checking was driven by concurrent program verification [Cla08]. In the early days, formal verification of a concurrent system typically resorted to *theorem proving* (mostly by means of invariants and deduction) through a set of lemmas and theorems. Theorem proving is tedious, expensive, challenging, and requires skilled mathematical intellectuals, even though computer aided proof assistants are available.

In 1981 Edmund M. Clark and E. Allen Emerson proposed the innovative notion of *model checking* and the model checking algorithm prototype in their pioneering paper [CE81]. Independently J. P. Queille and J. Sifakis published a similar method [QS82] in 1982. Model checking is an automated formal verification technique which is much easier to use than theorem proving. Clarke, Emerson and Sifaksi shared the ACM Turing Award in 1997. Model checking has been successfully used in analyzing circuit designs, communication protocols, concurrent and distributed algorithms. A number of well-known model checking tools were developed ever since such as Spin [Hol03], NuSMV [CCG+02], Embc [MKM15], Java PathFinder [HP00], Uppaal [BDL04], Kronos [BDM+98], mCRL2 [GRU08], Fdr3 [GABR14] etc.

The notion of time is of vital importance for *real-time reactive systems* that react to stimuli received from the evolving environment. The reaction time of these systems must obey the stringent time constraints. However, the timing aspect was not supported in the existing modeling languages by then, until in the early 1990s when several formalisms were proposed to model the real-time behaviors of a reactive system. One of the most popular formalisms is *timed automata* [AD94] introduced by Alur and Dill in 1994, which is capable of modeling instantaneous actions and time elapsing. Real-time model checkers such as Uppaal and Kronos based on the network of timed automata were developed. They have been applied to solve enormous industrial case studies [BGK+02, Feh99, SZHV09, MLR+10, BKLN14]. Uppaal is a user friendly integrated tool environment for modeling, verification, synthesis, simulation and analysis of real-time systems. The modeling language is an extended version of timed automata. The properties (also called queries) are specified by a subset of the timed computational tree logic (TCTL) [ACD93].

2.1.1 Timed Automata

A timed automaton [AD94, HNSY94] is a non-deterministic finite-state machine extended with non-negative real valued clock variables, and annotated with conditions and resets of clocks. All clocks progress synchronously at the same speed of one.

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: $g, g_1, g_2 ::= x \bowtie n \mid x - y \bowtie n \mid g_1 \wedge g_2$, where $x, y \in X$ are clocks, $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

Definition 1. Timed Automaton (TA) *[HNSY94] 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, each of which contains a source location, a guard, an action, a set of clocks to be reset and a target location. For simplicity an edge* $(\ell, g, a, r, \ell') \in E$ *is written as* $\ell \xrightarrow{g,a,r} \ell'$*;*
- $Inv : L \to \mathcal{B}(X)$ *sets an invariant for each location.*

Definition 2. *The semantics of a timed automaton* $\mathcal{A}$ *is a* Timed Transition System (TTS) $S_{\mathcal{A}} = (Q, Q_0, \Sigma, \to)$ *[AILS07], where:*

- $Q = \{(\ell, v) \mid (\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ *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:*
  - $(\ell, v) \xrightarrow{a} (\ell', v')$ *if there is an edge* $(\ell \xrightarrow{g,a,r} \ell') \in E$ *such that* $v \models g$*,* $v' = v[r \mapsto 0]$ *and* $v' \models Inv(\ell')$*;*
  - $(\ell, v) \xrightarrow{d} (\ell, v + d)$ *such that* $v \models Inv(\ell)$ *and* $v + d \models Inv(\ell)$*.*

Definition 3. *A* run *(or* trace*) [JR09]* 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 run with the final state of* q*. .*

Many real-life systems consist of a number of independent components running in parallel and communicating whenever necessary. A real-time system model is a *network of timed automata* (NTA) [AILS07] composed in parallel that can communicate with each other through *synchronization*. Synchronization means when one component raises a request on a particular *channel*, another component accepts the request on the same channel either synchronously or asynchronously. By convention, we use action $a!$ to stand for raising an output on channel a, and $a?$ to stand for accepting an input on channel a. It is worth noting that synchronization is instantaneous – the duration of synchronization action is zero time units.

Let Chan denote a finite set of channels. The set of synchronization actions on Chan is $H = \{c! \mid c \in \texttt{Chan}\} \cup \{c? \mid c \in \texttt{Chan}\}$. Actions $c!$ and $c?$ are said to be *complementary* [AILS07], which need to be performed jointly by both involved timed automata. Let τ represent the *internal communication* that is completed and unobservable similar to that in *Calculus of Communicating Systems* (CCS) [Mil80]. Let O denote a finite set of internal actions that are performed autonomously by individual automaton in an interleaved fashion.

Definition 4. *A* Network of Timed Automata (NTA) *[AILS07] is the parallel composition $\vec{\mathcal{A}} = \mathcal{A}_1 \mid \mathcal{A}_2 \mid \cdots \mid \mathcal{A}_n$ of n timed automata, where n is a positive integer. For each $i \in \{1, \ldots, n\}$, $\mathcal{A}_i = (L_i, \ell_0^i, E_i, Inv_i)$ is timed automaton over a set of clocks X and a set of actions $\Sigma = H \cup O \cup \{\tau\}$.*

Definition 5. *The semantics of a network of timed automata $\vec{\mathcal{A}}$ is a timed transition system $S_{\vec{\mathcal{A}}} = (\vec{Q}, \vec{Q}_0, \Sigma, \rightarrow)$ [AILS07], where:*

- $\vec{Q} = \{(\ell_1, \ell_2, \ldots, \ell_n, v) \mid (\ell_1, \ell_2, \ldots, \ell_n, v) \in L_1 \times L_2 \times \cdots L_n \times \mathbb{R}^X_{\geq 0}$ *and* $v \models \bigwedge_{i \in \{1,\ldots,n\}} Inv_i(\ell_i)\}$ *are states;*
- $\vec{Q}_0 = (\ell_0^1, \ell_0^2, \ldots, \ell_0^n, \vec{0})$ *is the initial state;*
- $\Sigma = H \cup O \cup \{\tau\}$ *is the finite set of actions;*
- $\rightarrow \subseteq \vec{Q} \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times \vec{Q}$ *is the transition relation defined separately for ordinary action $a \in O$, synchronization action τ and delay $d \in \mathbb{R}_{\geq 0}$ as:*
  - $(\ell_1, \ldots, \ell_i, \ldots, \ell_n, v) \xrightarrow{a} (\ell_1, \ldots, \ell_i', \ldots, \ell_n, v')$ *if there is an edge $(\ell_i \xrightarrow{g,a,r} \ell_i') \in E_i$ in the i^{th} component automaton such that $v \models g$, $v' = v[r \mapsto 0]$ and $v' \models Inv_i(\ell_i') \wedge \bigwedge_{k \neq i} Inv_k(\ell_k)\}$;*
  - $(\ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots \ell_n, v) \xrightarrow{\tau} (\ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots \ell_n, v')$ *if $i \neq j$ there are edges $(\ell_i \xrightarrow{g_i,\alpha,r_i} \ell_i') \in E_i$ and $(\ell_j \xrightarrow{g_j,\beta,r_j} \ell_j') \in E_j$ in the* i*-th and* j*-th component automata such that $\alpha, \beta \in H$ are complementary, $v \models g_i \wedge g_j$, $v' = v[r_i \cup r_j \mapsto 0]$ and $v' \models Inv_i(\ell_i') \wedge Inv_j(\ell_j') \wedge \bigwedge_{k \neq i,j} Inv_k(\ell_k)\}$;*
  - $(\ell_1, \ldots, \ell_n, v) \xrightarrow{d} (\ell_1, \ldots, \ell_n, v + d)$ *such that $v + d' \models \bigwedge_{i \in \{1,\ldots,n\}} Inv_i(\ell_i)$ for each real number $d' \in [0, d]$.*

The semantics of NTA is compatible with that of TA and the definitions of runs also apply to NTA, because we can think of a state in NTA as a state $(\vec{\ell}, v)$, where $\vec{\ell}$ denotes the vector of locations in NTA, and v denotes the vector of clock evaluations in NTA. The UPPAAL modeling language extends timed TA with additional features [BDL04]: data types (templates, constants, bounded integers, arrays, etc), channels (binary or urgent synchronization, broadcast), location modifiers (urgent, committed), user functions, and stop-watches [CL00] for modeling task preemption.

2.1.2 MC Query

TCTL has a two-stage syntax where the language is classified into state and path formulae. *State formulae* describe individual states, whereas *path formulae* quantify over runs of the model [BK08]. UPPAAL uses a simplified version of TCTL (let us call it TCTL$^\sharp$) to specify a property, where the nesting of formulae is not allowed

Definition 6. TCTL$^\sharp$ *[BDL04] state formulae* Φ *over the set of atomic propositions AP (only on locations, clocks and variables) and a keyword deadlock are formed according to the following grammar:*

$$\Phi ::= E\Diamond\varphi \mid A\Box\varphi \mid E\Box\varphi \mid A\Diamond\varphi \mid \varphi_1 \rightsquigarrow \varphi_2$$

$$\varphi ::= true \mid p \mid deadlock \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi$$

where $p \in AP$*, deadlock refers to the deadlock state. Quantifiers A pronounced as "for all runs", E as "for some run",* $\Box$ *as "always", and* $\Diamond$ *as "future".*

Definition 7. *Let* $Exec_{\mathcal{A}}(q)$ *denote the set of runs generated from a state q in the timed transition system* $S_{\mathcal{A}}$*. The satisfaction relation* $\models$ *is defined for the state formulate of* TCTL$^\sharp$ *on a state* $q \in S_{\mathcal{A}}$ *by:*

$q \models true$	*always satisfied;*
$q \models p$	*iff* $p \in q$*;*
$q \models deadlock$	*iff no action, synchronization or delay transition can be taken;*
$q \models \varphi_1 \wedge \varphi_2$	*iff* $(q \models \varphi_1)$ *and* $(q \models \varphi_2)$*;*
$q \models \neg\varphi$	*iff not* $q \models \varphi$*.*
$q \models E\Diamond\varphi$	*iff* $\exists\, \pi \in Exec_{\mathcal{A}}(q)$ *s.t.* $\pi[i] \models \varphi$ *for some* $i \geq 0$*;*
$q \models A\Box\varphi$	*iff* $\forall\, \pi \in Exec_{\mathcal{A}}(q)$ *s.t.* $\pi[i] \models \varphi$ *for all* $i \geq 0$*;*
$q \models E\Box\varphi$	*iff* $\exists\, \pi \in Exec_{\mathcal{A}}(q)$ *s.t.* $\pi[i] \models \varphi$ *for all* $i \geq 0$*;*
$q \models A\Diamond\varphi$	*iff* $\forall\, \pi \in Exec_{\mathcal{A}}(q)$ *s.t.* $\pi[i] \models \varphi$ *for some* $i \geq 0$*;*
$q \models \varphi_1 \rightsquigarrow \varphi_2$	*equivalent to* $q \models A\Box(\varphi_1 \rightarrow A\Diamond\varphi_2)$*, iff* $\forall\, \pi \in Exec_{\mathcal{A}}(q)$ *if* $\pi[i] \models \varphi_1$*, then* $\forall\, \pi' \in Exec_{\mathcal{A}}(\pi[i])$ *s.t.* $\pi'[j] \models \varphi_2$ *for all* $i \geq 0$ *and for some* $j > i$*.*

The satisfaction relation of a TCTL$^\sharp$ property Φ on the complete timed transition system $S_{\mathcal{A}}$ initiated from the initial state (ℓ_0, v_0) is given by:

$$S_{\mathcal{A}} \models \Phi \text{ iff } (\ell_0, v_0) \models \Phi$$

Figure 2 demonstrates the satisfaction relations of the five state formulae of TCTL$^\sharp$, which can be grouped into three categories as: reachability, safety and liveness [BDL04].

- **Reachability** is used for sanity check of a model.

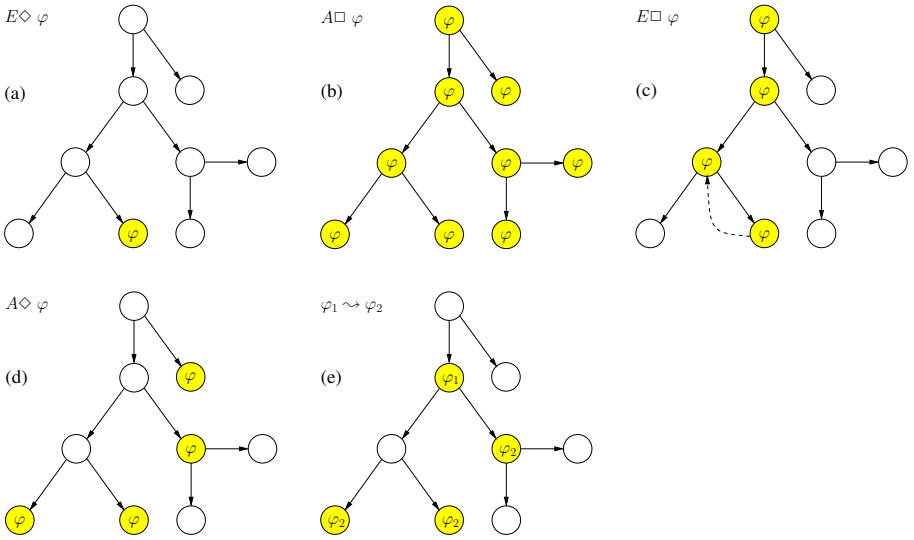

Fig. 2: Five Basic UPPAAL Verification Queries [BDL04]

$E\Diamond\varphi$ There exists some run on which φ holds at some state (Fig. 2 (a)).

- **Safety** express that "something bad will never happen".

$A\Box\varphi$ For all runs and for all states on those traces φ holds (Fig. 2 (b)).
$E\Box\varphi$ There exists a run on which φ always holds (Fig. 2 (c)).

- **Liveness** express that "something good should happen eventually".

$A\Diamond\varphi$ For all runs φ eventually holds (Fig. 2 (d)).
$\varphi_1 \rightsquigarrow \varphi_2$ Whenever φ_1 holds for a state, then φ_2 will always hold eventually for all runs starting from that state (Fig. 2 (e)).

2.1.3 MC Reachability Algorithms

The semantics $S_{\mathcal{A}}$ of TA will result in an infinite transition system of continuous time, and therefore model checking on $S_{\mathcal{A}}$ is undecidable. Alur and Dill proposed using *region*s [AD94] to build a finite state abstraction of $S_{\mathcal{A}}$. Regions partition all clock valuations v associated with the same location $\ell \in L$ into finite many equivalence classes such that two valuations from the same equivalence class will not create any significant difference in the behavior of the system [AILS07]. A discrete transition system *region graph* is built with finitely many regions. It is timed abstract bisimilar to $S_{\mathcal{A}}$, and model checking on the region graph is decidable.

However, the number of regions in the region graph is exponential in the number of clocks and in the maximal constants appearing in the guards [AD94].

A more efficient finite state abstraction called *zone* was proposed by Yi and Petterson in [YPD94]. Zone $Z = \{v \mid v \models g_z, g_z \in \mathcal{B}(X)\}$ is a convex union of clock valuations (or regions) and allows a coarser and more compact representation of the state-space [AILS07]. Zones are efficiently represented and stored in memory as *difference bound matrices* (DBM) [BY03]. A finite symbolic transition system named *symbolic reachability graph* is built on zones. UPPAAL works by exploring the symbolic reachability graph, where the nodes are *symbolic states* (ℓ, Z).

Definition 8. *The symbolic reachability graph of a timed automaton $\mathcal{A}$ is a label transition system $T_{\mathcal{A}} = (S, S_0, \Sigma \cup \{\lambda\}, \rightsquigarrow)$ [AILS07], where:*

- $S = \{(\ell, Z) \mid (\ell, Z) \in L \times \mathcal{B}(X)\}$ *are symbolic states;*
- $S_0 = (\ell_0, Z_0)$ *is the initial state, where $Z_0 = Inv(\ell_0)$ is the initial zone;*
- Σ *is the finite set of actions in $\mathcal{A}$, λ denotes time elapsing;*
- $\rightsquigarrow \subseteq S \times (\Sigma \cup \{\lambda\}) \times S$ *is the transition relation on symbolic states defined separately for action and delay transitions as follows:*
  - $(\ell, Z) \overset{a}{\rightsquigarrow} (\ell', (Z \wedge g)[r] \wedge Inv(l'))$ *if* $(\ell \xrightarrow{g,a,r} \ell') \in E$;
  - $(\ell, Z) \overset{\lambda}{\rightsquigarrow} (\ell, Z^{\uparrow} \wedge Inv(l))$.

  where $Z[r] = \{v[r \mapsto 0] \mid v \in Z\}$ is reset function, $Z^{\uparrow} = \{v + d \mid v \in Z \wedge d \in \mathbb{R}_{\geq 0}\}$ is future function.

Algorithm 1 shows the reachability algorithm [BY03] for checking if the goal states satisfying the reachability property `Goal` are reachable from the initial state. WAITING and PASSED keep unexplored and explored symbolic states respectively; and WAITING has initially the initial symbolic state (ℓ_0, Z_0). Inside procedure `Main`, as long as WAITING is not empty, an unexplored state is popped from WAITING. If the state is a goal state, the loop is quited and

Algorithm 1: MC Reachability

```
WAITING ⟵ {(ℓ0, Z0)}, PASSED ⟵ ∅
Procedure Main()
1   while WAITING ≠ ∅ do
2       select (ℓ, Z) from WAITING
3       if (ℓ, Z) ⊨ Goal then return TRUE
4       else if ¬∃(ℓ, H) ∈ PASSED s.t. Z ⊆ H then
5           add (ℓ, Z) to PASSED
6           forall the (ℓ', Z') such that (ℓ, Z) ⇝ (ℓ', Z') do
7               add (ℓ', Z') to WAITING
8   return FALSE
```

message TRUE is reported. Otherwise, the state is subject to symbolic state inclusion checking at line 4. The state is discarded if a previously explored state with the same location has an equal or larger zone than that of the current state. If the state passes inclusion checking, it is added to PASSED as already explored, and then its successor states are generated and added to WAITING. If all states in WAITING are explored, but the goal states are not reached, the reachability query `Goal` is not satisfied, and message FALSE is reported.

The reachability algorithm is implemented in UPPAAL as the pipeline architecture shown in Figure 3. The pipeline connects of a series of filter, buffer and pump components [BDLY03]. Filter receives input data by a `put` method, processes the data, and automatically sends the processed data to the next connecting component. Buffer awaits input data by `put`, stores the data, and offers the data to other components by a `get` method. Pump continuously pumps data from a buffer and sends it to a sequence of connected filters. The component filters inside this reachability pipeline are as follows.

1. **Delay** computers the delay of state bounded by its invariants.
2. **Extrapolation** widens a state-set according to the maximum constants that clocks are compared to in the model: for a given timed automaton, two states only differing with respect to the value of clocks exceeding the corresponding maximum constant are indistinguishable with respect to (location) reachability [Lar10]. The extrapolated state space is finite, which is crucial to guarantee termination of UPPAAL.
3. **Active Clock Reduction** only saves constraints on active clocks. Clock x is *inactive* at state S if on all runs initiated from S, x is always reset before being tested.
4. **Property** evaluates a state against the reachability property. It stops the pump (marked as a crossed circle) if the property is satisfied.
5. **Transition** computes all possible transitions from the current state.
6. **Successor** computes a symbolic state fired by each transition.

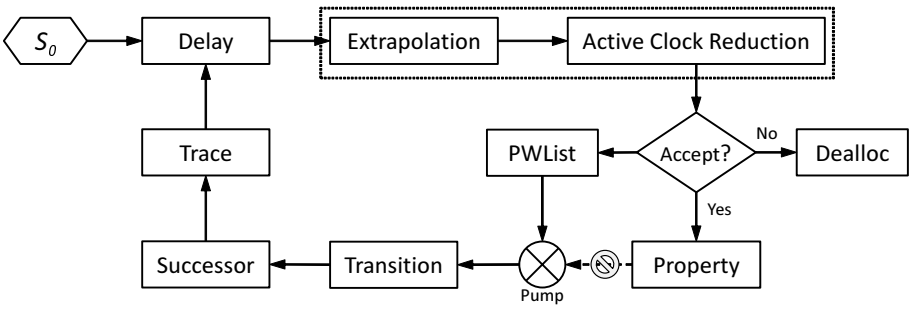

Fig. 3: Reachability Pipeline of UPPAAL [BDLY03]

7. **Trace** stores action transitions between states into a list container.

The initial state S_0 is inserted into the pipeline at the delay stage initially. A state is conditionally inserted into the PWList[1] buffer if it passes the inclusion checking (Accept = Yes), otherwise it is deallocated from memory. The pump drains an unexplored waiting state from the PWList buffer each time and inserts it into the pipeline.

The reachability algorithm is the basis for building safety and liveness algorithms as well as the time optimal reachability algorithm in Section 2.2. We will not continue to survey the algorithms for safety and liveness properties. Readers can refer [BK08, JR09] for detailed explanations.

2.2 Time Optimal Reachability Analysis

Network of timed automata is an elegant formalism for modeling scheduling problems with concurrent components in a static environment. It allows modeling the real-time behaviors, constraints and interactions between parallel components in a natural way. It is worth noticing that we assume the environment is static, such that the execution period of a task is fixed, and the scheduler has full controllability over the tasks. This is a one-player game or a open loop control. If there is uncertainty (or even stochasticity) in the model, i.e., task execution time is not fixed or is interruptible by some events, a dynamic schedule with replanning capability is needed.

Around the year 2000, researchers proposed that the time optimal scheduling problem of real-time systems can be reduced to the time optimal reachability (TOR) analysis [NTY00] on timed automata. The diagnostic trace to a goal state offered by real-time module checkers such as Uppaal or Kronos can be interpreted as a feasible schedule because the trace carries actions of a model as well as timing information of these actions to the goal. Because model checkers typically implement efficient 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 update topological and timing constraints in the model without being forced to change the underlying search algorithms [Feh99]. In [AAM06], Maler et al. deemed TOR as one orthogonal direction extending from verification, evolving from qualitative Yes or No answers to quantitative evaluation of model behaviors.

A generalization of time optimal scheduling is the cost optimal scheduling. Behrmann et al. proposed minimum cost reachability analysis on *priced timed automata* in [BFH$^+$01].

Definition 9. *A* Priced Timed Automaton (PTA) *extends TA as a 7-tuple* $\mathcal{P} = (\mathcal{A}, P)$ *[BFH$^+$01], where:* $\mathcal{A}$ *is timed automaton,* $P : (L \cup E) \to \mathbb{N}$ *assigns a discrete price to each edge and a price rate (cost per time unit) to each location.*

[1] PWList is implemented as an efficient uniform passed-waiting list [BDLY03].

The total cost along a run is calculated by accumulating the costs according to either discrete price annotations on transitions or price rates integrated over delays on locations. The run with the minimum cost was proved computable, and the tool named Uppaal-cora was developed and applied in case studies [LKZ16, BGH+16, BBHM05, BLR04]. The optimal reachability analysis on the more general *multi-priced timed automata* (MPTA) was also proved decidable [LR08]. However, the model checking problem on MPTA was proven undecidable [BBR04].

2.2.1 TOR Reachability Algorithms

Branch and Bound (B&B) is an algorithmic paradigm widely applied in optimization and planning algorithms on graphs. The purpose of using B&B is to find an optimal solution without necessarily having to enumerate 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 not to lead to an optimal solution [Feh00]. Behrmann *et al*. presented a branch-and-bound minimum cost reachability algorithm on PTA [BF01]. This algorithm was the basis for formulating the time optimal reachability algorithm.

Definition 10. *The* span *of a finite run* $\rho = q_0 \xrightarrow{d_1} q_0' \xrightarrow{a_1} q_1 \xrightarrow{d_2} q_1' \xrightarrow{a_2} \cdots \xrightarrow{d_n} q_{n-1}' \xrightarrow{a_n} q_n$ *is defined as the finite sum* $\Sigma_{i=1}^{n} d_i$ *[NTY00].*

Algorithm 2: Time Optimal Reachability [ZNL16b]

```
WAITING ⟵ {(ℓ0, Z0)}, PASSED ⟵ ∅, COST ⟵ ∞
Procedure Main()
1   while WAITING ≠ ∅ do
2       select (ℓ, Z) from WAITING
3       if (ℓ, Z) ⊨ Goal then
4           if MinCost(ℓ, Z) < COST then
5               COST ⟵ MinCost(ℓ, Z)
6       else if ¬∃(ℓ, H) ∈ PASSED s.t. Z ⊆ H and MinCost(ℓ, Z) < COST
        then
7           add (ℓ, Z) to PASSED
8           forall the (ℓ', Z') such that (ℓ, Z) ⇝ (ℓ', Z') do
9               add (ℓ', Z') to WAITING
10  return COST
```

UPPAAL keeps track of the span of a run in NTA by including an implicit observer clock ψ in addition to the original set of clocks X of the model. Clock ψ advances as the global elapsing time and remains unaffected from resets or guards or invariants in the model. Thus the zone Z is now over $X \cup \{\psi\}$.

Definition 11. *The* cost function *on a symbolic state* (ℓ, Z) *is defined as* $\texttt{MinCost}(\ell, Z) = \texttt{inf}\{v(\psi) \mid v \in Z\}$. *It is the infimum evaluation of the clock* ψ *in the symbolic state.*

Algorithm 2 shows the sequential TOR algorithm [ZNL16b] that computes the minimum span to reach the goal states satisfying the reachability property `Goal`. It resembles Algorithm 1 and has four extensions. First, a variable COST maintains the current best result that is infinity initially. Second, the cost function `MinCost` is applied on every symbolic state to evaluate the minimum span along a symbolic run reaching that symbolic state. Third, if the state is a goal state, COST is updated at line 4 rather than quiting the loop and reporting reachable in Algorithm 1. This implies a near optimal schedule to the goal is found. Fourth, if the state is not a goal state, it is subject to both inclusion checking and B&B elimination rule at line 6. Consequently, this algorithm will explore the entire state-space and reports either the minimum span found or infinity if `Goal` is not satisfied.

2.3 Controller Synthesis

Scheduling by time optimal reachability analysis in Section 2.2 is limited to a static environment. In other words, the scheduler is assumed to have full controllability over the components in the model, so that all components will function faithfully according to the pre-planed static schedule. In more general cases however, components have autonomy and uncertainty in its behaviors, and consequently the pre-planned static schedule will not work.

Scheduling a TA model with uncertainty can be formulated as a *two-player timed game* [AAM06] from the game theory perspective, because the evolution of the model depends on the actions of two players – *controller* and *environment*. The controller decides whether or not to take a controllable action and when according to the current situation. The environment is the generic name for all sources of uncontrollable external events or non-deterministic factors. Computing the control strategy for a timed game is normally referred to as *controller synthesis* (SYN).

Around 1990s controller synthesis of reactive systems has became a topic of growing interest. Pioneering researches on controller synthesis of discrete event systems were performed [RW87, RW89, McN93, Tho95]. The work inspired the research on controller synthesis of real-time reactive systems. In mid-1990s the formulation of timed games based on timed automata and the controller synthesis algorithms were proposed [MPS95, AMPS98]. Later in

2005, Cassez et al. published an efficient, fully on-the-fly algorithm [CDF+05] for solving timed games with perfect information by forward search of goal states and backward fix-point computation of winning states. The algorithm was implemented in the branch of UPPAAL called UPPAAL-TIGA [BCD+07]. Ehlers et al. proposed another concept to solve timed game by abstraction refinement of the game [EMP10] and implemented the algorithm in a tool called SYNTHIA [PEM11] around 2010.

In [AM99], time optimal controller synthesis for timed games was proven decidable. In [BCFL04], optimal cost for winning cost optimal reachability game on priced timed game automata was proven computable under some non-Zenoness assumption. Due to limited sensors, controller may have imperfect (or partial) information about the state of the environment. Timed control under partial observability is in general undecidable, but fixing the resources of the controller (i.e., maximum number of clocks or allowed constants in guards) regains decidability [BDMP03]. Due to economic reasons, we need to minimize the number of sensors installed. This inspires the topic of timed game with cost minimal observability. Bulychev et al. proposed a modeling strategy and an algorithm to solve this problem [BCD+12].

2.3.1 Timed Game

A timed game automaton is an extension of timed automaton whose actions are partitioned into *controllable* actions for the controller and *uncontrollable* actions for the environment. Besides discrete actions, each player can decide to wait in the current location. As soon as one player decides to play one of his available actions, time will stop elapsing and the action will be taken.

Definition 12. Timed Game Automaton (TGA) is a 7-tuple $\mathcal{G} = (L, \ell_0, X, \Sigma_c, \Sigma_u, E, Inv)$ [MPS95], *where:*

- Σ_c *is the finite set of controllable actions;*
- Σ_u *is the finite set of uncontrollable actions;*
- Σ_c *and* Σ_u *are disjoint;*
- $(L, \ell_0, X, \Sigma_c \cup \Sigma_u, E, Inv)$ *is a timed automaton as in Definition 1.*

Let $S_\mathcal{G}$ denote the timed transition system of $\mathcal{G}$. The semantics of $S_\mathcal{G}$ complies with that of $S_\mathcal{A}$. For a run ρ in $S_\mathcal{G}$, its action transitions are $a \in \Sigma_c \cup \Sigma_u$. Let $Exec_\mathcal{G}$ denote the set of runs of $\mathcal{G}$, and $Exec^f_\mathcal{G}$ denote the set of finite runs. The length of a finite run $\rho \in Exec^f_\mathcal{G}$ is the number of delay and action transitions on ρ. Let $Exec^m_\mathcal{G}$ denote the set of maximal runs that cannot be extended. It includes the following three kinds of runs [JR09]:

- Runs having infinitely many action transitions;
- Finite runs ending with an infinite delay;
- Finite runs from which neither delay nor action transition is possible.

2.3.2 SYN Query

Control objectives are described by the query language of UPPAAL-TIGA, which is a subset of TCTL$^\sharp$ as:

$$\Phi ::= A\Diamond\varphi \mid A\Box\neg\varphi$$

Controller synthesis is: given a timed game model $\mathcal{G}$ and a control objective Φ, synthesize a strategy C such that $\mathcal{G}$ supervised by C satisfies Φ regardless how the environment behaves ($C(\mathcal{G}) \models \Phi$). Assume φ corresponds to a set of states $K \subseteq L \times \mathbb{R}^X_{\geq 0}$. The formulae Φ define two kinds of control problems [JR09] supported by UPPAAL-TIGA.

- **Reachability control problem**: $C(\mathcal{G}) \models A\Diamond\varphi$ meaning that we want $\mathcal{G}$ supervised by C to reach K eventually.
- **Safety control problem**: $C(\mathcal{G}) \models A\Box\neg\varphi$ meaning that we want $\mathcal{G}$ supervised by C to avoid K constantly.

Besides, there are other control objectives that are outside the scope of this thesis, such as repeated reachability, Büchi or co-Büchi. We define a run $\rho \in Exec_{\mathcal{G}}$ as *winning* [JR09] in terms of the aforementioned control objectives.

- **Reachability Game.** The run ρ is winning if $\exists k \geq 0$, $s.t.$ $(\ell_k, v_k) \in K$.
- **Safety Game.** The run ρ is winning if $\forall k \geq 0$, $s.t.$ $(\ell_k, v_k) \notin K$.

2.3.3 Strategy

The notion of strategy is the central concept in game theory. During the evolution of a game, it constantly tells the controller how to play in order the win the game, depending on the history of the game.

Definition 13. *A* strategy *for controller in timed game $\mathcal{G}$ is a mapping $C : Exec^f_{\mathcal{G}} \to \Sigma_c \cup \{\lambda\}$, where λ denotes the delay transition. Then C satisfies the following conditions [JR09]: given a finite run ρ ending in state $q = last(\rho) = (\ell, v)$,*

- *If $C(\rho) = a \in \Sigma_c$, then there must exist an action transition $q \xrightarrow{a} q'$ in $S_{\mathcal{G}}$;*
- *If $C(\rho) = \lambda$, then there must exist a delay transition $q \xrightarrow{d} q'$ in $S_{\mathcal{G}}$.*

We denote the set of strategies for controller in $\mathcal{G}$ as $Strat(\mathcal{G})$. In some special situations, a strategy is independent of the history and only depends on the current state of the game. That is $\forall \rho, \rho' \in Exec^f_{\mathcal{G}}$, $last(\rho) = last(\rho')$, implies $C(\rho) = C(\rho')$, it is called a *positional* or *memoryless* strategy [Tho95]. The strategies for reachability and safety games, as the ones handled by UPPAAL-TIGA, are memoryless.

If a timed game $\mathcal{G}$ is supervised by a strategy $C \in strat(\mathcal{G})$, what would happen? The behaviors of the game is restricted, and all possible runs $Exec_{\mathcal{G}}$ of the game is narrowed down to a subset which complies with that strategy. The subset is named *outcome*, which defines the result of game.

Definition 14. *Let $q = (\ell, v) \in S_{\mathcal{G}}$ be a state, and $C \in Strat(\mathcal{G})$ be a strategy for timed game $\mathcal{G}$. The* outcome *of C from q in $S_{\mathcal{G}}$, denoted by $Out(q, C)$, is defined inductively as follows [JR09]:*

- $q \in Out(q, C)$;
- *Any finite run in $\mathcal{G}$ of the form $\rho' = \rho \xrightarrow{\sigma} q' \in Out(q, C)$ iff $\rho \in Out(q, C)$ and one of the following three conditions holds:*
  - $\sigma \in \Sigma_u$;
  - $\sigma \in \Sigma_c$ *and* $\sigma = C(\rho)$;
  - $\sigma \in \mathbb{R}_{>0}$ *and* $\forall 0 \leq d < \sigma$ *such that* $C(\rho \xrightarrow{d} q') = \lambda$.
- *An infinite run $\rho \in Out(q, C)$ if all the finite prefixes of ρ are in $Out(q, C)$.*

Let $Out^m(q, C) = Out(q, C) \cap Exec^m_{\mathcal{G}}$ denote the *maximal outcome* of applying strategy C on state q. Let $WinRun(q, \mathcal{G})$ denote *winning runs* in $\mathcal{G}$ from q. The winning strategies and winning states are defined as follows.

Definition 15. *Strategy C is* winning *from state q for the control objective Φ, if $Out^m(q, C) \subseteq WinRun(q, \mathcal{G})$, where $WinRun(q, \mathcal{G}) \models \Phi$. A state q is winning, if there exists a winning strategy from q [JR09].*

Finally, strategy C is said to be winning in $\mathcal{G}$ for a control objective Φ if it is winning from the initial state. We denote the set of winning strategies as $WinStrat(\mathcal{G}, \Phi)$, and the set of winning states as $WinState(\mathcal{G}, \Phi)$.

2.4 Statistical Model Checking

Real complex *cyber-physical systems* (CPS) often have stochastic behaviors due to equipment failure, unpredictable communication delay, environmental uncertainty etc. Timed automata is not powerful enough [DLL+15] to model these. Separately mathmatical frameworks for modeling stochastic behaviors were proposed [Nor98, Bel57] such as discrete-time Markov chain (DTMC), continuous time Markov chain (CTMC), Markov decision process (MDP), or stochastic extensions of timed automata etc. *Probabilistic model checkers* like Prism [KNP11] have been developed to support probability estimation or hypothesis testing problems. *Probabilistic model checking* (PMC) aims to provide exact probability values but suffers from complex and expensive calculations as well as large memory requirements, making it infeasible for many realistic large models.

Statistical model checking (SMC) [SVA04, You05] is a compromise between Monte Carlo simulation and probabilistic model checking. The core idea of SMC is to monitor some simulated sample runs of the probabilistic system model, then apply statistical algorithms to do probability estimation or hypothesis testing at a given level of confidence. The simulation-based

feature of SMC avoids exhaustive exploration of the state-space, imposing far less memory requirements (no need to store the state-space). SMC scales better than PMC and allows more expressiveness of the models.

Uppaal-smc [DLL+11a, DLL+11b] is an extension of Uppaal dedicated for statistical model checking on models of *stochastic price timed automata* (STPTA). It has been used to successfully solve many case studies listed at this address[2] and more [CFL+08, DDL+12, DDL+13]. Now Uppaal-smc supports the analysis of *stochastic hybrid automata* (SHA), which is suitable for expressing stochastic and non-linear dynamic features of CPS. SHA extends STPTA by allowing the clock rates to depend not only on values of discrete variables but also on the values of other clocks, effectively amounting to *ordinary differential equations* (ODE) [DDL+12].

2.4.1 Stochastic Semantics

A stochastic price timed automaton (STPTA) [DLL+11a] extends PTA (in Definition 9) with three modifications. First, partitioning the set of actions into disjoint input actions (Σ_i) and output actions (Σ_o) such that $\Sigma = \Sigma_i \uplus \Sigma_o$. Second, assigning to each location a rate vector for all clocks denoted by $R : L \to \mathbb{N}^X$. For $v \in \mathbb{R}^X_{\geq 0}$ and $d \in \mathbb{R}_{\geq 0}$, we write $v + R(\ell) \cdot d$ to denote the clock valuation defined by $(v + R(\ell) \cdot d)(x) = v(x) + R(\ell)(x) \cdot d$ for any $x \in X$. Third, refining the non-deterministic choices of output and delay inside PTA with the following two probabilistic interpretations:

- γ_q is the *output probability function* for STPTA to choose among multiple enabled transitions at the state q.
- μ_q is the *delay density function* for STPTA to choose a delay at the state q. It is given as either a uniform distribution in case of a time-bounded delay or an exponential distribution with a user specified rate in case of an unbounded delay.

Stochastic semantics on a network of STPTA (NSTPTA) [DLL+11a] is the mathematical foundation of Uppaal-smc. Consider a closed NSTPTA $\mathbf{A} = (\mathcal{P}_1 | \ldots | \mathcal{P}_n)$. For a concrete global state $q = (q_1, \ldots, q_n)$ and $a_1 a_2 \ldots a_k \in \Sigma^*$ we denote by $\pi(q, a_1 a_2 \ldots a_k)$ the set of all maximal runs from q with a prefix $t_1 a_1 t_2 a_2 \ldots t_k a_k$ for some $t_1, \ldots, t_2 \in \mathbb{R}_{\geq 0}$, that is, runs where the i^{th} action a_i has been output by the component $\mathcal{P}_{c(a_i)}$. The probability for getting such sets of runs is:

$$\mathbb{P}_{\mathbf{A}}(\pi(q, a_1 a_2 \ldots a_k)) = \int_{t \geq 0} \mu^c_q(t) \cdot \Big(\prod_{j \neq c} \int_{\tau > t} \mu^j_q(\tau) d\tau\Big) \cdot \gamma^c_{q^t}(a_1) \cdot \mathbb{P}_{\mathbf{A}}\big(\pi((q^t)^{a_1}, a_2 \ldots a_n)\big) dt$$

[2] http://people.cs.aau.dk/~adavid/smc/cases.html

where $c = c(a_i)$ is the *index* of component taking a_1, μ_q^c is the delay density function for component c to choose a delay t_i at q, and $\gamma_{q^t}^c$ is the output probability function for component c to choose an action a_i after q is delayed by t. The above nested integral reflects that the stochastic semantics of the network is defined based on race among components. All components are independent in giving their delays which are decided by the given delay density functions. The player component who offers the minimum delay is the winner of the race, and takes the turn to make a transition and (probabilistically) choosing the action to output [DFLZ14]. UPPAAL-SMC engine generates simulated runs according to this stochastic semantics.

2.4.2 SMC Query

The query language of UPPAAL-SMC [DLL+11a, DLL+15] is a conservative weighted extension of TCTL$^\sharp$ with a time- or cost-bound as:

$$\Phi ::= \Diamond_{x \leq C} \varphi \mid \Box_{x \leq C} \varphi$$

where x is an observer clock, C is a bound, and φ is a combination of propositions as in Definition 6.

Let $\mathbb{P}_M(\Phi)$ denote the probability that a random run of NSTPTA model M satisfies Φ. UPPAAL-SMC can solve the following three kinds of problems [DLL+15]:

1. **Probability Estimation.** Approximate $\mathbb{P}_M(\Phi)$, with the margin of error ϵ at the level of significance α.

2. **Hypothesis Testing.** Decide whether $\mathbb{P}_M(\Phi) \geq \theta$, given a threshold $\theta \in [0,1]$. The upper (lower) probability deviations δ_0 (δ_1) define the indifference region around θ. Type I (II) errors of test are α (β).

3. **Probability Comparison.** Decide whether $\mathbb{P}_M(\Phi_1) \geq \mathbb{P}_M(\Phi_2)$. It is transfered to the hypothesis testing of $H_0 : \frac{\mathbb{P}_M(\Phi_1)}{\mathbb{P}_M(\Phi_2)} \geq u_1$ and $H_1 : \frac{\mathbb{P}_M(\Phi_1)}{\mathbb{P}_M(\Phi_2)} \leq u_0$, where u_0 and u_1 defines the indifference region.

2.5 Distributed (Statistical) Model Checking

Model checking, time optimal reachability analysis and controller synthesis all work by exhaustively exploring the state-space. Because the size of the state-space may grow exponentially with the number of interactive components in a model, the state-space explosion problem may appear if the model is large. The state-space may grow so enormous that a single PC is incapable of storing or exploring it. The available memory on that PC may be depleted, or the single CPU may take a long time to explore and give an answer. A

number of optimization techniques were developed to reduce the size of the state-space, such as abstraction [CGL94], state compression [GdV99], binary decision diagram [Bry86] etc. Other enlightened methods were to restrict the state-space exploration to a much smaller subset such as partial order reduction [Pel98], symmetric reduction [Sis04], branch and bound [BFH+01], guided search [HLP00], sweep-line method [JKM12] etc. Both directions of methods aim to squeeze the explored state-space locally.

An approach to battle the state-space explosion is parallel and distributed model checking. The earliest monumental distributed model checker is the parallel Murφ verifier [SD97] proposed in 1997. Its design delineated the cornerstone upon which other parallel or distributed model checkers were built thereafter such as distributed UPPAAL [BHV00, Beh05], DIVINE [BBvR10], LTSMIN [BvdPW10], parallel FDR3 [GABR16], CADP [GMS13] etc. These tools work by partitioning the state-space and distributing the parts among distributed CPUs and memory resources by message passing. Given sufficient memory and time, this paradigm can verify very large models. Meanwhile enormous research efforts have been made to improve the state-space generation [BLvdPW11], the partition algorithm [NC97], the state storage [ESB+11], the communication and control mechanism [FGI12], as well as many other technical issues [VBBB09]. Since 2006 multi-core CPUs became pervasive inside PC, HPC and embedded markets, DIVINE, LTSMIN and FDR3 exploit multi-core shared memory technique to achieve even better performance on the modern hardware architecture. DIVINE also made fruitful attempts to accelerate model checking using GPUs from 2009 [BBBC12].

Other possible paradigms for parallel model checking were also proposed. Holzmann *et al.* introduced swarm verification [HJG08]. A large number of SPIN instances configured with well-tuned randomized search strategies and bitstate hashing run in parallel. Even though the state-space generation may be lossy and bitstate hashing may lead to false positive results, swarm verification was shown to give high quality results fast under restricted time and memory. Rasmussen *et al.* proposed agent-based search [RBL07]. Various agents with diverse search patterns put tasks to (or get tasks from) a pool, with tasks being sub-paths that lead to promising parts of the state-space.

Statistical model checking (SMC) is not memory intensive, because the simulation based feature does not need to store the state-space. But SMC can be computation intensive due to the demand of extremely high confidence. This is called *confidence explosion*, because it requires the SMC engine to generate a large number of simulation runs, each of which may itself be extremely time consuming. Paper [BDL+12] proposed the distributed SMC algorithm based on task parallelism because the simulation runs are independent and trivially parallelizable, but care must be taken to make hypothesis testing results correct. The master/slave architecture is chosen for implementing distributed UPPAAL-SMC: a group of slave processes register their ability to

generate simulations with a single master process that is used to collect those simulations and perform statistical test.

2.6 Concluding Remarks

This chapter provided an overview of a series of related timed automaton based engineering techniques: real-time model checking (MC), statistical model checking (SMC), time optimal reachability analysis (TOR), controller synthesis (SYN) and the distributed technologies. The mathematical foundations are defined. The up-to-date related work is surveyed. The dependencies between the background knowledge and our papers are: papers A to C rely on MC and TOR; paper D relies on SYN, MC and SMC.

3 Contributions

3.1 Distributed Time Optimal Reachability

In order to accelerate time optimal reachability (TOR) analysis and mitigate the state-space explosion problem, we proposed separately in papers A and B two different distributed computing paradigms: swarm verification [ZNL16b] and distributed memory state-space exploration [ZNL16a].

Our contributions are two folded. Firstly, we developed four swarm algorithms and five distributed algorithms for UPPAAL. An important feature of these algorithms is that we allow all worker processes to exchange the better results they have found. This shared knowledge allows each worker to have efficient branch and bound pruning as well as guided exploration in the state-space, therefore may speedup the global analysis progress [ZNL16b]. Secondly, we proposed four performance metrics to evaluate these algorithms comprehensively as follows [ZNL16b, ZNL16a].

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 and communication overhead[3] of algorithms. Smaller memory consumption improves scalability by allowing bigger state-space. Smaller communication overhead improves computing speed.

3.1.1 Swarm Verification

Swarm verification is a promising technique to accelerate time optimal reachability analysis. The large number of random searches can explore different parts of the state-space, thus finding different traces to the goal states in parallel and avoiding local optimality.

[3]Only distributed state-space exploration algorithms in paper B has its communication overhead evaluated because this paradigm is much more communication intensive by sending states

Algorithms. In paper A we developed the following four swarm algorithms using the random-depth-first search (RDFS) as the cornerstone. The two main reasons are: (1) in [BF01] RDFS was proved to be effective in finding sequences of usable (not necessarily optimal) solutions in a very large inexhaustible state-space; (2) we can easily generate many different randomized search strategies by configuring RDFS instances with diverse random seeds.

P-RDFS The basic swarm algorithm where all UPPAAL instances do RDFS independently in parallel with diverse random seeds. Local pruning within each instance is enabled. Any instance that completes first stops other peers.

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

S-Mix A variant of S-RDFS where one UPPAAL instance runs breadth-first search (BFS) rather than RDFS.

S-Agent The agent-based version of S-Mix, where the BFS instance serves as root, while the rest 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 clear its stored state-space and ask for a new state.

We built the above algorithms incrementally and systematically by including one advanced characteristic at a time. This allows us to clearly reveal the effect of a new characteristic in benchmark experiments by comparing the performance difference between the current algorithm and its previous more primitive algorithm.

- **S-RDFS vs. P-RDFS.** We compare S-RDFS against P-RDFS for evaluating the benefit of sharing costs among swarm instances.
- **S-Mix vs. S-RDFS.** We use one BFS instance in S-Mix because in UPPAAL BFS often runs and completes much faster than DFS/RDFS. The empirical explanation is that DFS/RDFS can cause higher degree of fragmentation of the underlying symbolic state-space, thus requiring generating many more symbolic states [BHV00]. BFS however 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 combines advantages of BFS and RDFS such that RDFS can report usable solutions fast and BFS may terminate the global swarm verification process fast.
- **S-Agent vs. S-Mix.** An RDFS agent is a light weight UPPAAL instance. It always clears its stored state-space when periodically asking for a new state from BFS root to search from. Therefore agents will have much smaller memory footprint than the full-length UPPAAL RDFS instances of S-Mix.

Results. We implemented the four swarm algorithms, and compared the performance of sequential and swarm algorithms by the benchmark experiments of four big models on a dedicated model-checking cluster (MCC)[4]. We arrived at the following conclusions [ZNL16b] regarding the four metrics.

- **Sequential Algoirthms:** BFS terminates much faster than DFS or RDFS.
- **Swarm Algorithms.** Conclusions for different swarm algorithms:
  - **P-RDFS:** It is much faster than sequential algorithms at finding near optimal results, especially when the model is infeasible for sequential BFS.
  - **S-RDFS:** Exchanging costs can in general speed up finding and proving the optimal results by 15% and 5% respectively, especially showing its usefulness when using a low number of cores.
  - **S-Mix:** Combining benefits of BFS and RDFS, S-Mix can report results and terminate faster than S-RDFS. Costs reported by RDFS node can help BFS root in pruning, but fine-grained improved costs may backfire, leaving room for optimization.
  - **S-Agent:** Besides the virtue of S-Mix, S-Agent has only 1/3 memory consumption of S-Mix.

Summary. 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 cost is beneficial for speedup at lower core settings.

3.1.2 Distributed State-Space Exploration

Paper B realized time optimal reachability analysis based on the second distributed computing paradigm – distributed memory state-space exploration [ZNL16a]. This paradigm may accelerate TOR in three ways: (1) the state-space is now partitioned and distributed among distributed CPU so that multiple worker processes now share the state-space exploration workload, thus the execution time may potentially be shortened greatly; (2) more traces/state-space will be explored in parallel, thus the fastest trace will be potentially found even faster than the swarm algorithm; (3) the disjoint parts of state-space are stored in the distributed memory, allowing fully use the memory of a cluster and handle even larger models than swarm verification.

Algorithms. We developed and investigated five distributed memory TOR algorithms [ZNL16a].

[4]MCC cluster at AAU: `https://sites.google.com/site/mccaau`

D-BFS Distributed breath-first search. Each worker runs local BFS while exchanging states with other workers.

D-BFS^{S} Distributed strict order BFS (also named *level synchronized* BFS). A synchronization protocol will ensure all workers completely explore states on the same current BFS level before moving on to the next level.

D-DFS Distributed depth-first search. Same principle as D-BFS except traversing depth-first.

D-DFS^{G} Distributed greedy DFS. In addition to D-DFS, each worker always picks the successor state of the lowest cost in each iteration.

D-RDFS Distributed random depth-first search. Same principle as D-DFS with a randomly picked successor state.

It is worth noting that except **D-BFS^{S}**, the global search orders of the other algorithms only approximate BFS/DFS/DFS^{G}/RDFS. Due to the varying communication delay or workload on computing nodes, states are received in nondeterministic order from run to run. The motivation of D-BFS^{S} is to keep a strong point of BFS that BFS causes very low fragmentation of the underlying symbolic state-space thus may runs faster than DFS-based orders (described previously in Section 3.1.1). However **D-BFS^{S}** also inherits the drawback of BFS that it typically only reports results late when it has searched nearly all states, making it infeasible for very large state-spaces. Compared with the swarm verification, there are two design challenges to consider for the distributed memory TOR algorithms.

- **State-space partition.** We applied a static hash function on the discrete part[5] of a symbolic state to compute the unique process ID that the symbolic state belongs to. This partition is uniform and leads to the lowest locality[6]. But it facilitates inclusion checking in the distributed context, because all symbolic states with the same discrete part will destine to the same worker process for deterministic and easy inclusion checking [ZNL16a]. A symbolic state that does not belong to the current worker process is send to its owner process by message passing.
- **Termination detection.** A distributed computation is globally terminated iff every process is locally terminated and there is no message in transit in the network. It is a non-trivial task because none process has complete knowledge of the global state. Nevertheless, the root process must infer when global computation and underlying communication have terminated. We applied the well-known *Safra protocol* [Tel01] for this purpose.

[5]Discrete part includes a vector of locations from all parallel component timed automata and the values for all discrete variables in the model.

[6]The probability for a newly generated successor state to stay local is $1/N$ if the algorithm is deployed in a cluster of N worker processes

Results. We implemented the five distributed memory algorithms on the MCC cluster. A detailed account for the implementation level contribution is provided in Part II of this thesis. We performed benchmark experiments for both sequential and distributed algorithms using the same four models and same settings as in the swarm verification. We arrived at the following conclusions [ZNL16a] regarding the four metrics listed above.

- **Sequential Algoirthms:** BFS terminates the fastest. DFS^G is fastest at finding the optimal result in general, but may be trapped by fine grained local optimal in some model.
- **Distributed Algorithms:** DFS-based algorithms are generally faster than sequential and swarm algorithms at finding near optimal results.
  - **D-BFS^S:** terminates the fastest in many cases, but suffers high synchronization overhead at high core settings.
  - **D-DFS^G:** can find the optimal result very fast in general followed by D-DFS/D-RDFS. It may be slower than the optimized sequential DFS^G at low core settings due to fragmentation.
  - The exact performance may depend on characteristics of the model.

Summary. For larger models, distributed algorithms can terminate while sequential algorithms cannot. D-BFS^S can terminate fast thus prove the optimal results fast. D-DFS/D-DFS^G/D-RDFS are good at finding (near-) optimal results. Distributed algorithms have high communication overhead.

3.1.3 Comparing Swarm with Distributed Memory Paradigms

Comparing the swarm verification with distributed memory state-space exploration algorithms, we summarize their pros and cons in Table 1.

Table 1: Swarm vs. Distributed

Swarm Pro	**Distributed Pro**
• Get high quality results fast under time- and memory constrains. • Easier to implement.	• Data parallelism accelerates state-space exploration • Handle larger models. State-space is partitioned and stored in distributed memories
Swarm Con	**Distributed Con**
• No data parallelism. Barely reduce the runtime to terminate thus prove the optimal result. • Duplicate work, duplicate copy of (partial) state space.	• Harder to implement. • Communication overhead is high due to sending states.

3.2 Multi-Objective Optimal Reachability

Multi-objective optimization (MOO) problem is optimizing a plan or a schedule regarding a variety of quantitative objectives in the best balance like the makespan of a schedule, performance, energy consumption, resource intensiveness, risk assessment etc. Because in most cases a subset of these objectives are conflicting, there may not always exist any single solution that can simultaneously optimize all objectives. Vilfredo Pareto (1848–1923) proposed the well-known concept of *Pareto optimality* as "the state of allocating resources where it is impossible to make any one individual better off without making at least one individual worse off." A solution is called *Pareto optimal* if none of the objectives can be improved in value without degrading some of the other objective values. Without additional preference information, all Pareto optimal solutions are considered equally good [ZNL+16c].

In paper C we proposed the modeling formalism of *simple priced automata* (SPTA) to model a subset of multi-objective scheduling problems. SPTA is a simple extension of TA where every action transition carries a vector prices for multiple cost variables. Every cost variable accumulates the prices on action transitions along a run. There exist more general *multi-priced timed automata* (MPTA) that also support price rates on locations. The optimal reachability analysis on MPTA was proved decidable [LR08]. However, the model checking problem on MPTA was proven undecidable [BBR04].

Data structures and reachability algorithms on MPTA are difficult to implement. Multiple cost variables with respect to price rates entail constructing high-dimensional priced zones and complex operations on priced zones. In contrast, SPTA only need a vector of integer variables to maintain different accumulated costs. SPTA may also suffice for a number of classical scheduling cases, where the tasks' spans are pre-determined thus the energy consumption of every task can be approximated in advance, or the resources required by every task are not affected by the task's span at all.

3.2.1 Pareto Optimal Reachability Analysis

Definition 16. *Let $\bar{c} = [c_1, c_2, \ldots, c_k]$, $\bar{b} = [b_1, b_2, \ldots, b_k]$ denote two cost vectors of integers. Vector $\bar{c}$* Pareto dominates *vector $\bar{b}$ (written as $\bar{c} \prec \bar{b}$), iff both the following conditions are true [ZNL+16c]:*

(1) $\forall i \in \{1, \ldots, k\} \quad c_i \leq b_i$;
(2) $\exists j \in \{1, \ldots, k\} \quad c_j < b_j$.

Definition 17. *Given a set of cost vectors $\mathcal{C}$, a cost vector $\bar{c}$ is* Pareto optimal *in $\mathcal{C}$ if there does not exist another cost vector $\bar{b}$ in $\mathcal{C}$ such that $\bar{b} \prec \bar{c}$. The set of Pareto optimal results is called the* Pareto frontier *[ZNL+16c].*

We provided the *Pareto optimal reachability* (POR) algorithm [ZNL+16c] for Uppaal to compute Pareto optimal costs when reaching target goal states.

The algorithm resembles Algorithm 2 in Section 2.2 for time optimal reachability analysis. The extensions are five folded as follows.

1. The symbolic state of SPTA extends that of TA as $(\ell, \langle Z, \bar{c}\rangle)$, where $\bar{c}$ is the cost (or cost vector) of a finite run ρ that ends in (ℓ, v) and $v \in Z$. Therefore, symbolic states in SPTA with the same ℓ and Z are discriminated by $\bar{c}$. We call $\langle Z, \bar{c}\rangle$ the *discrete priced zone* of a symbolic state in SPTA. Note that a UPPAAL model may contain both cost variables and normal variables, so we need to identify cost variables in UPPAAL.
2. We define Pareto dominance between discrete priced zones as: $\langle G, \bar{u}\rangle \preceq \langle Z, \bar{c}\rangle$ iff $Z \subseteq G \wedge (\bar{u} = \bar{c} \vee \bar{u} \prec \bar{c})$. That is, $\langle G, \bar{u}\rangle$ dominates $\langle Z, \bar{c}\rangle$,if zone Z is included by or equal to zone G, and cost vector $\bar{u}$ is equal to or Pareto dominates cost vector $\bar{c}$.
3. The POR algorithm uses the Pareto dominance relation between discrete priced zones for *Pareto inclusion checking* of symbolic states. For every new waiting state $(\ell, \langle Z, \bar{c}\rangle)$, if $\exists(\ell, \langle G, \bar{u}\rangle) \in$ PASSED s.t. $\langle G, \bar{u}\rangle \preceq \langle Z, \bar{c}\rangle$, then $(\ell, \langle Z, \bar{c}\rangle)$ is discarded for further exploration.
4. A global container named frontier maintains the Pareto optimal cost vectors at goal states. When a goal state is reached, the current cost at goal is checked for Pareto dominance with the solutions in the frontier, and the frontier is updated if necessary. In the pruning process, a state is discarded if its cost is dominated by a solution in the frontier.
5. The algorithm returns the frontier. In UPPAAL, multiple runs, each of which corresponding to a Pareto optimal solution inside the frontier, are also computed and stored into different files.

We also introduced three extensions to the basic POR algorithm to make it more powerful and flexible. These extensions however, work under specific additional conditions (Section 3.2 in paper C).

1. Support the formulation of objectives as a vector of objective functions $F(\bar{c}) = [f_1(\bar{c}), f_2(\bar{c}), \ldots, f_n(\bar{c})]$ parameterized by the cost vector $\bar{c}$;
2. Support a global clock (let us call it `now`) as a singular objective function to measure the makespan (accumulated delay) of a finite run;
3. Support negative prices on action transitions.

We extended UPPAAL with a new query to enable Pareto optimal reachabilty analysis of SPTA as [ZNL+16c]:

$$\texttt{PO}\ (f_1, f_2, \ldots, f_k)\ [-(\texttt{L1}|\texttt{L2})] : E\Diamond\varphi,$$

where `PO` is the keywords for Pareto optimum, f_i $(i \in [1, k])$ are objective functions on cost variables. Next comes the optional switches to disable pruning or Pareto inclusion checking. $E\Diamond\varphi$ is a normal reachability query.

Results. We implemented the Pareto optimal reachability algorithm in UPPAAL. We demonstrated the usefulness of this new feature by two case studies: (1) task graph scheduling, and (2) GOMX-3 nano satellite scheduling. Both cases show conflicting optimization objectives. Case 1 aims to finish a group of tasks fast while minimizing the overall energy consumption. Case 2 aims to maximize the productivity of satellite while minimizing the battery depletion risks. The tool can provide a set of Pareto optimal schedules, from which we choose one that optimizes all objectives in the best balance.

Summary. We proposed Pareto optimal reachability analysis on the simple priced timed automata. The technique is suitable for solving a subset of multi-objective scheduling problems, where the prices are independent of the elapsing time of a task in locations. Inclusion checking works on discrete priced zones, involving computing the Pareto dominance relation between cost vectors. The algorithm was implemented in UPPAAL with the extensions supporting objective functions, observer clocks and negative prices.

3.3 Verifying and Evaluating Timed Game Strategies

Controller synthesis can derive a strategy for a timed game for a reachability or safety objective. The strategy ensures the hard real-time guarantee for a controller to handle uncertainties of the environment and win the game. Paper D proposed to adopt the strategy back into the timed game as a closed system and examine robustness, reliability or performance of this strategy. Engineers can model check this strategy for additional correctness properties. More interestingly, engineers can annotate dynamics and stochasticity on top of the timed game model. This creates a stochastic timed game, which may behave closer to real cyber-physical systems. Then engineers can use statistical model checking to evaluate how the strategy supervises the stochastic timed game in terms of various reliability or performance aspects.

The effect of doing model checking (MC) and statistical model checking (SMC) of the closed system (comprising strategy and timed game model) is

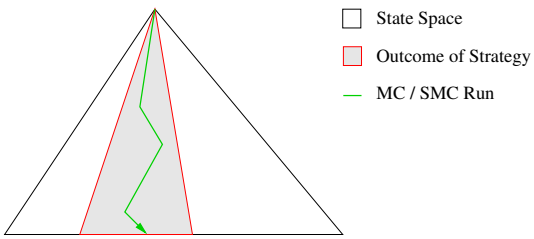

Fig. 4: MC and SMC inside the Outcome of Strategy

illustrated in Figure 4. The MC and SMC runs are constrained inside the outcome defined by the strategy. Two methods to exploit the strategy are presented in paper D. The first method translates the strategy into a controller timed automaton, which is then enclosed with the (stochastic) timed game model into new closed models. The second method keeps this strategy in memory, which is consulted by (statistical) model checking on-the-fly.

Method 1. The strategy printed out by UPPAAL-TIGA is list of (zone, action) pairs indexed by locations. The action in a pair can be a controller action, a bounded delay, or an unbounded delay. Translating a strategy into a TA consists of transforming all the (zone, action) pairs associated to one location into a basic TA. The complete controller TA is obtained by repeating the same translation procedure for all locations and connecting all resulting basic TA to the same initial state [DFLZ14]. To make the controller TA enforce control over TGA for MC and SMC, three steps are need to establish the two-way connections between the controller TA and all components in the TGA.

1. Use global location variables for each component in the TGA model, and use global clocks only. This allows the locations and clock valuations of the TGA components observable to the controller TA.
2. Declare channels for uncontrollable transitions, so that controller TA can monitor when uncontrollable transitions are taken.
3. Declare channels for controllable transitions, so that controller TA can give commands to the TGA components.

Method 2. A UPPAAL extension was built to connect UPPAAL-TIGA, UPPAAL and UPPAAL-SMC. The semantics and algorithms of MC and SMC were extended in order to apply the synthesized in-memory strategy when exploring the state-space in MC, and when generating random runs in SMC. The successor states generation under a given strategy is similar to standard MC or SMC when considering uncontrollable transitions because they are played by the environment. The opponent is stochastic for the purpose of SMC and when doing MC, all possible successors are tried. However, only the controllable transitions allowed by the strategy are allowed. In addition, delay is constrained by the delays of the strategy, i.e., if a controllable transition is to be taken after 5 time units, controller will not delay more than that. For MC, the strategy specifies how much delay is allowed and this constrains the standard delay operation. For SMC, this is resolved naturally through the race between components [DFLZ14]. Finally, if the TGA model are annotated with dynamic or stochastic information, the annotations are automatically ignored in the controller synthesis and model checking phases and only taken into effect in the statistical model checking phase.

Results. For method 1, we built the rules for (1) translating a strategy into a controller TA, and (2) enclosing the controller TA with the timed game model

into a closed system. For method 2, we implemented the tool integration of SYN, MC, SMC engines inside UPPAAL. We used each method to perform the same two case studies, and cross checked the experiment results of both methods. We also evaluated the performance of both methods regarding the running time of intensive SMC jobs.

Summary. Method 1 demonstrates the translation procedure of a strategy into a controller TA. Even thought it can be automated, the translated controller TA is usually huge and not very interesting. Enclosing the controller TA with the TGA model still needs a lot of manual work. Running the closed model is slower and needs more memory, because the controller TA usually has a lot of states and interactions with the rest components of TGA. Method 2 is preferred because it is very convenient and fast. Actually this work is used in UPPAAL-STATEGO [DJL+15], which aims at optimizing the strategy for a stochastic timed game towards some goal[7].

3.4 Concluding Remarks

This chapter reviews our work in four papers: distributed time optimal reachability analysis by swarm verification and distributed memory state-space exploration, Pareto optimal reachability analysis, and verifying and evaluating timed game strategies. The first three papers are about timed automaton based scheduling and planning. The fourth paper is about exploiting the synthesized strategy of a timed game.

At the technical side, now UPPAAL tool suite has three new members: distributed version of timed optimal reachability, Pareto optimal reachability, and UPPAAL-STATEGO which includes our work in paper D. The distributed computing module built in papers A and B is generic and reusable by the other members in the UPPAAL family to build their distributed versions.

[7] Traffic dilema example `http://people.cs.aau.dk/~marius/stratego/examples.html`

4 Conclusions and Future Work

"Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design and continuing throughout development and later life cycle phases [oSEI]." MBSE helps industry to gain quality, productivity improvements, and lower risk. In particular, MBSE can facilitate the development of trust worthy safety critical systems. A series of automated model-based techniques have been developed for MBSE. In this thesis, we contribute to time optimal reachability analysis, Pareto optimal reachability analysis, and evaluation of timed game strategies.

Conclusions

We fulfilled the initial research objectives defined in section 1.4. We found that: (1) two distributed paradigms – swarm verification and distributed memory exploration – are valuable for developing distributed time optimal reachability algorithms; (2) the notions of Pareto optimum and Pareto dominance are applicable for multi-objective scheduling problems on simple priced timed automata; (3) it is possible and useful to verify and evaluate the synthesized strategy of a timed game in an integrated manner.

Firstly, we proposed distributed time optimal reachability algorithms running on the cluster to efficiently accelerate finding the trace with optimal (or near optimal) results and mitigate the state-space explosion problem. Inside Uppaal, four algorithms were developed for the swarm verification paradigm, and five algorithms were developed for the distributed memory state-space exploration paradigm. The benchmark experiments showed very promising acceleration for finding the optimal results or high quality near optimal results. The distributed computing module is also reusable by other members in the Uppaal tool suite.

Secondly, as the extension of time optimal reachability analysis, we proposed the Pareto optimal reachability analysis on the simple priced timed automata for solving multi-objective scheduling problems. A set of schedules are optimized in Pareto optimum regarding a variety of quantitative objectives. The objectives are either objective functions parameterized by cost variables in the model, or an observer clock for measuring the makespan.

Thirdly, we proposed two methods to verify and evaluate a synthesized strategy – examining the consequences of adopting this strategy in the timed game. Timed games are useful for modeling scheduling and planning problems in the dynamic and partially controllable environment. Method 1 is ad hoc. The synthesized strategy is translated into a controller timed automaton, which is encapsulated as an extra interacting component with the timed game into a closed system. Then the closed system is loaded into normal

UPPAAL for MC or into UPPAAL-SMC for SMC separately. Method 2 is integrated. We combined three engines of SYN, MC and SMC into a single tool. The synthesized strategy is stored inside the memory and consulted on-the-fly by MC and SMC. The integration requires an extension in the semantics and algorithms of MC and SMC. The in-memory strategy acts as a supervisor when generating the state-space in MC, and as an extra player when generating random runs in SMC.

In conclusion, we contributed to the timed automaton based scheduling and planning techniques. In particular we have shown how distributed algorithms may be used to speedup and scale up the computation.

Future Work

There are many topics for possible future research and development based on this thesis. Particularly, we find the following topics are interesting.

- Investigate more search strategies for time or cost optimal reachability analysis such as trying to combine planning algorithms or ideas from artificial intelligence area.
- Develop hybrid algorithms that combine the benefits from distributed memory and swarm verification.
- Develop distributed algorithms to support the liveness model checking properties to make a fully functional distributed UPPAAL.
- Develop distributed algorithms for Pareto optimal reachability analysis on multi-priced timed automata, and controller synthesis on timed game automata.
- Further improve the performance by developing parallel and distributed algorithms applying multi-core shared memory and GPU acceleration.
- Improve robustness and fault tolerance of distributed implementations.

References

[AAM06] Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. Scheduling with timed automata. *Theoretical Computer Science*, 354(2):272–300, 2006.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AILS07] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007.

[AM99] Eugene Asarin and Oded Maler. As soon as possible: Time optimal control for timed automata. In *HSCC*, volume 1569 of *LNCS*, pages 19–30. Springer, 1999.

[AMPS98] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *IFAC*, pages 469–474. Elsevier Science, 1998.

[BBBC12] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Designing fast LTL model checking algorithms for many-core gpus. *Journal of Parallel and Distributed Computing*, 72(9):1083–1097, 2012.

[BBHM05] Gerd Behrmann, Ed Brinksma, Martijn Hendriks, and Angelika Mader. Production scheduling by reachability analysis - A case study. In *IPDPS*. IEEE Computer Society, 2005.

[BBR04] Thomas Brihaye, Véronique Bruyère, and Jean-François Raskin. Model-checking for weighted timed automata. In *FORMATS*, volume 3253 of *LNCS*, pages 277–292. Springer, 2004.

[BBvR10] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *HiBi/PDMC*, pages 4–7. IEEE, 2010.

[BCD+07] Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. UPPAAL-TIGA: Time for playing games! In *CAV*, number 4590 in LNCS, pages 121–125. Springer, 2007.

[BCD+12] Peter E. Bulychev, Franck Cassez, Alexandre David, Kim Guldstrand Larsen, Jean-François Raskin, and Pierre-Alain Reynier. Controllers with minimal observation power (application to timed systems). In *ATVA*, volume 7561 of *LNCS*, pages 223–237. Springer, 2012.

[BCFL04] Patricia Bouyer, Franck Cassez, Emmanuel Fleury, and Kim Guldstrand Larsen. Optimal strategies in priced timed game automata. In *FSTTCS*, volume 3328 of *LNCS*, pages 148–160. Springer, 2004.

[BDL04] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In *SFM*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.

[BDL+12] Peter E. Bulychev, Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikucionis, and Danny Bøgsted Poulsen. Checking and distributing statistical model checking. In *NFM*, volume 7226 of *LNCS*, pages 449–463. Springer, 2012.

[BDLY03] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, and Wang Yi. A tool architecture for the next generation of UPPAAL. In *UNU/IIST 10th Anniversary Colloquium. Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 352–366, 2003.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *CAV*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998.

[BDMP03] Patricia Bouyer, Deepak D'Souza, P. Madhusudan, and Antoine Petit. Timed control with partial observability. In *CAV*, volume 2725 of *LNCS*, pages 180–192. Springer, 2003.

[Beh05] Gerd Behrmann. Distributed reachability analysis in timed automata. *STTT*, 7(1):19–30, 2005.

[Bel57] Richard Bellman. A Markovian Decision Process. *Indiana University Mathematics Journal*, 6:679–684, 1957.

[BF01] Gerd Behrmann and Ansgar Fehnker. Efficient guiding towards cost-optimality in uppaal. In *TACAS*, volume 2031 of *LNCS*, pages 174–188. Springer, 2001.

[BFH+01] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Guldstrand Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In *HSCC*, volume 2034 of *LNCS*, pages 147–161. Springer, 2001.

[BGH+16] Morten Bisgaard, David Gerhardt, Holger Hermanns, Jan Krcál, Gilles Nies, and Marvin Stenger. Battery-aware scheduling in low orbit: The gomx-3 case. In *FM*, volume 9995 of *LNCS*, pages 559–576. Springer, 2016.

[BGK+02] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated verification of an audio-control protocol using UPPAAL. *The Journal of Logic and Algebraic Programming*, 52-53:163–181, 2002.

[BHV00] Gerd Behrmann, Thomas Hune, and Frits W. Vaandrager. Distributing timed model checking - how the search order matters. In *CAV*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BKLN14] Abdeldjalil Boudjadar, Jin Hyun Kim, Kim Guldstrand Larsen, and Ulrik Nyman. Compositional schedulability analysis of an avionics system using UPPAAL. In *ICAASE*, volume 1294 of *CEUR Workshop Proceedings*, pages 140–147. CEUR-WS.org, 2014.

[BLR04] Gerd Behrmann, Kim Guldstrand Larsen, and Jacob Illum Rasmussen. Priced timed automata: Algorithms and applications. In *FMCO*, volume 3657 of *LNCS*, pages 162–182. Springer, 2004.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology*, 20(4):14:1–14:64, September 2011.

[BLvdPW11] Stefan Blom, Bert Lisser, Jaco van de Pol, and Michael Weber. A database approach to distributed state-space generation. *Journal of Logic and Computation*, 21(1):45–62, 2011.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BvdPW10] Stefan Blom, Jaco van de Pol, and Michael Weber. Ltsmin: Distributed and symbolic reachability. In *CAV*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010.

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, CAV '02, pages 359–364, London, UK, UK, 2002. Springer-Verlag.

[CDF+05] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, volume 3653 of *LNCS*, pages 66–80. Springer, August 2005.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[CFL+08] Edmund M. Clarke, James R. Faeder, Christopher James Langmead, Leonard A. Harris, Sumit Kumar Jha, and Axel Legay. Statistical model checking in biolab: Applications to the automated analysis of t-cell receptor signaling pathway. In *CMSB*, volume 5307, pages 231–250. Springer, 2008.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CL00] Franck Cassez and Kim Guldstrand Larsen. The impressive power of stopwatches. In *CONCUR*, volume 1877 of *LNCS*, pages 138–152. Springer, 2000.

[Cla08] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *LNCS*, pages 1–26. Springer, 2008.

[DDL+12] Alexandre David, Dehui Du, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical model checking for stochastic hybrid systems. In *HSB*, volume 92 of *EPTCS*, pages 122–136, 2012.

[DDL+13] Alexandre David, Dehui Du, Kim Guldstrand Larsen, Axel Legay, and Marius Mikucionis. Optimizing control strategy using statistical model checking. In *NFM*, volume 7871 of *LNCS*, pages 352–367. Springer, 2013.

[DFLZ14] Alexandre David, Huixing Fang, Kim Guldstrand Larsen, and Zhengkui Zhang. Verification and performance evaluation of timed game strategies. In *Formal Modeling and Analysis of Timed Systems - 12th International Conference, FORMATS 2014, Florence, Italy, September 8-10, 2014. Proceedings*, volume 8711 of *LNCS*, pages 100–114. Springer, 2014.

[DJL+15] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikucionis, and Jakob Haahr Taankvist. Uppaal stratego. In *TACAS*, volume 9035 of *LNCS*, pages 206–211. Springer, 2015.

[DL77] Stuart E. Dreyfus and Averill M. Law. *Art and Theory of Dynamic Programming*. Academic Press, Inc., Orlando, FL, USA, 1977.

[DLL+11a] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, Jonas van Vliet, and Zheng Wang. Stochastic semantics and statistical model checking for networks of priced timed automata. *CoRR*, abs/1106.3961, 2011.

[DLL+11b] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Zheng Wang. Time for statistical model checking of real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 349–355. Springer, 2011.

[DLL+15] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *STTT*, 17(4):397–415, 2015.

[DRA15] Kirov D.A, Passerone R., and Ozihganov A.A. A methodology for design space exploration of real-time location systems. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 15(4):551–567, 2015.

[EMP10] Rüdiger Ehlers, Robert Mattmüller, and Hans-Jörg Peter. Combining symbolic representations for solving timed games. In *FORMATS*, volume 6246 of *LNCS*, pages 107–121. Springer, 2010.

[ESB+11] Stefan Edelkamp, Damian Sulewski, Jiri Barnat, Lubos Brim, and Pavel Simecek. Flash memory efficient LTL model checking. *Science of Computer Programming*, 76(2):136–157, 2011.

[Feh99] Ansgar Fehnker. Scheduling a steel plant with timed automata. In *RTCSA*, pages 280–286. IEEE Computer Society, 1999.

[Feh00] A. Fehnker. *Bounding and Heuristics in Forward Reachability Algorithms*. UB Nijmegen [Host], 2000.

[FGI12] Jean Fourie, Jaco Geldenhuys, and Cornelia Inggs. Improving communication for distributed model checking. In *SAICSIT*, SAICSIT, pages 41–50. ACM, 2012.

[GABR14] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 - A modern refinement checker for CSP. In *TACAS*, pages 187–201, 2014.

[GABR16] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3: a parallel refinement checker for CSP. *STTT*, 18(2):149–167, 2016.

[GdV99] Jaco Geldenhuys and P. J. A. de Villiers. Runtime efficient state compaction in SPIN. In *SPIN Workshops*, volume 1680 of *LNCS*, pages 12–21. Springer, 1999.

[GMS13] Hubert Garavel, Radu Mateescu, and Wendelin Serwe. Large-scale distributed verification using CADP: beyond clusters to grids. *Electronic Notes in Theoretical Computer Science*, 296:145–161, 2013.

[GNT16] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[GRU08] Jan Friso Groote, Michel A. Reniers, and Yaroslav S. Usenko. Verification of networks of timed automata using mcrl2. In *IPDPS*, pages 1–8. IEEE, 2008.

[HB13] Steven P. Haveman and G. Maarten Bonnema. Requirements for high level models supporting design space exploration in model-based systems engineering. In *CSER*, volume 16 of *Procedia Computer Science*, pages 293–302. Elsevier, 2013.

[HJG08] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification. In *ASE*, pages 1–6. IEEE, 2008.

[HLP00] Thomas Hune, Kim Guldstrand Larsen, and Paul Pettersson. Guided synthesis of control programs using uppaal. In *ICDCS Workshop*, pages E15–E22, 2000.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[Hol03] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, 2000.

[JKM12] Kurt Jensen, Lars Michael Kristensen, and Thomas Mailund. The sweep-line state space exploration method. *Theoretical Computer Science*, 429:169–179, 2012.

[JR09] Claude Jard and Olivier H. Roux. *Communicating Embedded Systems – Software and Design*. John Wiley & Sons, Inc, New Jersey, USA, December 2009.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[Lar10] Kim Guldstrand Larsen. Symbolic and compositional reachability for timed automata. In *RP*, volume 6227 of *LNCS*, pages 24–28. Springer, 2010.

[LKZ16] Christine Largouët, Omar Krichen, and Yulong Zhao. Temporal Planning with extended Timed Automata. In *ICTAI*, SAN JOSE, United States, Nov 2016. IEEE.

[LR08] Kim Guldstrand Larsen and Jacob Illum Rasmussen. Optimal reachability for multi-priced timed automata. *Theoretical Computer Science*, 390(2-3):197–213, January 2008.

[McN93] Robert McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[MKM15] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. Hardware verification using software analyzers. In *IEEE Computer Society Annual Symposium on VLSI*, pages 7–12. IEEE, 2015.

[MLR+10] Marius Mikucionis, Kim Guldstrand Larsen, Jacob Illum Rasmussen, Brian Nielsen, Arne Skou, Steen Ulrik Palm, Jan Storbank Pedersen, and Poul Hougaard. Schedulability analysis using uppaal: Herschel-planck case study. In *ISoLA*, volume 6416 of *LNCS*, pages 175–190. Springer, 2010.

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.

[NC97] David M. Nicol and Gianfranco Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 47(2):153–167, 1997.

[Nor98] James R. Norris. *Markov chains*. Cambridge series in statistical and probabilistic mathematics. Cambridge University Press, 1998.

[NTY00] Peter Niebert, Stavros Tripakis, and Sergio Yovine. Minimum-time reachability for timed automata. In *IEEE Mediteranean Control Conference*, 2000.

[oSEI] Technical Operations International Council on Systems Engineering INCOSE. INCOSE Systems Engineering Vision 2020. Technical report.

[Pel98] Doron A. Peled. Ten years of partial order reduction. In *CAV*, volume 1427 of *LNCS*, pages 17–28. Springer, 1998.

[PEM11] Hans-Jörg Peter, Rüdiger Ehlers, and Robert Mattmüller. Synthia: Verification and synthesis for timed automata. In *CAV*, volume 6806 of *LNCS*, pages 649–655. Springer, 2011.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[RBL07] Jacob Illum Rasmussen, Gerd Behrmann, and Kim Guldstrand Larsen. Complexity in simplicity: Flexible agent-based state space exploration. In *TACAS*, pages 231–245. Springer, 2007.

[Rus05] John M. Rushby. Automated test generation and verified software. In *VSTTE*, volume 4171 of *LNCS*, pages 161–172. Springer, 2005.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, January 1987.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the Murφ Verifier. In *CAV*, volume 1254 of *LNCS*, pages 256–278. Springer, 1997.

[Sis04] A.Prasad Sistla. Employing symmetry reductions in model checking. *Computer Languages, Systems and Structures*, 30(3-4):99–137, Oct 2004.

[SVA04] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *CAV*, volume 3114 of *LNCS*, pages 202–215. Springer, 2004.

[SZHV09] Mathijs Schuts, Feng Zhu, Faranak Heidarian, and Frits W. Vaandrager. Modelling clock synchronization in the chess gmac WSN protocol. In *QFM*, volume 13 of *EPTCS*, pages 41–54, 2009.

[Tel01] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *STACS*, pages 1–13, 1995.

[VBBB09] Kees Verstoep, Henri E. Bal, Jiri Barnat, and Lubos Brim. Efficient large-scale model checking. In *IPDPS*, pages 1–12. IEEE, 2009.

[You05] Håkan Lorens Samir Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon University, January 2005.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *IFIP*, volume 6 of *IFIP Proceedings*, pages 243–258. Chapman & Hall, 1994.

[ZNL16a] Zhengkui Zhang, Brian Nielsen, and Kim Guldstrand Larsen. Distributed algorithms for time optimal reachability analysis. In Martin Fränzle and Nicolas Markey, editors, *Formal Modeling and Analysis of Timed Systems - 14th International Conference, FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings*, volume 9884 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2016.

[ZNL16b] Zhengkui Zhang, Brian Nielsen, and Kim Guldstrand Larsen. Time optimal reachability analysis using swarm verification. In Sascha Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1634–1640. ACM, 2016.

[ZNL+16c] Zhengkui Zhang, Brian Nielsen, Kim Guldstrand Larsen, Gilles Nies, Marvin Stenger, and Holger Hermanns. Pareto optimal reachability analysis for simple priced timed automata. 2016.

Part II

Implementation

Implementation

5 Distributed Time Optimal Reachability Analysis

5.1 Distributed Memory Reachability

This section elucidates important design and technical issues that were not present in papers A and B due to the page limit. We recommend readers to read these two papers first then go back to this chapter. We focus on the distributed memory state-space exploration paradigm because it already covers the technologies needed by swarm verification. Algorithm 3 recalls the distributed algorithms in paper B. It extends the sequential version (Algorithm 2 in Section 2.2) with state-space partitioning, message passing and handling.

5.1.1 Distributed TOR Reachability Algorithms

Definition 18. *Let N be the set of worker processes and p denote the local process ID. The* partition function *is a total mapping:* $\texttt{Hash} : L \rightarrow N$ *from the set of locations to the set of processes. Process p is the* owner *of a symbolic state (ℓ, Z) if* $\texttt{Hash}(\ell) = p$*. A symbolic state is a* local *state if it is generated on its owner process, otherwise it is an* emigrant *state [ZNL16].*

Definition 19. *A process is* active *when doing local search or receiving messages. Initially all processes are active. A process is* idle *when its waiting list is empty and receiving no messages. Computation can* terminate *if all processes are idle and the network has no message in transit [ZNL16].*

Three new variables are used: Ecost maintains external better cost received from the network; Active specifies process status (active or idle); and Terminate controls the `Main` loop. Then two types of messages are defined: (1) UPDATE carries better costs; (2) STATE carries symbolic states. At the beginning of each iteration, `Ecost` compares with `Cost`. If `Ecost` is smaller, a better cost has been found by another process, and `Cost` is updated. If `Cost` is smaller, the current process finds a better cost, which is assigned to `Ecost` and then broadcasted. Line 5 marks process to idle according to Definition

Algorithm 3: Distributed Time Optimal Reachability for D-BFS/D-DFS/D-DFS^{G}/D-RDFS [ZNL16]

(Local Variables)

$\textsc{Waiting} \longleftarrow \begin{cases} \{(\ell_0, Z_0)\} & \text{if } p = \texttt{Hash}(\ell_0), \\ \emptyset & \text{otherwise.} \end{cases}$ $\quad \textsc{Passed} \longleftarrow \emptyset$

$\textsc{Terminate} \longleftarrow \textsc{False}, \textsc{Active} \longleftarrow \textsc{True}, \textsc{Cost} \longleftarrow \infty, \textsc{Ecost} \longleftarrow \infty$

(Message Types)

UPDATE, STATE

Procedure `Main()`

1 **while** $\neg$TERMINATE **do**
2 **if** ECOST $<$ COST **then** COST $\longleftarrow$ ECOST
3 **if** COST $<$ ECOST **then** `UpdateE(`COST`)`, `Broadcast(`UPDATE, COST`)`
4 **if** WAITING $= \emptyset$ **then**
5 **if** *receive no message* **then** ACTIVE $\longleftarrow$ FALSE, `CheckTerm()`
6 continue
7 select (ℓ, Z) from WAITING, ACTIVE $\longleftarrow$ TRUE
8 **if** $(\ell, Z) \models$ `Goal` **then**
9 **if** `MinCost(`ℓ, Z`)` $<$ COST **then**
10 COST $\longleftarrow$ `MinCost(`ℓ, Z`)`
11 **else if** $\neg\exists(\ell, H) \in$ PASSED *s.t.* $Z \subseteq H$ ***and*** `MinCost(`ℓ, Z`)` $<$ COST **then**
12 add (ℓ, Z) to PASSED
13 **forall the** (ℓ', Z') *such that* $(\ell, Z) \rightsquigarrow (\ell', Z')$ **do**
14 **if** $(r \longleftarrow$ `Hash(`ℓ'`)`$) \neq p$ **then** `Send(`STATE, $(\ell', Z'), r$`)`
15 **else** add (ℓ', Z') to WAITING
16 **return** COST

Procedure `UpdateE(`NEWCOST`)`

17 **if** NEWCOST $<$ ECOST **then** ECOST $\longleftarrow$ NEWCOST

Procedure `CheckTerm() // Safra protocol [Tel01]`

18 **if** $p = root$ ***and*** *all processes are idle* ***and*** *no message in transit* **then**
19 TERMINATE $\longleftarrow$ TRUE on all processes

(Message Processing Rules)

20 When receive UPDATE$\langle$NCOST$\rangle$: `UpdateE(`NCOST`)`, ACTIVE $\longleftarrow$ TRUE.
21 When receive STATE$\langle(n, F)\rangle$: add (n, F) to WAITING, ACTIVE $\longleftarrow$ TRUE.

19. The remaining lines inside `Main` resembles the local search in the sequential TOR algorithm, except at line 14 a successor state is hashed to compute its owner process ID and sent out if it does not belong to the current process p. When the root process becomes idle, it invokes the termination detection procedure `CheckTerm`. For message handlers: line 20 updates `Ecost` on reception of a cost message; line 21 adds a received emigrant state into WAITING [ZNL16]. After termination `Goal` is reachable at process p if `Cost` $\neq \infty$, otherwise it is not reachable at process p.

5.1.2 Architecture Design

Figure 5 depicts the implemented distributed reachability pipeline in UPPAAL for Algorithm 3. It differs the sequential reachability pipeline (Figure 3 in Section 2.1) by the additional functional components highlighted in orange red and two affected components annotated by "*":

- **Recv Msg** polls the communication channels for incoming messages, dispatches those received messages to the corresponding message handlers distinguished by the message types.
- **Hand Msg** processes a message by the corresponding message handler. If the message contains a state, the state is conditionally inserted into the PWList. If the message contains a better cost, this external cost may update the local cost. If message is a stop signal, the pump is terminated.
- **Bcast Cost** broadcasts the local cost – reported by the property filter – if it is better than the currently known best external cost.
- **Send State** sends an emigrant state to its owner process. Emigrant states are not stored in the local PWList.

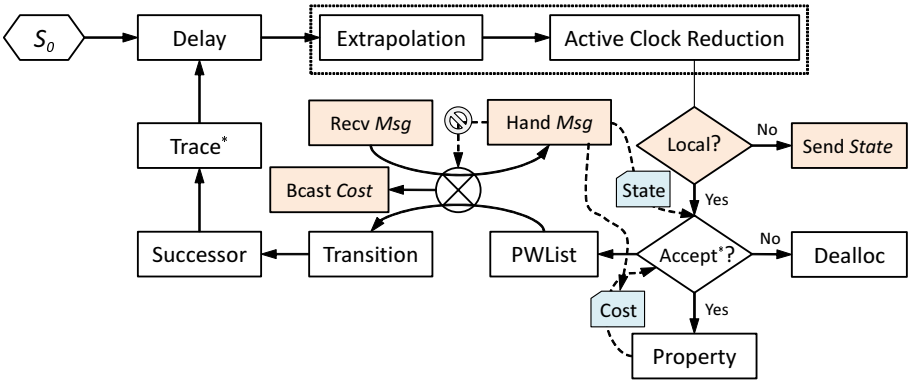

Fig. 5: Distributed Reachability Pipeline of UPPAAL

- **Accept*** as the guard of PWList insertion, it involves not only inclusion checking but also B&B pruning using the best cost found so far.
- **Trace*** now contains only trace segments rather than complete traces. In the distributed context, every trace is split into segments that scatter among a group of worker processes. This raises a question "how to generate a complete well-format time optimal trace." In fact even in the Swarm Agent algorithm, a time optimal trace may be also split into two segments – the head part resides at root and the tail part at an agent.

5.1.3 Implementation Solutions

The target hardware is a cluster with 9 computing nodes. Each node has 1 TB memory (NUMA architecture) and 64 cores @ 2.3 GHz (4 AMD Opteron 6376 Processors each with 16 cores), a 1 TB SATA disk and the Infiniband interconnection [ZNL16].

We used the Mpi library to build a communication module that supports the above mentioned additional functional components inside the distributed pipeline. This module implements a number of advanced features as follows to facilitate development and to gain better performance.

- **Asynchronous Communication.** Mpi offers send and receive APIs in asynchronous mode, which is the key to achieve high performance computing. Local computation is non-blocking, and the Mpi library takes care of communication automatically in the background.
- **Basic and Control Messages.** We defined two message categories: (1) basic messages for carrying the necessary data for the distributed algorithms like costs, states, trace segments etc; (2) control messages for monitoring or imposing status control of the algorithms like initialization, termination,

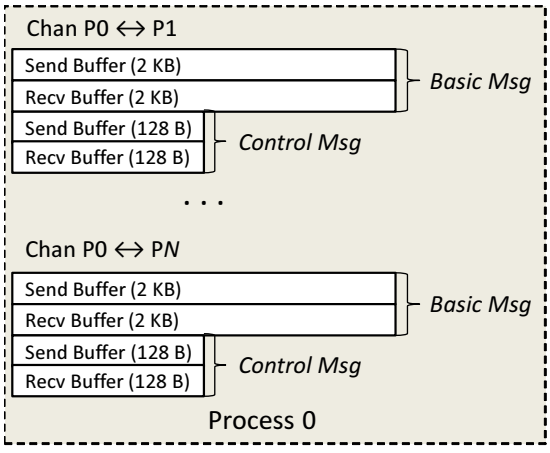

Fig. 6: Message Buffers for Distributed Reachability. In a group of (1+N) processes, process 0 has N channels to each of the other processes like P0 $\leftrightarrow$ P1, . . . , P0 $\leftrightarrow$ PN. For each channel, there are two buffers for sending and receiving basic messages separately, and two smaller buffers for sending and receiving control messages separately. Default capacity is 2 KB for a basic message buffer, and 128 B for a control message buffer. The same rules apply to the rest processes P1 $\sim$ PN.

synchronization. Control messages have higher priority in sending and processing than basic messages.

- **Message Buffer for Channel.** We define a logic communication channel to be the link between two worker processes. On a worker, every channel is associated with four buffers for sending or receiving basic messages and control messages separately as in Figure 6. This buffer design [Bar04] is simple, but can achieve a high level of parallelism using asynchronous communication, because multiple channels can work in parallel without interference.
- **Packet buffering and compression.** It is more efficient to bundle several basic messages into a bigger packet, then compress and send out this packet [VBBB09]. However control messages are sent out immediately.
- **Two Termination Modes.** *Active mode* is broadcasting termination signal (control message) by one worker, and force all other workers to stop. Swarm algorithms use this mode for termination. The other usage for this mode is shutting down the cluster smoothly in case of timeout, memory-out or error. *Passive mode* refers to termination detection by the Safra protocol. Distributed memory algorithms use this mode to terminate the global computation normally. The other usage of this mode is to drain the remaining message in the network after an active termination. All channels are cleared and prepared to initiate the next task in a batch job.
- **Protocol Stack.** There are totally a number of basic and control messages

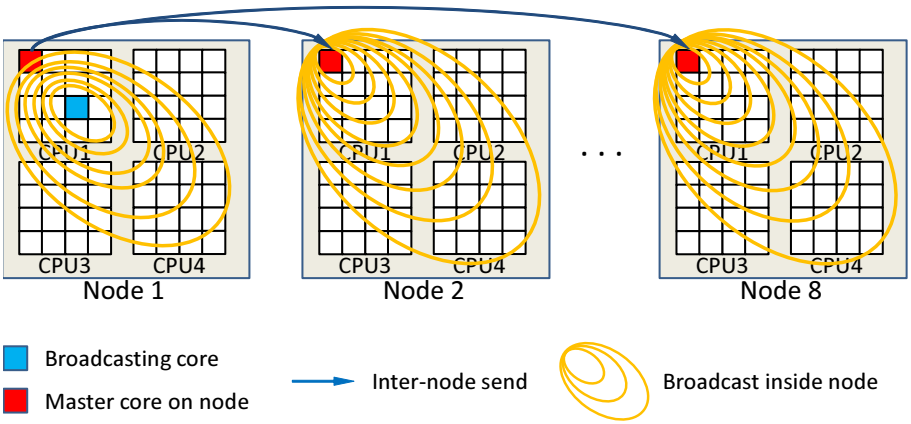

Fig. 7: Message Broadcasting using Virtual Topology. A blue core in Node 1 needs to broadcast a message within 512 cores over 8 nodes. This blue core only broadcasts inside Node 1. Once the red master core in Node 1 receives the broadcast message, it immediately sends this message to the master cores in the other 7 nodes. When the master core on any node, say Node 2 receives this message, it broadcasts this message inside Node 2. Only 7 inter-node communications are needed by this strategy, which would be otherwise $512 - 64 = 448$ inter-node communications.

designed to support the swarm and distributed algorithms. It is sensible to manage these different kinds of messages uniformly inside a protocol stack, where every message type is linked to a dedicated message handler (similar to the reactor pattern [CS95]). Whenever a message is received, a dispatcher will direct this message to the target handler for processing through the uniform interface.

- **Virtual Topology.** Because the semantics of MPI broadcast raw API does not fit[8] our algorithms, we redesigned the broadcast by asynchronous point-to-point communication. Rather than the naive one-to-all message sending, we allow the broadcast to make use of the topology information of the cluster to reduce a lot of inter-node communication which will overload the cluster heavily when running on 256 cores or above as in Figure 7. The key idea is creating a virtual hierarchy. The pre-assigned master core on each node exchanges inter-node broadcast messages over the infiniband network, and relays its received messages to other cores inside its residing node by the much faster internal bus.

5.2 Distributed Trace Collection

Only reporting the span of a time optimal trace to the user is not sufficient. A complete UPPAAL trace should be generated so that engineers can either visualize the trace in the UPPAAL simulator or parse the trace into a usable schedule. However as above mentioned, distributed reachability algorithms raise a challenge for generating such trace, because it is divided into segments in different workers. We developed the distributed trace collection algorithm executed by UPPAAL as a separate phase after the distributed reachability algorithms terminate.

5.2.1 Trace Generation Pipeline

The Trace filter in the reachability pipeline (Figure 5 in Section 5.1.2) is responsible for storing traces. Rather than linking symbolic states (pointers), Trace links action transitions (pointers) and store these actions inside a list container. Symbolic states are volatile and may be deallocated due to inclusion checking, pruning or being sent out. Action transitions in the model are stable. For a finite symbolic trace that is time optimal $\pi = S_0 \overset{d_1}{\rightsquigarrow} S'_0 \overset{a_1}{\rightsquigarrow} S_1 \overset{d_2}{\rightsquigarrow} \cdots \overset{d_n}{\rightsquigarrow} S'_{n-1} \overset{a_n}{\rightsquigarrow} S_n$, Trace practically links the actions of π backward as the *action trace* $\eta = a_\emptyset \leftarrow a_1 \leftarrow a_2 \leftarrow \cdots \leftarrow a_{n-1} \leftarrow a_n$, where $a_\emptyset$ is a *null head* for all actions fired from S_0 to link with. Each action in

[8] `MPI_Bcast` is a collective communication. It is called by all workers, and requires the rank of the broadcast source specified in its arguments. Receivers must know the broadcast source beforehand, which is not the case in our algorithms. `MPI_Bcast` is typically used to initialize the input data for each worker of a cluster before computation.

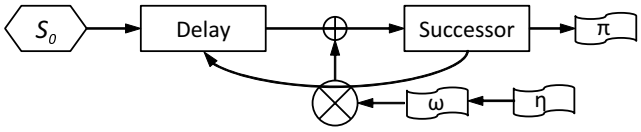

Fig. 8: Trace Generation Pipeline in Sequential UPPAAL

η is accessible by a non-negative integer offset, such that $\eta[0] = a_\emptyset$ and $\eta[i_k] = a_k$ for every offset i_k where $k \in [1, n]$.

For outputting the symbolic trace, UPPAAL feeds an action trace η into a trace generation pipeline (Figure 8) dedicated for computing π. This pipeline has Delay and Successor filters connected in a loop. Firstly η is transformed into a forward sequence $\omega = (a_1, a_2, \cdots, a_{n-1}, a_n)$. In every iteration, an action transition is retrieved (by the pump) from ω in sequence, and inserted into Successor together with the state processed by Delay for computing every symbolic state in π. The loop stops when all actions in ω have been used. Initial state S_0 is put at the head of π, and inserted at the delay stage initially.

Let function $\texttt{TraceID}(\eta)$ return the trace-id (offset of the last action traction) of a finite action trace η. Let $|\eta|$ denote the *length* (number of actions) of η excluding $a_\emptyset$. Given a trace-id, we can extract a unique finite action trace by going through the links backward from the last action a_n to $a_\emptyset$.

Definition 20. *Let function* $\texttt{Trace}(S_0, \texttt{TraceID}(\eta))$ *describes the trace generation pipeline, where* S_0 *is the initial state and* η *is a finite action trace. The complete finite symbolic trace* π *corresponding to* η *is built by concatenating the output of* $\texttt{Trace}(S_0, \texttt{TraceID}(\eta))$ *at the back of* S_0.

5.2.2 Design Choices

Assume when the sequential TOR algorithm (Algorithm 2 in Section 2.2.1) terminates, a finite symbolic trace $\pi = S_0 \rightsquigarrow S_1 \rightsquigarrow S_2 \rightsquigarrow S_3 \rightsquigarrow S_4 \rightsquigarrow S_5 \rightsquigarrow S_6 \rightsquigarrow S_7 \rightsquigarrow S_8$ that is time optimal with the goal state of S_8, exists in the generated state-space. The Trace filter maintains an action trace $\eta = a_\emptyset \leftarrow a_1 \leftarrow a_2 \leftarrow a_3 \leftarrow a_4 \leftarrow a_5 \leftarrow a_6 \leftarrow a_7 \leftarrow a_8$ corresponding to π.

Figure 9-(a) shows an imaginary situation when the distributed memory algorithm (Algorithm 3) running on three workers (with P0 as the root) terminates. The state-space is partitioned, and the finite symbolic trace π is also divided into four segments stored inside the three workers. Emigrant states S_2, S_5 and S_7 were deallocated (struck out in the figure) after being sent to their owner processes. The Trace filter in each process kept an action segment of η locally. Note that action a_8 is dangling (no parent action to connect to) in P1, because its previous action a_7 is in P2. To generate the complete symbolic trace π in this situation, there are two difficulties to solve: (1) how to find all

segments of η among all workers and sort them in order; (2) how to run the trace generation pipeline using segments of η.

For solving the first difficulty, trace-id of each action segment is sent together with the emigrant state in the reachability analysis phase, so that all distributed action segments will maintain the ordered links between them. An action segment can find out in which worker process its preceding segment is stored. When the state message extended with a trace-id is received, three steps are performed at the receiver side: (1) the received state is inserted into the PWList; (2) a new null head is appended to the action list in the Trace filter to match this state, and all action transitions fired out from this state are linked to the newly added null head; (3) the same null head also uses an additional field p to store a pair (`source`, `trace-id`) as the *backward reference* to the preceding action segment in a remote process. In this pair, `source` is

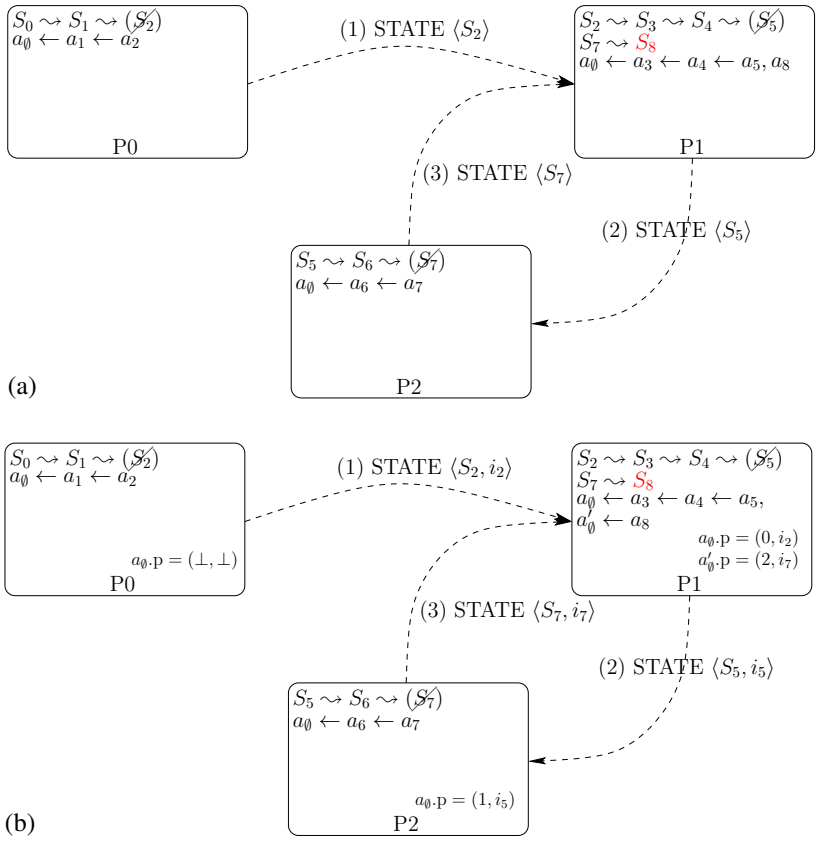

Fig. 9: Distributed Reachability Termination Status. (a) Sending Pure STATE Messages. (b) Carrying the Backward Reference Information trace-id in STATE Messages

ID of the source process sending the emigrant state, and `trace-id`, which is sent together with the state, is the trace-id of the preceding action segment.

Figure 9-(b) shows when trace-ids are sent with emigrant states, the ordered link between action segments of η are maintained and stored in the null heads as $a_\emptyset.p$. The initial null head at P0 uses $a_\emptyset.p = (\perp, \perp)$ to indicate it is the head of all action traces with no preceding trace segments. When P1 received message (1), $a_\emptyset.p$ was assigned to $(0, i_2)$ meaning that the preceding segment with its trace-id of i_2 is at P0. The same rule applied to P2 receiving message (2). When P1 received message (3), a new null head $a'_\emptyset$ was appended into the list container of Trace with its p field assigned to $(2, i_7)$, before action a_8 was linked to $a'_\emptyset$. We can also see $a'_\emptyset$ separates two action segments $a_3 \leftarrow a_4 \leftarrow a_5$ from a_8 as a delimiter at P1.

For solving the second difficulty of running the trace generation pipeline using action segments, there are two modes to follow: centralized or distributed. In the centralized mode, all distributed action segments of η are sent to the root process for sorting and concatenating the complete action trace η, which is fed to the trace generation pipeline at root. This method looks straightforward and elegant, but it is practically challenging to implement, because the elements of the action segment are pointers to local memory which are not relocatable. An action is associated with a large amount of static information about the model and may link to some dynamic information of the symbolic state. In principle it is possible to extract the concrete action information pointed to by an action pointer, send this information to the root process and reconstruct this action. But it requires a very deep understanding and insight about UPPAAL internals.

We choose the distributed mode where trace generation pipeline is run on each worker process using the local action segments in each process as input. The generated symbolic trace segments are sent to the root process for sorting and concatenating. But according to function `Trace` in Definition 20, we also need the head symbolic state for each action segment as the argument.

5.2.3 Distributed Trace Collection Procedure

The distributed trace collection algorithm (further detailed in Appendix A of Part II) works through three phases:

Phase 1 Probe which worker reached the goal state. Command that worker to start phase 2 backward propagation. Messages ASK, ACK and GO are used in phase 1.

Phase 2 Backward propagate the action segments from the goal state to the initial state using the backward reference information maintained inside $a_\emptyset.p$. Meanwhile, transfer a backward reference linking two adjacent action segments residing in two processes into the *forward*

references. When the initial null head $a_\emptyset.p = (\perp, \perp)$ preceding the action a_1 is reached at root, start phase 3 forward propagation. Message GOB is used in phase 2.

Phase 3 Forward propagate the action segments in sequence based on the forward references built in phase 2. Run the trace generation pipelines on different workers to generate symbolic trace segments using all associated action segments. Send generated symbolic trace segments to the root process for sorting and concatenating. Messages GOF and COLLECT are used in phase 3.

In phase 2 we build *reverse indexes* for the actions, because the length of the complete action trace is not known in advance. That is, for an action trace of length N, the reverse index for the last action is 0, and the reverse index for the first action is $N - 1$. For example we can mark the reverse index of each action in η in the superscript position as $\eta = a_\emptyset \leftarrow a_1^7 \leftarrow a_2^6 \leftarrow a_3^5 \leftarrow a_4^4 \leftarrow a_5^3 \leftarrow a_6^2 \leftarrow a_7^1 \leftarrow a_8^0$ as shown in Figure 10.

Forward reference is a key-value pair. The key is the reverse index of the head action in an action segment η^*. The value is a triple (`length`, `trace-id`, `rank`), where `length` is $|\eta^*|$, `trace-id` identifies η^*, and `rank` is the rank ID of the MPI process where η^* resides. Forward references are maintained in a *forward reference table* in each worker, which are consulted by the GOF message in phase 3 to find out which process has the next adjacent action trace segment. For example at P0 in Figure 10-(b), the forward reference of a local action segment $\eta_1^* = a_\emptyset \leftarrow a_1^7 \leftarrow a_2^6$ is $[7 : (2, i_2, 0)]$, where key 7 is the reverse index of a_1, the triple value has $|\eta_1^*| = 2$, $\texttt{TraceID}(\eta_1^*) = i_2$ and $\texttt{rank} = 0$. The other forward reference $[5 : (0, 0, 1)]$ refers to the next remote action segment $\eta_2^* = a_\emptyset \leftarrow a_3^5 \leftarrow a_4^4 \leftarrow a_5^3$ with the head reverse index of 5 at P1. We omit the `length` and `trace-id` (set to zero) in the forward reference for a remote action segment, because they are stored in the local forward reference inside the process identified by `rank`.

Figure 10 demonstrates how the distributed trace collection works after the termination status of Figure 9-(b). Figure 10-(a) shows the probe phase 1. (1) Root P0 broadcasts ASK to P1 and P2. (2) P1 replies ACK $\langle$true$\rangle$, because P1 reached the goal state S_8. P2 replies ACK $\langle$false$\rangle$ for it does not have a goal state. (3) P0 sends GO to P1 to start the backward phase on P1.

Figure 10-(b) shows the backward phase 2. (1) P1 extracts $a'_\emptyset \leftarrow a_8^0$ by the trace-id i_8, inserts the corresponding local forward reference $[0 : (1, i_8, 1)]$ into the forward travel table, determines from $a'_\emptyset.p = (2, i_7)$ that the preceding action segment with trace-id i_7 is at P2, then sends a message GOB $\langle 1, i_7 \rangle$ to P2 because the tail action of the preceding action segment should have the reverse index of 1 and trace-id of i_7. (2) Once P2 receives the GOB message, it firstly inserts $[0 : (0, 0, 1)]$ into the table as the forward reference for the remote adjacent action segment $a'_\emptyset \leftarrow a_8^0$ at P1. After that it extracts $a_\emptyset \leftarrow$

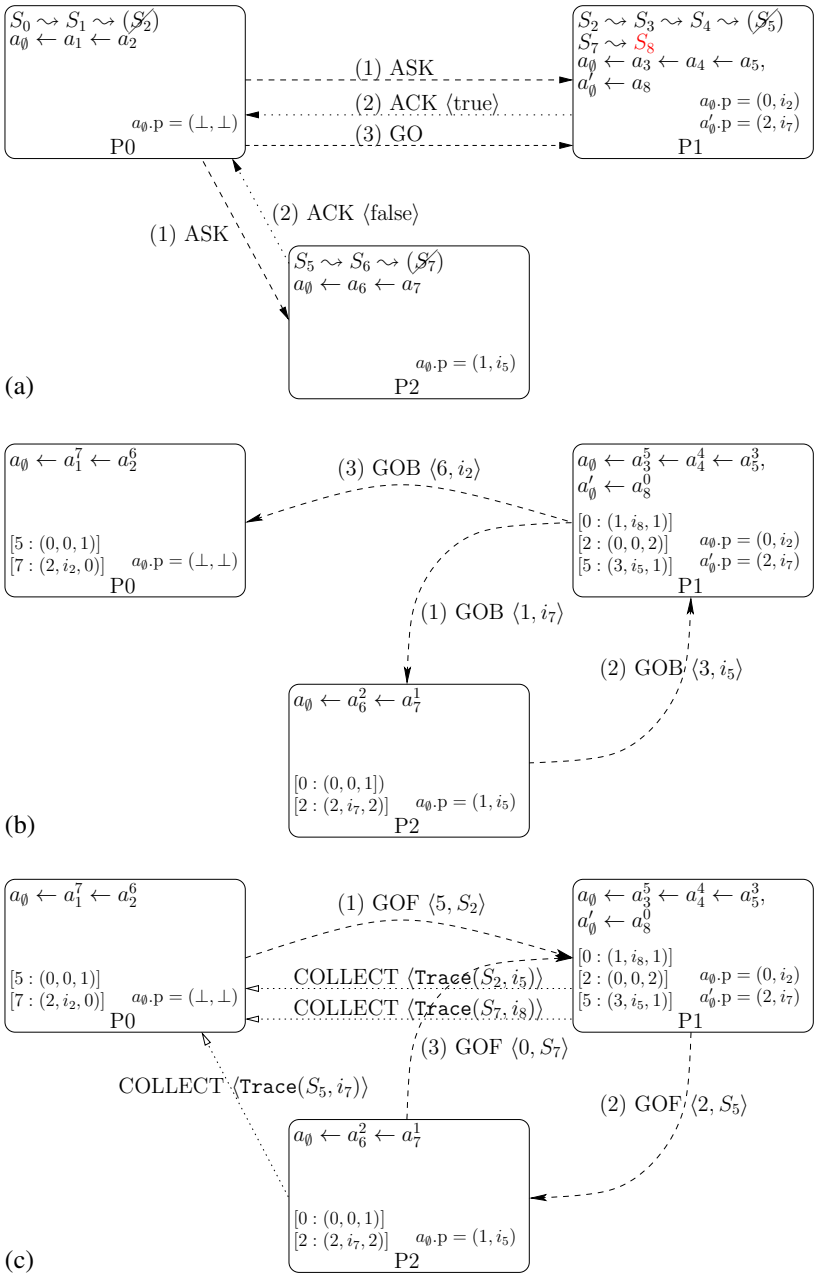

Fig. 10: Distributed Trace Collection in Three Phases: (a) Probe, (b) Backward, and (c) Forward.

$a_6^2 \leftarrow a_7^1$ by the received trace-id i_7, inserts the local forward reference $[2 : (2, i_7, 2)]$, determines from $a_\emptyset.p = (1, i_5)$ that the preceding action segment with trace-id i_5 is at P1, then sends a message GOB $\langle 3, i_5 \rangle$ to P1. (3) Once P1 receives the GOB message, it inserts the remote forward reference $[2 : (0, 0, 2)]$, extracts $a_\emptyset \leftarrow a_3^5 \leftarrow a_4^4 \leftarrow a_5^3$ by the received trace-id i_5, inserts the local forward reference $[5 : (3, i_5, 1)]$, then send a message GOB $\langle 6, i_2 \rangle$ to P0. (4) Once P0 receives the GOB message, it inserts the remote forward reference $[5 : (0, 0, 1)]$, extracts $a_\emptyset \leftarrow a_1^7 \leftarrow a_2^6$, inserts the local forward reference $[7 : (2, i_2, 0)]$, notices $a_\emptyset.p = (\perp, \perp)$ and starts the forward phase.

Figure 10-(c) shows the forward phase 3. (1) P0 finds the local forward reference $[7 : (2, i_2, 0)]$ by the reverse index 7 of a_1, generates a symbolic trace segment $\pi^* = \texttt{Trace}(S_0, i_2) =\rightsquigarrow S_1 \rightsquigarrow S_2$, assembles $\pi = S_0 \cup \pi^*$ locally, determines from the remote forward reference $[5 : (0, 0, 1)]$ that the next adjacent action segment has the head reverse index of 5 at P1, then sends a message GOF $\langle 5, S_2 \rangle$ to P1. (2) Once P1 receives the GOF message, it finds the local forward reference $[5 : (3, i_5, 1)]$ by the received reverse index 5, generates a symbolic trace segment $\pi^* = \texttt{Trace}(S_2, i_5) =\rightsquigarrow S_3 \rightsquigarrow S_4 \rightsquigarrow S_5$, sends a message COLLECT $\langle \pi^* \rangle$ to P0, then sends a message GOF $\langle 2, S_5 \rangle$ to P2 based on the remote forward reference $[2 : (0, 0, 2)]$. (3) Once P2 receives the GOF message, it finds the local forward reference $[2 : (2, i_7, 2)]$ by the received reverse index 2, generates $\pi^* = \texttt{Trace}(S_5, i_7) =\rightsquigarrow S_6 \rightsquigarrow S_7$, sends a message COLLECT $\langle \pi^* \rangle$ to P0, then sends a message GOF $\langle 0, S_7 \rangle$ to P1 based on the remote forward reference $[0 : (0, 0, 1)]$. Finally P1 receives the GOF message, it finds the local forward reference $[0 : (1, i_8, 1)]$, generates $\pi^* = \texttt{Trace}(S_7, i_8) =\rightsquigarrow S_8$, sends a message COLLECT $\langle \pi^* \rangle$ to P0, and stops the forward phase because the tail action a_8 (of reverse index 0) is reached.

A Distributed Trace Collection Algorithm

Algorithms 4 and 5 show the distributed trace collection procedure. It works through three phases: (1) probe processes having the goal state, (2) backward propagate to build forward reference tables, and (3) forward propagate to collect symbolic trace segments. Variable *FTable* is a forward reference table. It is built in the backward phase and used in the forward phase. *FTable* maps the head reverse index of an action segment η^* to a triple (`length`, `trace-id`, `rank`), where `length` is $|\eta^*|$, `trace-id` identifies η^*, and `rank` refers to the worker η^* resides. Variable *gtid* with the default value of -1 is set in the reachability phase. If an action segment leads to a goal state with the optimal cost, *gtid* is set to the trace-id of that action segment. Only the root process has the symbolic trace container π for concatenating symbolic state segments. Container π has the initial state S_0 inserted initially. Six types of messages are defined: ASK, ACK and GO for the probe phase 1; GOB for the backward phase 2; GOF and COLLECT for the forward phase 3.

The root process executes `RootMain` to initiate and manage the distributed

Algorithm 4: Distributed Trace Collection Part I

(Local Variables)
FTable $\longleftarrow \emptyset$, *gtid* (set in reachability analysis), $\pi = S_0$ ($p = Root$)

(Message Types)
ASK, ACK, GO, GOB, GOF, COLLECT

Procedure `RootMain()`

```
1  if gtid ≠ −1 then Backward(0, gtid)
2  else
3      Broadcast(ASK), Await(receive all ACK(v) responses)
4      if ∀ v = False then Stop cluster, report "Not Reachable."
5      Select one worker j with v = True, Send(GO, j)
6  Await(receive all COLLECT(π*) responses)
7  Sort and concatenate all π*s into π.
8  Stop cluster, report "Reachable" and π.
```

(Message Processing Rules)

```
9  When receive ASK from Root: Send(ACK, (gtid ≠ −1), Root)
10 When receive GO from Root: Backward(0, gtid).
11 When receive GOB(idx, tid) from worker j:
12     if idx ≠ 0 then FTable.insert((idx − 1) : (0, 0, j))
13     Backward(idx, tid).
14 When receive GOF(idx, S): Forward(idx, S).
```

trace collection procedure. Meanwhile all workers continuously handle received messages according to the message processing rules until they get terminated by root. Inside `RootMain`, root directly jumps to the backward phase if it has found a goal state and hence $gtid \neq -1$ at line 1. The probe phase 1 ranges from lines 3 to 5. It broadcasts ASK messages to ask if any worker has reached a goal state. If all ACK responses carry FALSE, it means none worker has reached a goal state. Otherwise, it selects one worker[9] whose $gtid \neq -1$, then sends a GO message to that worker to start the backward propagation. Line 6 receives all symbolic trace segments π^*s in the forward phase. Finally, it assembles all received π^*s into a complete symbolic trace π.

Let function `Extract`(tid) return the longest action segment η^* referenced by the given trace-id tid in the action list container of the Trace filter. For example, in process P1 of Figure 10-(c) in Section 5.2.3, `Extract`$(i_8) = a'_\emptyset \leftarrow a^0_8$, and `Extract`$(i_5) = a_\emptyset \leftarrow a^5_3 \leftarrow a^4_4 \leftarrow a^3_5$. Let function `Tail`$(\pi^*)$ return the tail symbolic state of a symbolic trace segment π^*.

In the backward phase, the GOB protocol backward propagates the adjacent action segments connecting the goal state to the initial state. The backward reference information left in $a_\emptyset.p$ during the reachability analysis plays

Algorithm 5: Distributed Trace Collection Part II

```
   Procedure Backward(idx, tid)
1  |  η* = Extract(tid), a_∅ ⟵ η*[0], len ⟵ |η*|
2  |  if len > 0 then
3  |  |  idx ⟵ idx + len − 1
4  |  |  FTable.insert(idx : (len, tid, p)), tid ⟵ tid + 1
5  |  if a_∅.p ≠ (⊥, ⊥) then
6  |  |  tid ⟵ a_∅.p.trace-id, src ⟵ a_∅.p.source
7  |  |  Send(GOB, idx, tid, src)
8  |  else Forward(idx − 1, S_0)
   Procedure Forward(idx, S)
9  |  (len, tid, pid) ⟵ FTable.find(idx), π* ⟵ Trace(S, tid)
10 |  if p = Root then Sort and concatenate π* into π.
11 |  else Send(COLLECT, π*, Root)
12 |  idx ⟵ idx − len + 1
13 |  if idx > 0 then
14 |  |  idx ⟵ idx − 1, S = Tail(π*)
15 |  |  (len, tid, pid) ⟵ FTable.find(idx)
16 |  |  Send(GOF, idx, S, pid)
```

[9]It is possible that more than one worker reach the goal state. Root will arbitrarily choose one such worker to contact so that only one symbolic trace in generated.

an important role here. The product of this phase is a forward reference table containing the guiding information for the forward phase. Procedure `Backward` takes two parameters: the reverse index *idx* and the trace-id *tid* of an action segment η^*. It is called for the first time at either line 1 or line 10 in Algorithm 4. Inside `Backward`, after the target action segment η^* is extracted given *tid* from the Trace filter, the null head and length of η^* are obtained at line 1. Line 3 computes the head reverse index of η^*. Line 4 insert into the forward reference table a local forward reference corresponding to η^*. Then *tid* is self-increased by one to match the tail action of the preceding action segment at a remote worker. If the initial null head $a_\emptyset.p = (\perp, \perp)$ is not found in η^* at line 5, the backward phase should continue. The backward reference is obtained from $a_\emptyset.p$, and a GOB message is sent to the worker which has the preceding action segment. Otherwise backward propagation stops and the forward phase 3 starts at root at line 8. Note that *tid* has already been one beyond the reverse index of a_0, thus it should minus one. Separately, line 12 in Algorithm 4 inserts a remote forward reference for the adjacent action segment at the source process j, from which a worker receives a GOB message. The head reverse index of the next adjacent action segment is $idx - 1$, because the received *idx* was already self-increased by one at line 4 in `Backward` at process j.

In the forward phase, the GOF protocol forward propagates action segments guided by the local and remote forward references. It invokes the local trace generation pipeline on each worker to generate symbolic trace segments, which are sent to root for assembling the complete symbolic trace. Procedure `Forward` takes two parameters: the head reverse index of an action segment η^*, and the head symbolic state that fires η^*. When `Forward` is called for the first time at line 8 in `Backward`, *tid* is the reverse index of a_0, and $S = S_0$. Line 9 obtains the triple referencing the local action segment η^* by *tid*. The symbolic trace segment π^* corresponding to η^* is generated by the local trace generation pipeline (denoted by `Trace` as in Definition 20 of Section 5.2.1). Root will assemble its local π^* into π locally. Other workers send π^*s by the COLLECT messages to root. Line 12 computes the tail reverse index of η^* for checking if the tail action (with $idx = 0$) of the complete action trace is reached at line 13. Otherwise, the forward phase should continue. Line 14 computes the head reverse index of the following adjacent action segment resides in a remote worker. The corresponding head symbolic state is obtained at the tail of π^*. The rank ID of the remote worker is obtained by looking up the remote forward reference by the newly calculated *idx* at line 15. Finally a GOF message is sent to the target remote worker.

References

[Bar04] Jiri Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Masaryk University Brno, Faculty of Informatics, 2004.

[CS95] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Tel01] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

[VBBB09] Kees Verstoep, Henri E. Bal, Jiri Barnat, and Lubos Brim. Efficient large-scale model checking. In *IPDPS*, pages 1–12. IEEE, 2009.

[ZNL16] Zhengkui Zhang, Brian Nielsen, and Kim Guldstrand Larsen. Distributed algorithms for time optimal reachability analysis. In Martin Fränzle and Nicolas Markey, editors, *Formal Modeling and Analysis of Timed Systems - 14th International Conference, FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings*, volume 9884 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2016.

Part III

Papers

Paper A

Time Optimal Reachability using Swarm Verification

Zhengkui Zhang, Brian Nielen, and Kim Guldstand Larsen

The paper has been published in the
Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC),
pp. 1634–1640, 2016.

The layout has been revised.

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.*

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 [1, 2]. A diagnostic trace to a goal state offered by real-time model checkers such as Uppaal [3] and Kronos [4] 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 [5].

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 [6]. Behrmann *et al.* presented a branch-and-bound minimal-cost reachability algorithm on priced timed automata (PTA) in [7, 8]. Another way to restrict and guide the state-space exploration is adding constrains on transitions [9].

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 [10, 11], DiVinE [12], LTSmin [13], 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 al.* in [15]. 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 [8]. 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 [10], 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: $g, g_1, g_2 ::= x \bowtie n \mid x - y \bowtie n \mid g_1 \wedge g_2$, where $x, y \in X$ are clocks, $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

Definition 1. *A* Timed Automaton (TA) [16] *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: $Q = \{(\ell, v) \mid (\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ 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 $(\ell \xrightarrow{g,a,r} \ell') \in E$ 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* $\texttt{MinSpan}(\ell, v)$ *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* $\texttt{MinSpan}(\ell)$ *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* [17], which is normally efficiently represented and stored in memory as *difference bound matrices* (DBM) [18].

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).*

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.

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
```

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`.

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

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.

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 to 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 $\longleftarrow \{(\ell^*, Z^*)\}$.

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 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 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 to find the time optimal schedule for all six people to finish reading.

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

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 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 A.1 to A.3 show for the first three models the median runtime to reach optimal cost (t_{opt} corresponding to metric 1), and runtime to prove

Table A.1: Runtime (sec) of Job-Shop-6

#C	BFS		DFS		RDFS			
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}		
1	108	108	7877	9899	650	2283		
#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	535	1996	459	1821	102	102	39	99
4	310	1730	280	1683	102	102	25	98
8	176	1720	116	1653	101	102	17	97
16	100	1591	87	1506	99	101	14	97
32	65	1488	52	1494	64	99	13	96
64	32	1452	24	1385	24	98	16	95
128	4	1412	4	1367	4	98	29	95
256	2	1365	<1	1341	<1	98	28	95
512	<1	1337	<1	1263	<1	100	31	96

Table A.2: Runtime (sec) of Aircraft-Landing-15

#C	BFS		DFS		RDFS			
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}		
1	157	159	73	428	191	964		
#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	87	845	93	873	91	91	66	90
4	42	737	35	729	46	90	21	91
8	22	712	16	727	17	89	25	88
16	17	676	11	699	11	89	6	86
32	2	638	1	633	2	89	12	86
64	<1	604	<1	582	<1	88	5	85
128	<1	597	<1	620	<1	88	2	84
256	<1	581	<0.1	565	<0.1	88	7	84
512	<1	585	<0.1	573	<0.1	117	5	86

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.

Table A.3: 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.

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 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 A.1 and A.2 show how near optimal results improve with runtime for models job-shop-6 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 im-

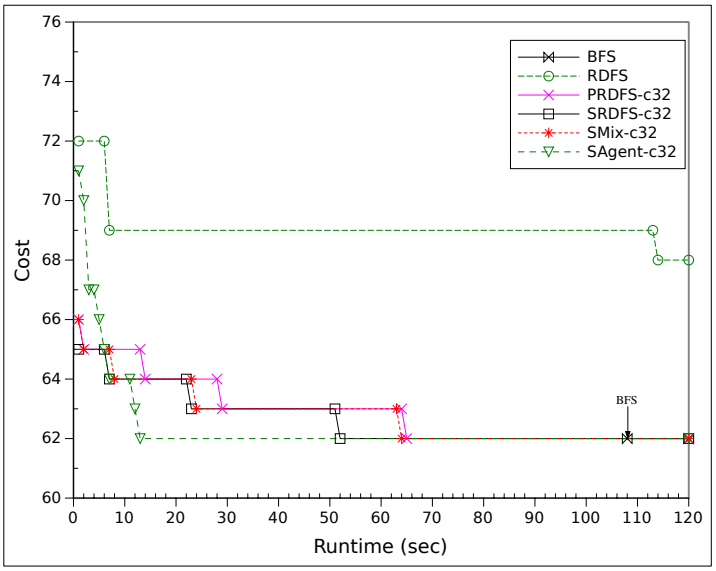

Fig. A.1: Cost vs. Runtime for Job-Shop-6

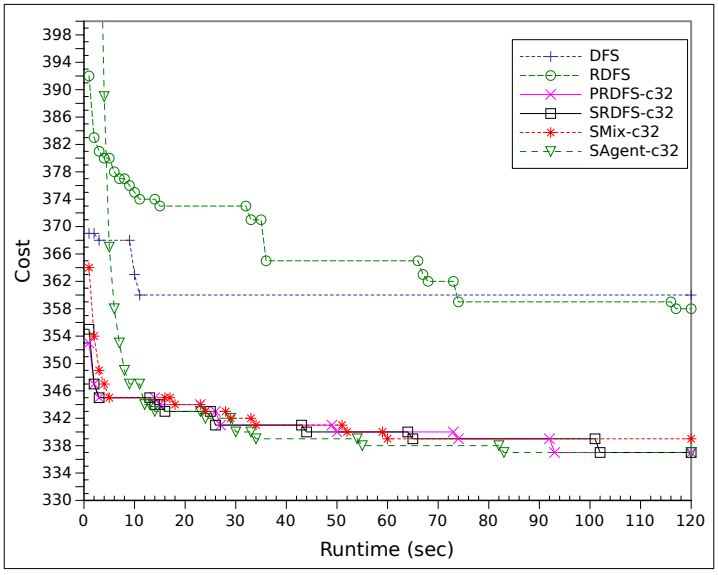

Fig. A.2: Cost vs. Runtime for Task-Graph-88

proved 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 A.4 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.

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 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

Table A.4: 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.

more.

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

5 Conclusion

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.

Acknowledgments

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

References

[1] Y. Abdeddaïm, E. Asarin, and O. Maler, "Scheduling with timed automata," *Theoretical Computer Science*, vol. 354, no. 2, pp. 272–300, 2006.

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

[3] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *SFM*, ser. LNCS, vol. 3185. Springer, 2004, pp. 200–236.

[4] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *CAV*, ser. LNCS, vol. 1427. Springer, 1998, pp. 546–550.

[5] A. Fehnker, "Scheduling a steel plant with timed automata," in *RTCSA*. IEEE Computer Society, 1999, pp. 280–286.

[6] ——, *Bounding and Heuristics in Forward Reachability Algorithms*. UB Nijmegen [Host], 2000. [Online]. Available: http://books.google.dk/books?id=7cEPtwAACAAJ

[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*, ser. LNCS, vol. 2034. Springer, 2001, pp. 147–161.

[8] G. Behrmann and A. Fehnker, "Efficient guiding towards cost-optimality in uppaal," in *TACAS*, ser. LNCS, vol. 2031. Springer, 2001, pp. 174–188.

[9] T. Hune, K. G. Larsen, and P. Pettersson, "Guided synthesis of control programs using uppaal," in *ICDCS Workshop*, 2000, pp. E15–E22.

[10] G. Behrmann, T. Hune, and F. W. Vaandrager, "Distributing timed model checking - how the search order matters," in *CAV*, ser. LNCS, vol. 1855. Springer, 2000, pp. 216–231.

[11] G. Behrmann, "Distributed reachability analysis in timed automata," *STTT*, vol. 7, no. 1, pp. 19–30, 2005.

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

[13] S. Blom, J. van de Pol, and M. Weber, "Ltsmin: Distributed and symbolic reachability," in *CAV*, ser. LNCS, vol. 6174. Springer, 2010, pp. 354–359.

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

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

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

[17] K. G. Larsen, P. Pettersson, and W. Yi, "Model-checking for real-time systems," in *FCT*, ser. LNCS, vol. 965. Springer, 1995, pp. 62–88.

[18] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets*, ser. LNCS, vol. 3098. Springer, 2003, pp. 87–124.

Paper B

Distributed Algorithms for Time Optimal Reachability Analysis

Zhengkui Zhang, Brian Nielsen, and Kim Guldstand Larsen

The paper has been published in the
Proceedings of the 14th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS), Vol. 9884, pp. 157–173, 2016.

The layout has been revised.

Abstract

Time optimal reachability analysis is a novel model based technique for solving scheduling and planning problems. After modeling them as reachability problems using timed automata, a real-time model checker can compute the fastest trace to the goal states which constitutes a time optimal schedule. We propose distributed computing to accelerate time optimal reachability analysis. We develop five distributed state exploration algorithms, implement them in Uppaal *enabling it to exploit the compute resources of a dedicated model-checking cluster. We experimentally evaluate the implemented algorithms with four models in terms of their ability to compute near- or proven-optimal solutions, their scalability, time and memory consumption and communication overhead. Our results show that distributed algorithms work much faster than sequential algorithms and have good speedup in general.*

1 Introduction

Time optimal reachability (TOR) analysis is a novel model based technique for solving scheduling and planning problems [1, 2]. After modeling these problems using timed automata, a real-time model checker such as Uppaal [3] and Kronos [4] can compute the fastest trace to the goal states which constitutes a time optimal schedule, because the trace carries actions of the model and timing information of these actions to the goal states. TOR allows natural modeling of real-time behavior, constrains and interactions of components, as well as flexible choices of efficiently implemented search algorithms inside model checkers.

However the well-known state-space explosion problem may arise when the number of components in the model is large. In [5] we developed swarm algorithms to mitigate this problem and accelerate TOR. The core idea is employing a large number of parallel Uppaal instances or agents with randomized search strategies to search the state spaces independently, thus finding different traces to the goal state in parallel and avoiding local optimality. The advantages of this approach are: (1) easy to implement; (2) find optimal (or near optimal) results fast without exploring the full state-space. A weak point however is the lack of data parallelism nor sharing of the explored state-space, thus the execution time is hardly reduced and memory is limited to that of a single instance.

In this paper, we extend our previous work in [5] by developing distributed algorithms that may accelerate TOR in three ways. First, the state-space is now partitioned and distributed among distributed CPU and memory resources so that multiple worker processes now share the state-space exploration workload, thus the execution time may potentially be shortened greatly. Second, more traces/state-space will be explored in parallel, thus the

fastest trace will be potentially found even faster than the swarm algorithms. Third, the disjoint parts of state-space are stored in the distributed memory, allowing fully use the memory of a cluster and handle even larger models than swarm algorithms.

Related Work. Because time optimal reachability involves the notion of cost and optimality, *branch and bound* (B&B) is used for efficient state-space exploration. By a bounding function and the current best solution to the goal states, B&B can effectively prune parts of the state-space that guarantee not to lead to an optimal solution [6] thus avoiding enumerating the entire state-space. Behrmann *et al.* presented a branch-and-bound minimal-cost reachability algorithm on the priced timed automata (PTA) in [7, 8].

The earliest and monumental distributed model checker is the parallel Murφ verifier proposed in 1997 [9]. Its design delineated the cornerstone upon which other distributed model checkers were built thereafter such as distributed Uppaal [10, 11], DiVinE [12], LTSmin [13], etc. Meanwhile, enormous research efforts have been made to improve the state-space generation algorithm, the partition algorithm, the state storage data structure, the communication and control mechanism, as well as many other technical issues. Since 2006 multi-core CPUs became pervasive inside PC, HPC and embedded markets, DiVinE, LTSmin and Fdr3 [14] exploit multi-core shared memory technique to achieve even better performance on the modern hardware architecture. DiVinE also made fruitful attempts to accelerate model checking using GPUs from 2009 [15].

Contribution. We developed five distributed TOR algorithms and implemented them in the Uppaal model checker to accelerate TOR analysis. In addition to sharing explored states, worker processes exchange computed better costs to the goal states. This enables each worker to prune its local state-space efficiently by B&B, hence need less execution time and memory consumption.

D-BFS: distributed breath-first search. Each worker runs local BFS while exchanging states with other workers.

D-BFS^{S}: distributed strict order BFS (also named *level synchronized* BFS). A synchronization protocol will ensure all workers completely explore states on the same current BFS level before moving on to the next level.

D-DFS: distributed depth-first search. Same principle as D-BFS except traversing depth-first.

D-DFS^{G}: distributed greedy DFS. In addition to D-DFS, each worker always picks the successor state of the lowest cost in each iteration.

D-RDFS: distributed random depth-first search. Same principle as D-DFS with a randomly picked successor state.

It is worth noting that in distributed BFS/DFS/DFS^{G}/RDFS, their global search orders only approximate BFS/DFS/DFS^{G}/RDFS. Due to the varying communication delay or workload on computing nodes, states are received in

nondeterministic order from run to run. This influences the successor states generation locally and changes the number of states explored [10]. Another important observation is that in UPPAAL BFS often completes explorations 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. The motivation of D-BFS^S is to keep this strong point of BFS. However 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.

We employ the following metrics to compare the algorithms:

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

Metric 2: time to completely explore the state-space and terminate thus proving the optimal result (t_{prov}). Users prefer an algorithm to terminate fast.

Metric 3: time to progressively improved solutions (a.k.a. *near optimal* solutions). It shows how fast the results converge to the optimal as a function of running time. 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 that are closer to the optimality.

Metric 4: memory consumption and communication overhead of algorithms. Smaller memory consumption improves scalability by allowing bigger state-space. Smaller communication overhead improves computing speed.

Outline. The rest of the paper is organized as follows. Section 2 recalls the definitions of timed automata and sequential TOR algorithm. Section 3 explains the distributed TOR algorithms. Section 4 shows benchmark experiment results of the sequential and distributed TOR algorithms. Section **??** concludes.

2 Sequential Time Optimal Reachability

This section recalls timed automaton and the sequential time-optimal reachability algorithm. For brevity parallel composition of timed automata is omitted.

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: $g, g_1, g_2 ::= x \bowtie n \mid x - y \bowtie n \mid g_1 \wedge g_2$, where $x, y \in X$ are clocks, $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

Definition 1. *A* Timed Automaton (TA) [16] *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, each of which contains a source location, a guard, an action, a set of clocks to be reset and a target location;* $Inv : L \to \mathcal{B}(X)$ *sets an invariant for each location. For simplicity an edge* $(\ell, g, a, r, \ell') \in E$ *is written as* $\ell \xrightarrow{g,a,r} \ell'$.

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:* $Q = \{(\ell, v) \mid (\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ *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:*
(1) $(\ell, v) \xrightarrow{a} (\ell', v')$ *if there is an edge* $(\ell \xrightarrow{g,a,r} \ell') \in E$ *such that* $v \models g$, $v' = v[r \mapsto 0]$ *and* $v' \models Inv(\ell')$;
(2) $(\ell, v) \xrightarrow{d} (\ell, v + d)$ *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 *(or* q'*) is* reachable *if there exists a finite trace with the final state of* q *(or* 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* $\texttt{MinSpan}(\ell, v)$ *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* $\texttt{MinSpan}(\ell)$ *is the infimum of spans of finite traces ending in* (ℓ, v) *for all possible* v.

2.2 Sequential Time Optimal Reachability Algorithm

The real-time model checker 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* [17], which is normally efficiently represented and stored in memory as *difference bound matrices* (DBM) [18]. Besides, we denote the action and delay transitions between symbolic states uniformly as $\rightsquigarrow$.

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

Algorithm 1: Sequential Time Optimal Reachability

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

UPPAAL keeps track of the trace span by including an implicit clock ψ in addition to the original set of clocks X of the model. Clock ψ drifts as the global elapsing time and remains unaffected from resets or guards or invariants in the model. Thus the zone Z is now over $X \cup \{\psi\}$; and $\texttt{MinCost}(\ell, Z)$ is calculated on-the-fly by evaluating the lower bound of ψ in Z.

Algorithm 1 shows the sequential TOR algorithm that computes the minimum span to reach the goal states satisfying the proposition `Goal`. WAITING and PASSED keep unexplored and explored symbolic states respectively; and WAITING has the initial state. COST maintains the current best result that is infinity initially. Inside procedure `Main`, whenever WAITING is not empty, an unexplored state is popped from WAITING in a loop. If the state is a goal state, COST is updated. This implies a near optimal schedule to the goal is found. If the state is not goal state, it is subject to symbolic state inclusion checking and B&B elimination rule at line 6. A symbolic state (ℓ, Z) is included in PASSED and discarded if $\exists(\ell, H) \in$ PASSED s.t. $Z \subseteq H$ [17, 18]. That is, a previously explored state with the same location has an equal or larger zone than the current state. The same state is pruned if its cost function is no less than the current COST. If the state gets through the two tests on line 6, it is added to PASSED as already explored, and then its successor states are generated and added to WAITING.

3 Distributed Time Optimal Reachability

This section describes the distributed time optimal reachability algorithms that extends the sequential version with state-space partitioning and message

passing. Each algorithm is uniform and executed by all worker processes in a cluster.

3.1 Distributed Algorithm

Algorithm 2 shows the distributed algorithms: D-BFS, D-DFS, D-DFS^{G} and D-RDFS. A key activity of this algorithm is partitioning and distributing the state-space. A partition function uniquely computes the ID of a process that a symbolic state belongs to, hence divides the entire state-space into disjoint subsets on all processes. We use a hash function to partition; and the hash value is only calculated on the location ℓ[1]. The reason is that the inclusion checking of a symbolic state requires looking up all states with the same location in PASSED. Therefore, in distributed settings all states of the same location will destine to the same process for deterministic and easy inclusion checking. The other key work of this algorithm is message passing and handling. We define two messages: (1) UPDATE carries better costs; (2) STATE carries symbolic states.

Definition 6. *Let N be the set of worker processes and p denote the local process ID. The* partition function *is a total mapping:* `Hash` $: L \rightarrow N$ *from the set of locations to the set of processes. Process p is the* owner *of a symbolic state (ℓ, Z) if* `Hash`$(\ell) = p$*. A symbolic state is a* local *state if it is generated on its owner process, otherwise it is an* emigrant *state.*

Definition 7. *A process is* active *when doing local search or receiving messages. Initially all processes are active. A process is* idle *when its waiting list is empty and receiving no messages. Computation can* terminate *if all processes are idle and the network has no message in transit.*

Three new variables are used: ECOST maintains external better cost received from the network, ACTIVE specifies process status (active or idle), and TERMINATE controls the `Main` loop. At the beginning of each iteration, `Ecost` compares with `Cost`. If `Ecost` is smaller, a better cost has been found by another process, and `Cost` is updated. If `Cost` is smaller, the current process finds a better cost, which is assigned to `Ecost` and then broadcasted. Line 5 marks process to idle according to Definition 7. The remaining lines inside `Main` resembles the local search in the sequential algorithm, except at line 14 a successor state is hashed to compute its owner process ID and sent out if it does not belong to the current process p. When the root process becomes idle, it invokes the termination detection procedure `CheckTerm` based on the well-known token-based Safra protocol [19]. Line 20 updates `Ecost` on reception of a cost message. Line 21 adds a received emigrant state into WAITING.

[1]For real UPPAAL models, the location is a vector of locations from each parallelly composed timed automata and the values of all discrete variables in the model.

Algorithm 2: Distributed Time Optimal Reachability for D-BFS/D-DFS/D-DFS^{G}/D-RDFS

(Local Variables)

$\textsc{Waiting} \longleftarrow \begin{cases} \{(\ell_0, Z_0)\} & \text{if } p = \texttt{Hash}(\ell_0), \\ \emptyset & \text{otherwise.} \end{cases}$ $\quad \textsc{Passed} \longleftarrow \emptyset$

$\textsc{Terminate} \longleftarrow \textsc{False}, \textsc{Active} \longleftarrow \textsc{True}, \textsc{Cost} \longleftarrow \infty, \textsc{Ecost} \longleftarrow \infty$

(Message Types)

UPDATE, STATE

Procedure `Main()`

```
1  while ¬Terminate do
2      if Ecost < Cost then Cost ⟵ Ecost
3      if Cost < Ecost then UpdateE(Cost),
       Broadcast(UPDATE, Cost)
4      if Waiting = ∅ then
5          if receive no message then Active ⟵ False, CheckTerm()
6          continue
7      select (ℓ, Z) from Waiting, Active ⟵ True
8      if (ℓ, Z) ⊨ Goal then
9          if MinCost(ℓ, Z) < Cost then
10             Cost ⟵ MinCost(ℓ, Z)
11     else if (ℓ, Z) ∉ Passed and MinCost(ℓ, Z) < Cost then
12         add (ℓ, Z) to Passed
13         forall the (m, D) such that (ℓ, Z) ⇝ (m, D) do
14             if (r ⟵ Hash(m)) ≠ p then Send(STATE, (m, D), r)
15             else add (m, D) to Waiting
16 return Cost
```

Procedure `UpdateE(NewCost)`

```
17 if NewCost < Ecost then Ecost ⟵ NewCost
```

Procedure `CheckTerm() // Safra protocol [19]`

```
18 if p = root and all processes are idle and no message in transit then
19     Terminate ⟵ True on all processes
```

(Message Processing Rules)

20 When receive UPDATE⟨Ncost⟩: `UpdateE(Ncost)`, $\textsc{Active} \longleftarrow \textsc{True}$.

21 When receive STATE⟨(n, F)⟩: add (n, F) to $\textsc{Waiting}$, $\textsc{Active} \longleftarrow \textsc{True}$.

The computation starts at the process p that owns the initial state $s_0 = (\ell_0, Z_0)$ determined by `Hash`. Successor states s'_i are generated by local search

on s_0, meanwhile `Hash` computes the owner process ID of s'_i. If s'_i belongs to a different process r than p, it is sent to process r. Otherwise it is stored locally for future exploration. Once process r receives a state, it stores it in its WAITING and eventually starts to generate successors from it; and the partition function works in a similar fashion. Gradually all processes start to work. Finally the entire state-space is generated, and no unexplored successor states could be found. When all processes become idle and no message is in transit, the computation can stop.

3.2 Distributed Algorithm for Strict BFS

Algorithm 3 highlights the changes on Algorithm 2 to make a distributed strict BFS. NLQ stands for the next level queue that collects the states on the next BFS level. WAITING keeps states on the current BFS level. Lines 2 to 5 explore states on one level and generate successor states for the next level, which are either sent out or stored in NLQ. The emigrant states received from the network are also stored in NLQ at line 13. After exhausting the states on the current level in WAITING, all processes synchronize on the condition that each process has completely harvested all STATE messages from the network into NLQ. After line 8 WAITING contains the states for the next BFS level and NLQ is empty.

Algorithm 3: Distributed Time Optimal Reachability for D-BFS^S

NLQ $\longleftarrow \emptyset$

Procedure `Main()`

```
1  while ¬TERMINATE do
2      while WAITING ≠ ∅ do
           same code as lines 2 to 3 and 7 to 12 in Algorithm 2
3          forall the (m, D) such that (ℓ, Z) ⇝ (m, D) do
4              if (r ⟵ Hash(m)) ≠ p then Send(STATE, (m, D), r)
5              else add (m, D) to NLQ
6      Await(Synchronize(receive all STATE messages) or TERMINATE)
7      if TERMINATE then break
8      Swap(WAITING, NLQ)
9      if WAITING = ∅ and receive no message then
10         ACTIVE ⟵ FALSE, CheckTerm()
11 if ECOST < COST then COST ⟵ ECOST
12 return COST
```

13 When receive STATE$\langle(n, F)\rangle$: add (n, F) to NLQ, ACTIVE $\longleftarrow$ TRUE.

4 Experiments

We developed a new version of UPPAAL implementing the distributed algorithms. The implementation involves three key tasks: (1) build a communication module by the MPI library; (2) interact with UPPAAL's internal memory management to fetch send-out states or insert received states; (3) implement the distributed search orders with the support from (1) and (2). We applied several optimization techniques [20] to improve communication such as: asynchronous communication, buffering states and sending them in packets, and packet compression.

We ran benchmark experiments in a cluster with 9 computing nodes. Each node has 1 Tb memory (NUMA architecture) and 64 cores at the frequency of 2.3 GHz (4 AMD Opteron 6376 Processors each with 16 cores), a 1 TB SATA disk and the Infiniband interconnection. All five distributed algorithms were executed 10 runs for every core setting: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512. These core settings follow an even mapping topology as the (nodes, cores-per-node) pairs: $1 \rightarrow (1,1), 2 \rightarrow (2,1), 4 \rightarrow (4,1), 8 \rightarrow (4,2), 16 \rightarrow (4,4), 32 \rightarrow (4,8), 64 \rightarrow (4,16), 128 \rightarrow (4,32), 256 \rightarrow (4,64), 512 \rightarrow (8,64)$. For instance, 2 cores are mapped to 2 nodes with 1 core on each node; 32 cores are mapped to 4 nodes with 8 cores on each node.

4.1 Models

We use the same models as in [5]. The first three models are up-scaled versions of those in normal UPPAAL distribution by adding more parallel components. The last model is transformed from an industrial task graph benchmark[2].

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 to 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 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

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

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 heterogeneous processors at the speeds of [1,1,2,4]. The objective is to compute 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 B.1 to B.4 show for models Job-Shop-6, Job-Shop-8 and Viking-Bridge-15 the median runtime to reach optimal cost (t_{opt} corresponding to metric 1), and the median runtime to prove optimal cost (t_{prov} corresponding to metric 2). We want to know how the distributed algorithms scale with an increasing number of cores denoted by #C (#C=1 for sequential algorithms). We set the 4-hour time bound for the experiments; and "-" indicates timeout.

Job-Shop-6. For the sequential algorithms, DFS^{G} has the best t_{opt} and BFS has the best t_{prov}. DFS is the slowest for both t_{opt} and t_{prov}.

For the distributed algorithms, D-BFS runs slower than BFS at lower core settings (2 to 8), and a speed-up requires more cores. D-BFS^{S} is much more

Table B.1: Runtime (sec) of Job-Shop-6

#C	BFS				DFS		DFS^{G}		RDFS	
	t_{opt}	t_{prov}			t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}
1	100	100			7411	9309	6	175	616	2125
#C	D-BFS		D-BFS^{S}		D-DFS		D-DFS^{G}		D-RDFS	
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}
2	246	246	153	153	467	1398	301	842	532	1505
4	346	390	88	88	53	419	15	311	23	401
8	98	129	33	33	1	199	1	158	2	187
16	58	73	23	23	1	116	1	88	1	112
32	28	38	16	16	1	62	1	63	1	60
64	18	31	14	14	1	34	1	32	1	34
128	12	15	16	16	1	17	1	21	1	17
256	8	11	38	39	2	11	1	9	2	11
512	6	10	70	72	2	17	1	10	2	19

competitive. We note that UPPAAL is highly optimized for single core execution, and therefore requires several cores to offset initialization and message passing overhead. Further, the *core mapping topology* leads to high communication latency especially at 2 and 4 cores where each core is mapped to one computing node using an (infiniband) network that is relatively slower than the internal bus within a node. The intensive state exchange on the single channel between 2 cores will also incur a *communication channel congestion*. The congestion fades when using more cores. If we map 2 and 4 cores on the same computing node as shown in table B.2, t_{prov} is improved by 40% for D-BFS^S, and 15% for D-BFS.

Table B.2: Runtime (sec) for Cores on Same Node

#C	D-BFS		D-BFS^S	
	t_{opt}	t_{prov}	t_{opt}	t_{prov}
2	195	195	95	95
4	340	352	53	53
8	107	118	29	29

The difference between BFS (D-BFS) and D-BFS^S is caused by the symbolic states. BFS is good at building larger zones (see Section 1). But since D-BFS only approximates BFS, it causes more *fragmentation* at 2 and 4 cores where workers consequently generate and communicate more symbolic states. Using more cores from 16 to 512 cores, D-BFS steadily reduces execution time for t_{opt} and t_{prov}. D-BFS^S runs faster than D-BFS at core settings (2 to 64). Above 128 cores t_{opt} and t_{prov} slow down indicating higher level synchronization overhead.

Comparing the three remaining depth-first based distributed algorithms, they all have a noticeable t_{opt} of merely 1 or 2 seconds above 8 cores. D-DFS^G seems overall to be a good choice, despite sligtly less speedup.

Job-Shop-8. This is an enlarged version of Job-Shop-6. This experiment shows that when all sequential algorithms confront timeout, the distributed algorithms can terminate normally and prove the optimal result. D-BFS^S can already terminates at 4 cores while other algorithms need more than 16 cores. It also has the best t_{prov}; and shows a linear speedup from 4 to 128 cores. D-DFS^G has extremely good t_{opt}.

Aircraft-Landing-15 All distributed algorithms show similar pattern as Job-Shop-6. A difference is that D-BFS has much faster t_{opt} above 8 cores and somewhat faster t_{prov} than D-BFS^S at higher core settings (in appendix A).

Viking-Bridge-15. This model is good for BFS, but extremely bad for DFS/RDFS that suffer timeout. We observed from the log files that this model produces solutions with a wide cost spectrum from 638195 to 220 and decrement step of just one. This explains why DFS^G is also 30 times slower than BFS because it gets trapped by the fine grained local optimal (as depicted in fig B.2).

No distributed algorithms could beat BFS without using enough many cores. D-BFS and D-BFS^S show pool speedup. The good news is that D-DFS

Table B.3: Runtime (sec) of Job-Shop-8

#C	BFS				DFS^{G}	
	t_{opt}	t_{prov}			t_{opt}	t_{prov}
1	-	-			849	-
#C	D-BFS		D-BFS^{S}		D-DFS^{G}	
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}
2	-	-	-	-	6249	-
4	-	-	9295	9295	366	-
8	-	-	4900	4900	1	-
16	8881	13050	2568	2568	1	-
32	5390	7276	1315	1315	1	13293
64	3157	4623	693	663	2	7956
128	2052	2470	450	450	2	4406
256	1335	1495	260	260	4	2547
512	1012	1231	365	365	11	1773

"-": denotes 4-hour timeout.

Table B.4: Runtime (sec) of Viking-Bridge-15

#C	BFS			DFS		DFS^{G}		RDFS	
	t_{opt}	t_{prov}		t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}
1	66	66		-	-	2016	2146	-	-
#C	D-BFS		D-BFS^{S}	D-DFS		D-DFS^{G}		D-RDFS	
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}
2	533	559	125	125	-	-	-	-	-	-
4	308	314	106	106	956	1058	-	-	868	980
8	400	430	70	70	439	515	181	267	458	520
16	165	196	41	41	216	280	71	115	173	246
32	115	120	31	31	116	183	28	69	82	132
64	64	69	23	23	40	63	14	32	41	69
128	32	36	61	61	20	49	8	24	17	34
256	50	55	113	114	15	24	6	11	14	23
512	21	25	165	168	16	34	13	18	17	36

"-": denotes 4-hour timeout.

and D-RDFS can complete in half an hour above 4 cores and present linear speedup in t_{opt} and t_{prove} until 128 cores. D-DFS^{G} confronts heavy fragmentation and timeout at 2 and 4 cores. But using enough many workers avoids local optimal, and D-DFS^{G} gains the best t_{opt} and t_{prov} among all algorithms.

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. But we show how the near optimal results converge according to runtime in section 4.3.

Conclusions. (1) D-DFS^{G} can find the optimal result very fast in general followed by D-DFS/D-RDFS. (2) D-DFS/D-RDFS have good speedup on the time to prove. (3) For larger models, distributed algorithms provide results while the sequential algorithms cannot, with D-BFS^{S} in many cases being the fastest to terminate. (4) D-BFS/D-DFS^{G} may be slower than the optimized sequential BFS/DFS^{G} at low core settings due to fragmentation. (5) The exact performance may depend on the characteristics of the model.

4.3 Results versus Time (Metric 3)

Figures B.1 to B.3 show how near optimal results improve with running time for Job-Shop-6, Viking-Bridge-15 and Task-Graph-88. The sample window is 2 minutes. For the distributed algorithms, we look at the intermediate core setting of 32.

Job-Shop-6. For the sequential algorithms, DFS^{G} is the fastest in finding the optimal result of 62 at 6 sec. BFS reports 62 at 100 sec. DFS reports no results within the sample window. RDFS only reports near optimal results from 72 to 68. For the distributed algorithms, D-DFS/D-DFS^{G}/D-RDFS reach the optimal result immediately at 1 sec. D-BFS/D-BFS^{S} reach the optimal result at 28 sec and 16 sec respectively. Compared with the best of our swarm

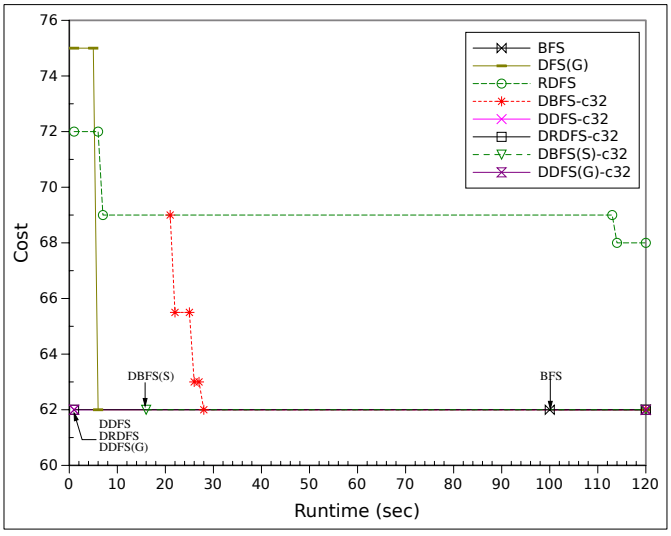

Fig. B.1: Cost vs. Runtime for Job-Shop-6

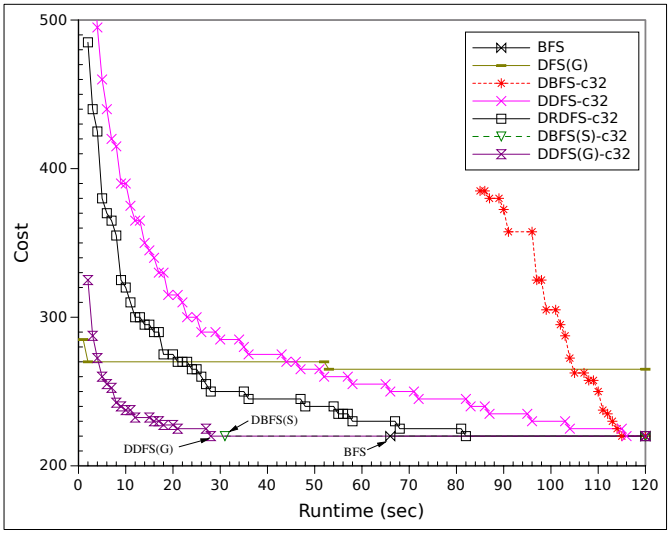

Fig. B.2: Cost vs. Runtime for Viking-Bridge-15

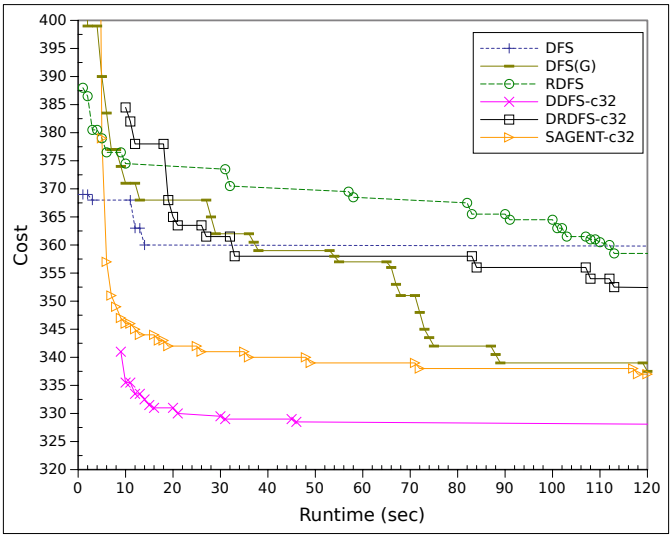

Fig. B.3: Cost vs. Runtime for Task-Graph-88

algorithms (S-Agent) presented in [5], finding the optimal took 13 seconds.

Aircraft-Landing-15. It shows the similar pattern as Job-Shop-6.

Viking-Bridge-15. For the sequential algorithms, BFS finds the optimal result of 220 at 66 sec. DFS/RDFS are omitted because they only report

results far exceeding 220. DFS^{G} gets trapped by the local optimal within the cost range from 285 to 265. For the distributed algorithms, D-DFS^{G} reports the best quality near optimal results and reaches the optimal at 28 sec. D-BFS^{S} finds the optimal result at 31 sec. Both are faster than sequential BFS. D-RDFS/D-DFS/D-BFS report near optimal results from high to low quality.

Task-Graph-88. BFS based algorithms BFS/D-BFS/D-BFS^{S} are omitted because they report no result even in 4 hours. Only DFS based algorithms can report near optimal results. For the sequential algorithms, DFS^{G}/RDFS/DFS find results from high to low quality. For the distributed algorithms, D-DFS is superior among all algorithms. Compared with the best of our swarm algorithms (S-Agent, included in the plot) presented in [5], D-DFS finds better solutions faster and reaches the highest quality result of 328 as early as 46 sec.

Conclusions. D-DFS/D-DFS^{G}/D-RDFS are generally very fast at finding near optimal results.

4.4 Memory and Communication (Metric 4)

Table B.5 shows statistics about memory consumption and communication overhead. For each model, column M shows the total memory (in GB) consumed by all UPPAAL processes. Distributed algorithms will generate more states than sequential algorithms, because emigrant states are generated by multiple workers and then sent to the owner process. These emigrant states contribute to the computation and communication overhead. Column R compares the amount of emigrant states generated and transmitted against the state-space size. Taking jb-6 as an example, D-BFS generates a large amount

Table B.5: Resident Memory (GB) and Communication Overhead

Models	jb-6	alp-15	vik-15
	M	M	M
BFS	0.19	0.15	0.48
DFS	2.54	0.16	*19.51*
DFS^{G}	0.22	0.12	2.57
RDFS	0.85	0.25	*19.47*

Models	jb-6			alp-15			vik-15		
	M	R	C%	M	R	C%	M	R	C%
D-BFS	1.71	14.7	26.3	1.17	14.7	23.5	3.25	15.4	63.3
D-BFS^{S}	0.67	3.9	62.5	0.67	4.0	56.3	1.31	2.8	77.4
D-DFS	1.77	14.9	45.2	1.14	22.0	45.8	6.22	20.9	64.5
D-DFS^{G}	1.82	14.6	27.6	1.33	23.1	44.7	3.93	10.8	58.0
D-RDFS	1.82	14.5	36.7	1.18	21.3	47.6	5.15	16.1	59.8

Distributed algorithms are at 32 cores. *Italic* font denotes 4-hour timeout.

of emigrant states that is 14.7 times the state-space, but the factor is only 3.9 for D-BFS^S.

Column C shows the portion of the communication time out of the total execution time in percentage. Because a lot of emigrant states are transmitted, the communication overhead of distributed algorithms is also high (up about 47% of the total execution time). For D-BFS^S, this may be even higher due to the level synchronization overhead.

Distributed algorithms normally consume more memory than their sequential counterparts. The average scale ratio is 5 times for the three models. There are three reasons. First, when a UPPAAL process initiates prior to local search, it consumes about 13 MB memory for core data structures and libraries. Then 32 UPPAAL processes will consume 416 MB that already double the size of some sequential algorithms' peak memory usage. Second, MPI allocates system buffers for message passing. Third, UPPAAL's internal memory management only marks obsolete memory slots for re-use to store newly explored states rather than returning them to the operation system (quite sensibly expecting to save time for repeated page (re-)allocation). When a distributed algorithm is running however, more memory is used mainly because it receives and stores a large number of emigrant states from the network as indicated by the R columns which approximate 14 times the size of the state-space for most distributed algorithms.

Conclusions. The distributed algorithms have memory, computation and communication overhead. There may be several ways to optimize memory management and communication (e.g. caching more emigrant states locally). However, we have currently decided against implementing these, in part because the performance is quite reasonable, and in part because we envision that the most effective approach will be to incorporate multi-core shared memory techniques that provides more fine-grained parallelism, better locality and lower communication and synchronization overhead. However, implementing an efficient (lock-less or lock-free) multi-core and thread-safe version of the internal exploration and optimized memory layout in UPPAAL requires great care.

5 Conclusions

We developed five distributed algorithms to accelerate timed optimal reachability analysis. We performed four benchmark experiments in terms of ability to compute near- or proven-optimal solutions, scalability, time and memory consumption and communication overhead. The experiment results are very promising. Based on the evaluation we conclude: (1) D-BFS^S can terminate fast thus prove the optimal result for large models; (2) D-DFS/D-DFS^G/D-RDFS are good at finding (near-) optimal results. For the future work, we

will develop parallel and distributed algorithms applying multi-core shared memory as the significant optimization. We will develop hybrid algorithms that combine the benefits as well as the state-of-the-art advances from distributed and swarm verification [5].

Acknowledgments

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

A Results for Runtime

Table B.6: Runtime (sec) of Aircraft-Landing-15

#C	BFS				DFS		DFS^G		RDFS	
	t_{opt}	t_{prov}			t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}
1	155	155			71	419	5	135	184	935
#C	D-BFS		$D\text{-}BFS^S$		D-DFS		$D\text{-}DFS^G$		D-RDFS	
	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}	t_{opt}	t_{prov}
2	422	454	134	135	77	408	13	351	193	628
4	71	243	69	69	1	345	< 1	207	2	326
8	4	86	42	42	< 1	139	< 1	123	< 1	140
16	7	39	26	26	< 1	80	< 1	66	< 1	82
32	11	34	16	16	< 1	48	< 1	47	< 1	42
64	5	14	11	11	1	22	< 1	21	1	23
128	2	10	15	16	1	12	< 1	14	1	14
256	1	8	28	29	< 1	9	< 1	7	< 1	9
512	1	8	49	51	< 1	17	< 1	10	< 1	15

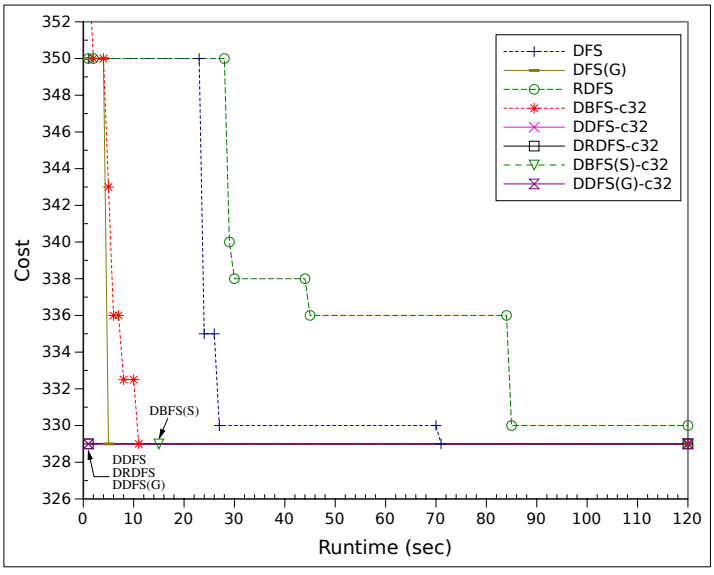

Fig. B.4: Cost vs. Runtime for Aircraft-Landing-15

References

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

[2] Y. Abdeddaïm, E. Asarin, and O. Maler, "Scheduling with timed automata," *Theoretical Computer Science*, vol. 354, no. 2, pp. 272–300, 2006.

[3] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *SFM*, ser. LNCS, vol. 3185. Springer, 2004, pp. 200–236.

[4] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *CAV*, ser. LNCS, vol. 1427. Springer, 1998, pp. 546–550.

[5] Z. Zhang, B. Nielsen, and K. G. Larsen, "Time optimal reachability analysis using swarm verification," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, S. Ossowski, Ed. ACM, 2016, pp. 1634–1640.

[6] A. Fehnker, *Bounding and Heuristics in Forward Reachability Algorithms*. UB Nijmegen [Host], 2000. [Online]. Available: http://books.google.dk/books?id=7cEPtwAACAAJ

[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*, ser. LNCS, vol. 2034. Springer, 2001, pp. 147–161.

[8] G. Behrmann and A. Fehnker, "Efficient guiding towards cost-optimality in uppaal," in *TACAS*, ser. LNCS, vol. 2031. Springer, 2001, pp. 174–188.

[9] U. Stern and D. L. Dill, "Parallelizing the Murφ Verifier," in *CAV*, ser. LNCS, vol. 1254. Springer, 1997, pp. 256–278.

[10] G. Behrmann, T. Hune, and F. W. Vaandrager, "Distributing timed model checking - how the search order matters," in *CAV*, ser. LNCS, vol. 1855. Springer, 2000, pp. 216–231.

[11] G. Behrmann, "Distributed reachability analysis in timed automata," *STTT*, vol. 7, no. 1, pp. 19–30, 2005.

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

[13] S. Blom, J. van de Pol, and M. Weber, "Ltsmin: Distributed and symbolic reachability," in *CAV*, ser. LNCS, vol. 6174. Springer, 2010, pp. 354–359.

[14] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3 - A modern refinement checker for CSP," in *TACAS*, 2014, pp. 187–201.

[15] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Designing fast LTL model checking algorithms for many-core gpus," *Journal of Parallel and Distributed Computing*, vol. 72, no. 9, pp. 1083–1097, 2012.

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

[17] K. G. Larsen, P. Pettersson, and W. Yi, "Model-checking for real-time systems," in *FCT*, ser. LNCS, vol. 965. Springer, 1995, pp. 62–88.

[18] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets*, ser. LNCS, vol. 3098. Springer, 2003, pp. 87–124.

[19] G. Tel, *Introduction to Distributed Algorithms*, 2nd ed. New York, NY, USA: Cambridge University Press, 2001.

[20] K. Verstoep, H. E. Bal, J. Barnat, and L. Brim, "Efficient large-scale model checking," in *IPDPS*. IEEE, 2009, pp. 1–12.

Paper C

Pareto Optimal Reachability Analysis for Simple Priced Timed Automata

Zhengkui Zhang, Brian Nielsen, Kim Guldstrand Larsen,
Gilles Nies, Marvin Stenger, and Holger Hermanns

The paper is under submission

The layout has been revised.

Abstract

We propose Pareto optimal reachability analysis to solve multi-objective scheduling and planing problems using real-time model checking techniques. Not only the makespan of a schedule, but also other objectives involving quantities like performance, energy, risk, cost etc, can be optimized simultaneously in balance. We develop the Pareto optimal reachability algorithm for Uppaal *to explore the state-space and compute the goal states on which all objectives will reach a Pareto optimum. After that diagnostic traces are generated from the initial state to the goal states, and Pareto optimal schedules are obtainable from those traces. We demonstrate the usefulness of this new feature using two case studies.*

1 Introduction

In reactive system design, engineers face the challenge of optimizing schedules regarding a variety of quantitative objectives like the makespan of a schedule, performance, energy consumption, resource intensiveness, risk assessment etc. Because in most cases a subset of these objectives are conflicting, there may not always exist any single solution that can simultaneously optimize all objectives, but advisable trade-offs ought to be made by human decision makers. This problem is generally called *multi-objective optimization* (MOO) which is common in many academic fields. Vilfredo Pareto (1848–1923) proposed the well-known concept of *Pareto optimality* as "the state of allocating resources where it is impossible to make any one individual better off without making at least one individual worse off." A solution is called *Pareto optimal* if none of the objectives can be improved in value without degrading some of the other objective values. Without additional preference information, all Pareto optimal solutions are considered equally good.

Related Work. Pareto optimality concepts have been used to solve numerical multi-objective scheduling problems in many fields, such as operation research [1], cloud services [2], networking [3], etc. In real-time system design, mature model checkers like Uppaal [4] and Kronos [5] have been successfully extended to do quantitative analysis. In particular, Uppaal-cora aims at solving optimal scheduling and planning problems modeled by *priced timed automata* (PTA) [6, 7]. PTA uses an additional observer clock to accumulate cost according to either discrete price annotations on transitions or price rates on locations. The scheduling problem boils down to a cost-optimal reachability problem. The reachability algorithm is also enhanced by *branch and bound* (B&B), which can effectively prune parts of the state-space that for sure will not to lead to an optimal solution, avoiding exploring the entire state-space. In [8] the optimal reachability analysis on the *multi-priced timed automata* (MPTA) was proved decidable. However, the model checking

problem on MPTA was proven undecidable [9].

Contributions. Firstly we introduce *simple priced timed automata* (SPTA) to model a subset of multi-objective scheduling problems. Particularly we only allow discrete prices on transitions for multiple cost variables. The formalism still proves to be very useful in practice, although price rate on locations are not supported. For instance, if the span of a task is known in advance, its overall energy consumption is (approximately) determined, or if a resource is not affected at all by the task's span. **Secondly** we provide the *Pareto optimal reachability* (POR) algorithms to compute Pareto optimal costs when reaching target goal states. Diagnostic traces are obtainable from the initial state to the goal states, and Pareto optimal schedules are obtainable from those traces. **Thirdly** we implement the semantics of SPTA and POR algorithms as a new feature in Uppaal. **Fourthly** we demonstrate the usage of this feature using two case studies: (1) time-optimal and power-aware scheduling of a task graph; (2) power-aware scheduling of the GomX-3 nano satellite.

Outline. The rest of the paper is organized as follows. Section 2 defines simple priced automata and Pareto optimality. Section 3 explains the Pareto optimal reachability algorithms and implementation. Section 4 gives the experiment results of two case studies. Section 5 concludes.

2 Preliminaries

This section gives the formal definitions for simple priced timed automata (SPTA) and Pareto optimality. For brevity parallel composition of SPTA is omitted.

2.1 Simple Priced Timed Automata

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

Definition 1. *A* Timed Automaton (TA) [10] *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, each of which contains a source location, a guard, an action, a set of clocks to be reset and a target location;* $Inv : L \to \mathcal{B}(X)$ *sets an invariant for each location. For simplicity an edge* $(\ell, g, a, r, \ell') \in E$ *is written as* $\ell \xrightarrow{g,a,r} \ell'$.

Let $\bar{p} = [p_1, p_2, \dots, p_k]$ denote a finite vector of k *prices*, where $p_i \in \mathbb{N}$.

Definition 2. *A* Simple Priced Timed Automaton (SPTA) *extends TA as a 7-tuple $\mathcal{S} = (\mathcal{A}, P)$, where: $\mathcal{A}$ is timed automaton, $P : E \to \mathbb{N}^k$ assigns vectors of prices $\bar{p}$ to edges.*

Definition 3. *The semantics of a simple priced timed automaton $\mathcal{S}$ is a priced timed transition system $S_{\mathcal{S}} = (Q, Q_0,$
$\Sigma, \to)$, where: $Q = \{(\ell, v) \mid (\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ 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:*
(1) $(\ell, v) \xrightarrow{a}_{\bar{p}} (\ell', v')$ if there is an edge $(\ell \xrightarrow{g,a,r} \ell') \in E$ such that $v \models g$, $v' = v[r \mapsto 0]$, $v' \models Inv(\ell')$, and $\bar{p} = P(\ell \xrightarrow{g,a,r} \ell')$ is the vector of prices for this edge;
(2) $(\ell, v) \xrightarrow{d}_{\bar{0}} (\ell, v + d)$ such that $v \models Inv(\ell)$, $v + d \models Inv(\ell)$, and $\bar{0}$ denotes the zero-price vector for delay.

Definition 4. *A* trace *(or* run*) ρ of $\mathcal{S}$ can be expressed in $S_{\mathcal{S}}$ as a sequence of alternative delay and action transitions starting from the initial state: $\rho = q_0 \xrightarrow{d_1}_{\bar{0}} q'_0 \xrightarrow{a_1}_{\bar{p}_1} q_1 \xrightarrow{d_2}_{\bar{0}} q'_1 \xrightarrow{a_2}_{\bar{p}_2} \cdots \xrightarrow{d_n}_{\bar{0}} q'_{n-1} \xrightarrow{a_n}_{\bar{p}_n} q_n \cdots$, 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 of q.*

Definition 5. *The* cost *(or* cost vector*) of a finite trace ρ is defined as the finite sum of all the prices along the trace $\texttt{Cost}(\rho) = \Sigma_{i=1}^{n} \bar{p}_i$. For a given location ℓ,* multi-objective scheduling *on SPTA is to minimize $\texttt{Cost}(\rho)$, where finite traces ρ end in (ℓ, v) for all possible v.*

2.2 Pareto Optimality

Multi-objective scheduling (MOS) tries to minimize a vector of costs. We resort to the concept of Pareto optimality to compute a set of Pareto optimal result cost vectors that are mutually incomparable. Then human decision makers can choose the most appropriate results that best fit and balance the problem objectives.

Definition 6. *Let $\bar{c} = [c_1, c_2, \ldots, c_k]$, $\bar{b} = [b_1, b_2, \ldots, b_k]$ denote two cost vectors. $\bar{c}$* Pareto dominates *$\bar{b}$ (written as $\bar{c} \prec \bar{b}$), iff both the following conditions are true:*
(1) $\forall i \in \{1, \ldots, k\} \quad c_i \leq b_i$;
(2) $\exists j \in \{1, \ldots, k\} \quad c_j < b_j$.

Definition 7. *A result cost vector $\bar{c}$ is* Pareto optimal *if there does not exist another cost vector $\bar{b}$ such that $\bar{b} \prec \bar{c}$. The set of Pareto optimal results is called the* Pareto frontier.

3 Pareto Optimal Reachability

The real-time model-checker UPPAAL works by exploring a finite *symbolic reachability graph*, where the nodes are *symbolic states*. A symbolic state of TA 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* [11], which is normally efficiently represented and stored in memory as *difference bound matrices* (DBM) [12].

The symbolic state of SPTA extends that of TA as $(\ell, \langle Z, \bar{c}\rangle)$, where $\bar{c}$ is the cost (or cost vector) of a finite trace ρ that ends in (ℓ, v) and $v \in Z$. Therefore, symbolic states in SPTA with the same ℓ and Z are discriminated by $\bar{c}$. We call $\langle Z, \bar{c}\rangle$ the *discrete priced zone* of a symbolic state in SPTA. We further define Pareto dominance between discrete priced zones as: $\langle G, \bar{u}\rangle \preceq \langle Z, \bar{c}\rangle$ iff $Z \subseteq G \wedge (\bar{u} = \bar{c} \vee \bar{u} \prec \bar{c})$.

3.1 Pareto Optimum on Prices

Algorithm 1 shows the Pareto optimal reachability algorithm that computes the Pareto optimal cost vector at goal states satisfying the proposition `Goal`. WAITING and PASSED keep unexplored and explored symbolic states respectively; and WAITING has the initial state. FRONT maintains the Pareto frontier consisting of current Pareto optimal costs at goal states. Inside procedure `Main`, whenever WAITING is not empty, an unexplored state is popped from WAITING in a loop. If the state is a goal state, the current cost $\bar{c}$ is passed into procedure `Update` to check for Pareto dominance with the elements inside FRONT, and update FRONT if necessary. At line 10 all elements in FRONT that are Pareto dominated by $\bar{c}$ are discarded. At line 12 $\bar{c}$ is added into FRONT, if existing elements in FRONT do not Pareto dominate it.

If the state is not goal state, it is subject to both inclusion checking and B&B elimination at line 5, and discarded if either test satisfies. Procedure `Included` says that a state is included, if a previously explored state in PASSED with the same location has its discrete priced zone dominate that of the current state (as $\langle Z', \bar{c}'\rangle \preceq \langle Z, \bar{c}\rangle$). A state is eligible for pruning in procedure `Prune`, if $\bar{c}$ is Pareto dominated by an element in FRONT. If the state endures the two tests at line 5, it is added to PASSED as already explored, and then its successor states are generated and added to WAITING. For simplicity we denote the action and delay transitions between symbolic states uniformly as $\rightsquigarrow$.

3.2 Pareto Optimum on Objective Functions

We propose three extensions to make Algorithm 1 more powerful and flexible: (1) support formatting multi-objectives as a vector of objective functions $F(\bar{c}) = [f_1(\bar{c}), f_2(\bar{c}), \ldots, f_n(\bar{c})]$ parameterized by the cost vector $\bar{c}$; (2) support

Algorithm 1: Pareto Optimal Reachability

```
WAITING ⟵ {(ℓ_0, ⟨Z_0, 0̄⟩)}, PASSED ⟵ ∅, FRONT ⟵ ∅
Procedure Main()
1   | while WAITING ≠ ∅ do
2   |   | select (ℓ, ⟨Z, c̄⟩) from WAITING
3   |   | if (ℓ, ⟨Z, c̄⟩) ⊨ Goal then
4   |   |   | Update(c̄)
5   |   | else if ¬Included((ℓ, ⟨Z, c̄⟩)) and ¬Prune(c̄) then
6   |   |   | add (ℓ, ⟨Z, c̄⟩) to PASSED
7   |   |   | forall the (ℓ', ⟨Z', c̄'⟩) such that (ℓ, ⟨Z, c̄⟩) ⇝ (ℓ', ⟨Z', c̄'⟩) do
8   |   |   |   | add (ℓ', ⟨Z', c̄'⟩) to WAITING
9   | return FRONT
Procedure Update(c̄)
10  | FRONT ⟵ FRONT \ {φ ∈ FRONT | c̄ ≺ φ}
11  | if ∀φ ∈ FRONT s.t. c̄ ⊀ φ then
12  |   | FRONT ⟵ FRONT ∪ {c̄}
Procedure Included((ℓ, ⟨Z, c̄⟩))
13  | if ∃(ℓ, ⟨Z', c̄'⟩) ∈ PASSED s.t. Z ⊆ Z' ∧ (c̄' = c̄ ∨ c̄' ≺ c̄) then return
    |   TRUE
14  | return FALSE
Procedure Prune(c̄)
15  | if ∃φ ∈ FRONT, φ ≺ c̄ then return TRUE
16  | return FALSE
```

a global clock (let us call it `now`) as a singular objective function to measure the makespan (accumulated delay on a finite trace); (3) support negative prices on action transitions. The first extension requires procedures `Update` and `Prune` to evaluate $F(\bar{c})$, and `Front` to contain Pareto optimal outcomes of $F(\bar{c})$ on goal states. We define *monotonically increasing* for $f \in F(\bar{c})$ as: $\bar{c} \prec \bar{c}' \Rightarrow f(\bar{c}) < f(\bar{c}')$. These three extensions however, are applied under specific additional conditions:

Cond 1 For extension 1, if $\exists f \in F(\bar{c})$ is not monotonically increasing, `Prune` must be disabled, and the Pareto check of cost vectors $\bar{c}' \prec \bar{c}$ at line 13 in `Included` must be skipped.

Cond 2 For extension 2, clock `now` must not be reset nor tested in guards or invariants.

Cond 3 For extension 3, (1) the state-space graph of the model must be acyclic; (2) if $\forall f \in F(\bar{c})$ are monotonically increasing, `Prune` must

be disabled; (3) if $\exists f \in F(\bar{c})$ is not monotonically increasing, do as in **Cond 1**.

B&B pruning and Pareto inclusion checking are valid only if the costs and evaluation results of objective functions are monotonically increasing. Conditions 1 & 3 are of utmost importance to notice, otherwise there is a risk to have incomplete results due to discarding some intermediate states prematurely that may lead to better results on goal states. The consequence of applying these two conditions is to explore the full state-space. Because the Pareto inclusion checking decays to normal inclusion checking as in the standard reachability algorithm, and the state-space is not pruned.

We extended UPPAAL to compute Pareto optimum on prices and objective functions of SPTA. The query to enable this new feature inside the verifier follows the syntax of:

$$\texttt{PO}\ (f_1, f_2, \ldots, f_k)\ [-(\texttt{L1}|\texttt{L2})] : \texttt{E} <> \texttt{Goal},$$

where `PO` is the keywords for Pareto optimum, f_i $(i \in [1,k])$ are objective functions or cost variables. Next comes the optional switch: $[-\texttt{L1}]$ disables pruning only, or $[-\texttt{L2}]$ disables both pruning and Pareto inclusion checking as in **Cond 1**. Following the colon is the normal reachability query. `Goal` is the proposition to specify the target goal states. If an objective is to be maximized, it is equivalent to put it in negative. But this typically turns a monotonically increasing objective function into decreasing, then $[-\texttt{L2}]$ is necessary.

4 Experiment Results

4.1 Case Study 1: Task Graph Scheduling

A task graph consists of a number of computation tasks with precedence constraints (predecessor tasks) such that a task can start only if all its predecessor tasks have completed. In this case study, an embedded system has 16 tasks, whose precedence constraints are within [0,3] and processing time are predictable and within the range of [1,66] clock cycles. Those jobs can be scheduled on four processors with the speeds of [1,1,2,2] clock cycles per time unit and the power consumptions at busy state of [10,10,40,40] micro watts per time unit. We neglect the power for processors at idle state. The objective is to synthesize a non-preemptive schedule that can minimize the time for all tasks to terminate as well as the total power consumption by four processors.

Figure C.1 depicts the dependency graph of 16 tasks. These precedence constraints are coded as a dependency matrix in the UPPAAL model. Figure

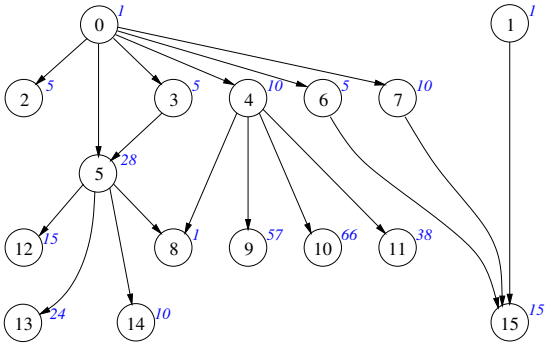

Fig. C.1: Task dependency graph. Task Ids are in the center of nodes. Predicted clock cycles of tasks are in blue italic font on the top right corner of nodes.

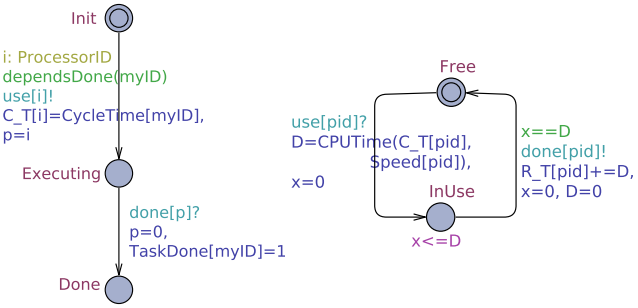

Fig. C.2: Templates for `Task` (left) and `Processor` (right)

C.2 shows the templates for task and processor. `Task` is scheduled if the guard "`dependsDone()`" approves that all its predecessor tasks are completed. Once a processor is available, the task is bound to that processor. The predefined clock cycles for this task is also passed to that processor. Then the task starts executing until it is notified for termination by signal "`done[p]`". `Processor` transforms from `Free` to `InUse` once it is scheduled to handle a task, meanwhile "`CPUTime()`" calculates the expected execution time `D` in time units from clock cycles of a task and current processor speed. After delaying at `InUse` for `D` time units, `Processor` moves back to `Free` and notifies the binding task. `R_T[pid]`, which keeps the accumulated elapsing time at `InUse` for each processor and acts as cost variables, is also increased by D.

```
PO (10*(R_T[0]+R_T[1])+40*(R_T[2]+R_T[3]), now) :
E<> forall (i:TaskID) Task(i).Done && now<=65
```

The original goal to minimize makespan and energy consumption is expressed as the query above. The Pareto optimality section contains two objective functions: the total energy consumption expressed as the linear combi-

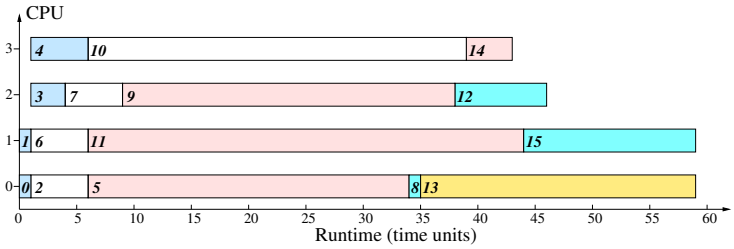

Fig. C.3: Schedule for the 3rd outcome (4700, 59). Horizontal bar denotes tasks scheduled on a processor. The segments inside each bar denote individual tasks with the task ids in the front of each segment.

nation of power and processor in-use time, and a global clock `now` measuring the makespan. The reachability proposition section specifies all tasks are to complete and the makespan is equal to or less than 65 time units. UPPAAL reports seven Pareto optimal outcomes as follows with the corresponding traces. Assuming we prefer the 3rd outcome (4700, 59) with the energy consumption of 4700 micro joules and makespan of 59 time units, we can parse the trace into a visualizable schedule as shown in Figure C.3.

```
1. (4600, 63)   3. (4700, 59)   5. (4800, 54)   7. (4560, 65)
2. (4770, 55)   4. (4850, 51)   6. (4590, 64)
```

4.2 Case Study 2: Nano Satellite Scheduling

The GOMX-3 CubeSat is a 3 liter 3 kg nano satellite commissioned by the European Space Agency (ESA). It was designed, delivered, and operated by GomSpace in Aalborg Denmark, and was launched from Japan aboard the HTV-5 cargo spacecraft on August 19th 2015. GOMX-3 was successfully deployed on October 5 2015. The satellite supports precise 3-axis rotation by gyroscopes and magnetorquers which enable the following main payloads: (1) in-flight tracking of ADS-B beacons emitted by commercial aircrafts, (2) monitoring signals from geostationary INMARSAT satellites by L-Band receiver, (3) high-speed downlinking collected data to stations in Toulouse (France) or Kourou (French Guiana) by X-Band transmitter and UHF radio module, (4) uplinking new instructions to and monitoring status of GOMX-3 from GomSpace by the UHF module.

The purposes of GOMX-3 are tracking commercial aircrafts, testing X-Band transmitters, and monitoring INMARSAT satellites. ESA and GomSpace want to maximize the amount of *jobs* (operations of payloads) without depleting the on-board battery. Power is the most critical sparse resource for a satellite in orbit. In particular when GOMX-3 passes into eclipse, the battery is the only source for it to draw power from. If battery voltage drops below

14.4 volts, the satellite switches to the safe mode, where all non-essential hardware components are switched off, preventing the satellite from being productive. Since GomX-3 follows an equatorial orbit, insolation periods and possible operation windows for different payloads are predictable over the time horizon of a few days, hence the power budget of jobs can be predicted.

In [13] a PTA model of GomX-3 was analyzed by Uppaal-cora to generate productive and power-aware schedules for GomX-3 to carry out jobs over 20 orbits (about 31 hours) around the earth. Three types of jobs were scheduled in the model: data collection by payloads (2), data downlink by payload (3), and satellite control by payload (4). The principle idea was to assign a penalty price rate R_i to each skipped job J_i ($i \in [0,6]$). The satellite control jobs should always be scheduled whenever possible, and data downlink jobs are given the highest penalty price rate. If a job J_k is skipped, the integral of the R_k over the operation window W_k of this skipped job (equal $R_k \times W_k$) contributes to the global penalty cost (or weighted sum of skipped jobs). Then Uppaal-cora searches the entire state-space and finds the cost optimal trace which has the smallest penalty cost.

In this paper we adapt the original PTA model in [13] into a SPTA model, and use Pareto optimal reachability analysis to generate schedules that optimize the productivity of payloads and energy consumption simultaneously in a natural way. Instead of using penalty price rates, we directly record the number of operated jobs for data collection, data downlink and satellite control. Not only to maximize the different kinds of operated jobs, but also the

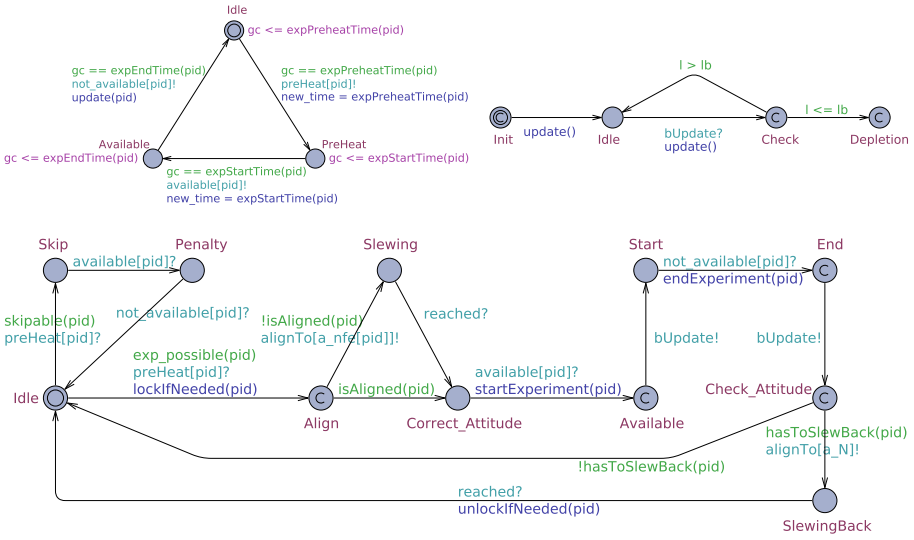

Fig. C.4: Templates for `Provider` (top left), `Battery` (top right) and `Experiment` (bottom)

data collection and downlink jobs are important to be kept in balance. We wish the remaining battery level is high enough too.

There are six template automata in the model (detailed description in [13]). Figure C.4 shows the three principle ones. (1) `Provider` takes care of initiating and terminating jobs on each payload repeatedly. It waits at location `Idle` for every predicted operation window to come, notifies `Experiment` to start preheating and to start actual operation after preheating is completed, then moves back to `Idle`. (2) `Battery` represents a linear battery model with capacity. It can be charged/discharged with piecewise constant energy gain from solar panels or energy drain by payloads. If the battery level is below a threshold `lb` which is 40% of the maximum capacity, a deadlock state is reached via the transition `Check` $\rightarrow$ `Depletion`. (3) `Experiment` models two possible outcomes when it is notified of a job opportunity. A job can be skipped because of low priority or resource constrains. Otherwise, the job can execute by slewing to the predefined attitude, performing actual operation and slewing back to the normal attitude. We count the number of fully executed jobs on each individual payloads as cost variables. The remaining three templates are: (4) `AttitudeControl` for slewing GomX-3 to the predetermined attitude of each job, (5) `Sun` for switching on (off) energy harvesting from solar panels based on the predicted insolation and eclipse time, and (6) `OrbitCounter` for monitoring and counting the completed orbits of GomX-3.

```
PO (-getAllJobs(), -getCollectJobs(), -getDownlinkJobs(), -l)[-L2] :
E<> n==20 && l>89856000 && ac_lock==0 && lastXband
```

Our goal to maximize the number of all executed jobs, data collection and downlink jobs, and remaining battery level over 20 orbits is expressed as the query above. Functions `getAllJobs`, `getCollectJobs` and `getDownlinkJobs` count the number of executions for all jobs and data collection jobs and data downlink jobs respectively, and `l` is the battery level. They are put in negative form because we want to get their maximum values. Realizing that prices on battery level `l` are negative[1] when discharging, and the former three objective functions are monotonically decreasing, switch $[-$`L2`$]$ is turned on (sec **??**). The reachability proposition section specifies that the orbit count is to reach 20, the battery level[2] is above 55% of capacity, and two additional conditions proposed by GomSpace engineers.

Uppaal reports 15 Pareto optimal outcomes as follows with the corresponding traces. Assuming we prefer the 12th outcome (-15, -4, -5, -97648603) with 15 jobs in total, 4 data collection and 5 data downlink jobs, and remaining battery level at 97648603 milli joules. We can parse the trace and obtain

[1] If negative prices are present, the state-space graph must be acyclic. This is guaranteed by the finite time horizon over 20 oribts (about 31 hours).

[2] This level is larger than the threashold of 40% in the linear battery model so as to make the satellite on-board battery that is non-linear work safer in the real situation.

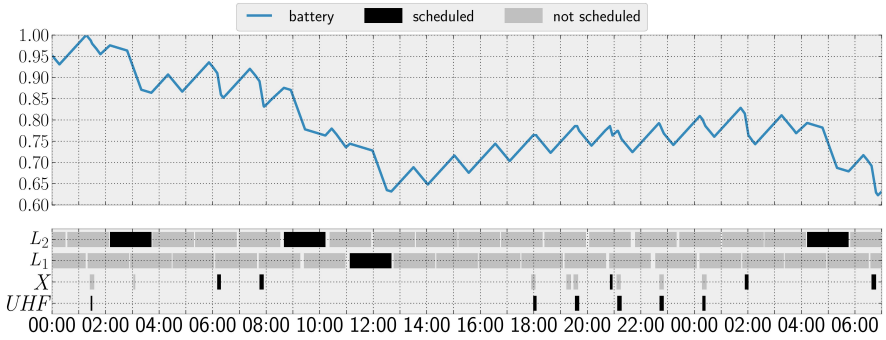

Fig. C.5: Schedule for the 12th outcome (-15, -4, -5, -97648603) consisting of battery level plot (top) and payload operation plot (bottom). In the payload operation plot, L_1 and L_2 show 4 data collection jobs, X shows 5 data downlink jobs, and *UHF* shows 6 satellite control jobs. Data collection and downlink jobs are maximized and kept in balance.

the visualizable schedule as shown in Figure C.4.

```
1.  (-11, -4, -1, -124833863)     9. (-14, -3, -5, -118593763)
2.  (-11, -3, -2, -138544388)    10. (-14, -4, -4, -104763388)
3.  (-12, -3, -3, -133181973)    11. (-14, -2, -6, -131801398)
4.  (-12, -4, -2, -118645643)    12. (-15, -4, -5,  -97648603)
5.  (-12, -5, -1, -100905253)    13. (-15, -3, -6, -110989688)
6.  (-13, -4, -3, -112367463)    14. (-15, -2, -7, -123752127)
7.  (-13, -3, -4, -126078388)    15. (-16, -3, -7, -102940417)
8.  (-13, -5, -2,  -96406878)
```

5 Conclusions

We introduced the Pareto optimal reachability analysis to solve multi-objective scheduling and planing problems modeled by simple priced timed automata. We developed the algorithms for Uppaal and performed two case studies. The future work are in three directions: (1) support the price rate on delays by merging the priced zone in Uppaal-cora into Uppaal; (2) develop efficient data structures for the Pareto frontier; (3) solve more industrial case studies.

Acknowledgments

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

References

[1] I. Kacem, S. Hammadi, and P. Borne, "Pareto-optimality approach for flexible job-shop scheduling problems: hybridization of evolutionary algorithms and fuzzy logic," *Mathematics and Computers in Simulation*, vol. 60, no. 3-5, pp. 245–276, 2002.

[2] A. S. A. Beegom and M. S. Rajasree, "A particle swarm optimization based pareto optimal task scheduling in cloud computing," in *ICSI*, ser. LNCS, vol. 8795. Springer, 2014, pp. 79–86.

[3] M. Khalesian and M. R. Delavar, "Wireless sensors deployment optimization using a constrained pareto-based multi-objective evolutionary approach," *Engineering Applications of Artificial Intelligence*, vol. 53, pp. 126–139, 2016.

[4] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *SFM*, ser. LNCS, vol. 3185. Springer, 2004, pp. 200–236.

[5] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *CAV*, ser. LNCS, vol. 1427. Springer, 1998, pp. 546–550.

[6] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Priced timed automata: Algorithms and applications," in *FMCO*, ser. LNCS, vol. 3657. Springer, 2004, pp. 162–182.

[7] R. Alur, S. La Torre, and G. J. Pappas, "Optimal paths in weighted timed automata," *Theoretical Computer Science*, vol. 318, no. 3, pp. 297–322, 2004.

[8] K. G. Larsen and J. I. Rasmussen, "Optimal reachability for multi-priced timed automata," *Theoretical Computer Science*, vol. 390, no. 2-3, pp. 197–213, Jan. 2008.

[9] T. Brihaye, V. Bruyère, and J. Raskin, "Model-checking for weighted timed automata," in *FORMATS*, ser. LNCS, vol. 3253. Springer, 2004, pp. 277–292.

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

[11] K. G. Larsen, P. Pettersson, and W. Yi, "Model-checking for real-time systems," in *FCT*, ser. LNCS, vol. 965. Springer, 1995, pp. 62–88.

[12] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets*, ser. LNCS, vol. 3098. Springer, 2003, pp. 87–124.

[13] M. Bisgaard, D. Gerhardt, H. Hermanns, J. Krcál, G. Nies, and M. Stenger, "Battery-aware scheduling in low orbit: The gomx-3 case," in *FM*, ser. LNCS, vol. 9995. Springer, 2016, pp. 559–576.

Paper D

Verification and Performance Evaluation of Timed Game Strategies

Alexandre David, Huixing Fang, Kim Guldstand Larsen,
and Zhengkui Zhang

The paper has been published in the
Proceedings of the 12th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS), Vol. 8711, pp. 100–114, 2014.

The layout has been revised.

Abstract

Controller synthesis techniques, based on timed games, derive strategies to ensure a given control objective, e.g., time-bounded reachability. Model checking verifies correctness properties of systems. Statistical model checking can be used to analyse performance aspects of systems, e.g., energy consumption. In this work, we propose to combine these three techniques. In particular, given a strategy synthesized for a timed game and a given control objective, we want to make a deeper examination of the consequences of adopting this strategy. Firstly, we want to apply model checking to the timed game under the synthesized strategy in order to verify additional correctness properties. Secondly, we want to apply statistical model checking to evaluate various performance aspects of the synthesized strategy. For this, the underlying timed game is extended with relevant price and stochastic information. We first explain the principle of translating a strategy produced by Uppaal-tiga *into a timed automaton, thus enabling the deeper examination. However, our main contribution is a new extension of* Uppaal *that automatically synthesizes a strategy of a timed game for a given control objective, then verifies and evaluates this strategy with respect to additional properties. We demonstrate the usefulness of this new branch of* Uppaal *using two case-studies.*

1 Introduction

Model checking (MC) of real-time systems [1] has been researched for over 20 years. Mature tools such as Uppaal [2] and Kronos [3] have been applied to numerous industrial case studies. Nowadays, more interesting formal methods for real-time systems are inspired by or derived from model-checking. Two remarkable ones are controller synthesis and statistical model checking. Controller synthesis techniques [4], based on games, derive strategies to ensure some given objective while handling uncertainties of the environment. Statistical model checking (SMC) [5], based on statistical analysis of simulations, is used to analyse reliability and performance aspects of systems, e.g., energy consumption. In the Uppaal toolbox, efficient implementations of these new techniques are found in the branches Uppaal-tiga [6] and Uppaal-smc [7].

We believe the three techniques can complement each other. Given a timed game and a control objective, controller synthesis will generate a strategy if the game is controllable. The strategy may ensure hard timing guarantees for a controller to win the game. We aim at verifying additional correctness properties by applying MC to the timed game under this strategy. Similarly, SMC should allow to infer more refined performance consequences (cost, energy consumption etc) of the synthesized strategy. For this, we extend the underlying timed game with prices and stochastic semantics.

There have been a few previous attempts to combine modeling, synthesis, verification and performance evaluation in a single paradigm. In [8] Franck et al. presented a tool chain – UPPAAL-TIGA for synthesis, PHAVER for verification, SIMULINK for simulation – to solve the energy consumption and wear control problem of an industrial oil pump case-study. In [9] UPPAAL-TIGA was combined with MATLAB and SIMULINK to achieve synthesis, simulation and executable code generation for the climate controller of a pig stable. These tool chains are not integrated inside one tool and require translations to let the different tools interact.

As the first contribution in this paper, we propose the principle of translating a synthesized strategy, as obtained from UPPAAL-TIGA, into a controller timed automaton. One can build a closed system using the controller and do model-checking in UPPAAL or statistical model-checking in UPPAAL-SMC. The second contribution is an extension of the semantics and algorithms of MC and SMC to use a synthesized strategy when exploring the state space (for MC) or generating random runs (for SMC). The third contribution is an implementation of this extension based on UPPAAL referred here as Control-SMC, which allows users to synthesize a timed game strategy then verify and evaluate this strategy automatically. It is worth noting that UPPAAL-TIGA may not guarantee that the synthesized strategy is time optimal and here we are interested in evaluating a given strategy w.r.t. a number of different cost measures.

The rest of the paper is organized as follows. Section 2 defines timed games and strategies. Section 3 provides the stochastic semantics of SMC. Section 4 describes the translation of a strategy to a timed automaton. Section 5 presents the extended SMC semantics and implementation of Control-SMC. Section 6 gives the experiment results on two case-studies using Control-SMC. The paper concludes with the future work in Section 7.

2 Timed Game

This section recalls the basic theory of timed game and controller synthesis. Controller synthesis aims at solving the following problem: Given a system S and an objective ϕ, synthesize a controller C such that C can supervise S to satisfy ϕ ($C(S) \models \phi$) regardless how the environment behaves. The problem can be formulated as a two-player game between the controller and the environment.

2.1 Timed Game 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: $g, g_1, g_2 ::= x \bowtie n \mid x - y \bowtie$

$n \mid g_1 \wedge g_2$, where $x, y \in X$ are clocks, $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

Definition 1. *A* Timed Automaton (TA) [10] *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:* $Q = \{(\ell, v) \mid (\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ 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* $(\ell \xrightarrow{g,a,r} \ell') \in E$ *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* $v + d \models Inv(\ell)$.

A timed game automaton is an extension of a timed automaton whose actions are partitioned into controllable actions for the controller and uncontrollable actions for the environment. Besides discrete actions, each player can decide to wait in the current location. As soon as one player decides to play one of his available actions, time will stop elapsing and the action will be taken.

Definition 3. *A* Timed Game Automaton (TGA) [11] *is a 7-tuple* $\mathcal{G} = (L, \ell_0, X, \Sigma_c, \Sigma_u, E, Inv)$ *where:* Σ_c *is the finite set of controllable actions,* Σ_u *is the finite set of uncontrollable actions,* Σ_c *and* Σ_u *are disjoint, and* $(L, \ell_0, X, \Sigma_c \cup \Sigma_u, E, Inv)$ *is a timed automaton.*

Let $S_{\mathcal{G}}$ be the timed transition system of $\mathcal{G}$. A *run* ρ of $\mathcal{G}$ can be expressed in $S_{\mathcal{G}}$ as a sequence of alternative delay and action transitions: $\rho = q_0 \xrightarrow{d_1} q'_0 \xrightarrow{a_1} q_1 \xrightarrow{d_2} q'_1 \xrightarrow{a_2} \cdots \xrightarrow{d_n} q'_{n-1} \xrightarrow{a_n} q_n \cdots$, where $a_i \in \Sigma_c \cup \Sigma_u$, $d_i \in \mathbb{R}_{\geq 0}$, q_i is state (ℓ_i, v_i), and q'_i is reached from q_i after delay d_{i+1}. $Exec_{\mathcal{G}}$ denotes the set of runs of $\mathcal{G}$ and $Exec^f_{\mathcal{G}}$ denotes the set of its finite runs.

Definition 4. *Given a timed game automaton* $\mathcal{G}$ *and a set of states* $K \subseteq L \times \mathbb{R}^X_{\geq 0}$, *the* control objective ϕ *can be: (i) a* reachability control problem *if we want* $\mathcal{G}$ *supervised by a strategy to reach* K *eventually, or (ii) a* safety control problem *if we want* $\mathcal{G}$ *supervised by a strategy to avoid* K *constantly.*

We can define a run $\rho \in Exec_{\mathcal{G}}$ as *winning* in terms of its control objective. For a reachability game, ρ is winning if $\exists k \geq 0, (\ell_k, v_k) \in K$. For a safety game, ρ is winning if $\forall k \geq 0, (\ell_k, v_k) \notin K$.

Definition 5. *A* strategy *for a controller in the timed game $\mathcal{G}$ is a mapping $s : Exec^f_{\mathcal{G}} \to \Sigma_c \cup \{\lambda\}$ satisfying the following conditions: given a finite run ρ ending in state $q = last(\rho)$, if $s(\rho) = a \in \Sigma_c$, then there must exist a transition $q \xrightarrow{a} q'$ in $S_{\mathcal{G}}$, or if $s(\rho) = \lambda$, λ being the delay action, then there must exist a positive delay $d \in \mathbb{R}_{>0}$ such that $q \xrightarrow{d} q'$ in $S_{\mathcal{G}}$.*

When a strategy only depends on the current state of the game, that is $\forall \rho, \rho' \in Exec_{\mathcal{G}}, last(\rho) = last(\rho')$ implies $s(\rho) = s(\rho')$, it is called a *positional* or *memoryless* strategy. The strategies for reachability and safety games, as the ones handled by Uppaal-tiga, are memoryless.

The analysis of TA and TGA is based on the exploration of a finite *symbolic reachability graph*, where the nodes are *symbolic states*. A symbolic state S is a pair (ℓ, Z), where $\ell \in L$, and $Z = \{v \mid v \models g_z, g_z \in \mathcal{B}(X)\}$ is a *zone* [1], which is normally efficiently represented and stored in memory as *difference bound matrices* (DBM) [12]. Uppaal-tiga uses efficient on-the-fly algorithms [4] that manipulate zones to solve timed games. The winning strategy $\hat{s}$ produced by Uppaal-tiga is also represented using zones. More precisely, for each location ℓ, $\hat{s}$ gives a finite set of pairs as $\hat{s}(\ell) = \{(Z_1, a_1), \ldots, (Z_n, a_n)\}$, where $a_i \in \Sigma_c \cup \{\lambda\}, Z_i \cap Z_j = \emptyset$ if $i \neq j$.

2.2 A Running Example

Fig. D.1 [4] shows a timed game automaton named `Main` which has one clock x and two types of edges: controllable (solid) and uncontrollable (dashed). The control objective is to find a strategy that can supervise `Main` to reach `goal`, regardless of the environment's behavior. The object is expressed as `control: A<> Main.goal`. The game is controllable, and Uppaal-tiga pro-

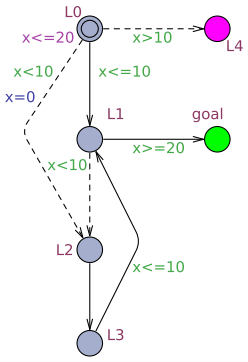

Fig. D.1: TGA `Main`

```
State: ( Main.L1 )
While you are in (10<=Main.x && Main.x<20), wait.
When you are in (20<=Main.x), take transition
    Main.L1->Main.goal { x >= 20, tau, 1 }
State: ( Main.L3 )
While you are in (Main.x<10), wait.
When you are in (Main.x==10), take transition
    Main.L3->Main.L1 { x <= 10, tau, 1 }
State: ( Main.L0 )
When you are in (Main.x==10), take transition
    Main.L0->Main.L1 { x <= 10, tau, 1 }
While you are in (Main.x<10), wait.
State: ( Main.L2 )
When you are in (Main.x<=10), take transition
    Main.L2->Main.L3 { 1, tau, 1 }
State: ( Main.goal )
While you are in true, wait.
```

Fig. D.2: A Strategy for `Main`

vides a strategy as shown in Fig. D.2 if running the command line version of UPPAAL-TIGA– `verifytga` with the option `-w0`. The strategy is a list of (zone, action) pairs indexed by locations. For example when `Main` is at `L1`, the action is to wait if $10 \leq x < 20$, or to take the action to reach `goal` if $x \geq 20$.

3 Stochastic Priced Timed Automata

In this section, we briefly recall the definition of priced timed automata and stochastic semantics of SMC. We borrow the definitions from [13].

3.1 Priced Timed Automata

Priced timed automata are a generalization of timed automata where clocks may have different rates in different locations. We note by $R(\ell) : X \to \mathbb{N}$ the *rate vector* assigning a rate to each clock of X at location ℓ. For $v \in \mathbb{R}^X_{\geq 0}$ and $d \in \mathbb{R}_{\geq 0}$, we write $v + R(\ell) \cdot d$ to denote the clock valuation defined by $(v + R(\ell) \cdot d)(x) = v(x) + R(\ell)(x) \cdot d$ for any $x \in X$.

Definition 6. *A* Priced Timed Automaton (PTA) *is a tuple* $\mathcal{P} = (L, \ell_0, X, \Sigma, E, R, I)$ *where: (i)* L *is a finite set of locations, (ii)* $\ell_0 \in L$ *is the initial location, (iii)* X *is a finite set of clocks, (iv)* $\Sigma = \Sigma_i \uplus \Sigma_o$ *is a finite set of actions partitioned into inputs* (Σ_i) *and outputs* (Σ_o)*, (v)* $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ *is a finite set of edges, (vi)* $R : L \to \mathbb{N}^X$ *assigns a rate vector to each location, and (vii)* $I : L \to \mathcal{B}(X)$ *assigns an invariant to each location.*

3.2 Stochastic Semantics

Consider a closed network of PTAs $\mathbf{A} = (\mathcal{P}_1 | \dots | \mathcal{P}_n)$ with a state space $St = St_1 \times \cdots \times St_n$. For a concrete global state $q = (q_1, \dots, q_n) \in St$ and $a_1 a_2 \dots a_k \in \Sigma^*$ we denote by $\pi(q, a_1 a_2 \dots a_k)$ the set of all maximal runs from q with a prefix $t_1 a_1 t_2 a_2 \dots t_k a_k$ for some $t_1, \dots, t_2 \in \mathbb{R}_{\geq 0}$, that is, runs where the i'th action a_i has been output by the component $\mathcal{P}_{c(a_i)}$. We give the probability for getting such sets of runs as:

$$\mathbb{P}_{\mathbf{A}}(\pi(q, a_1 a_2 \dots a_k)) = \int_{t \geq 0} \mu^c_q(t) \cdot \Big(\prod_{j \neq c} \int_{\tau > t} \mu^j_q(\tau) d\tau\Big) \cdot \gamma^c_{q^t}(a_1) \cdot \mathbb{P}_{\mathbf{A}}\big(\pi((q^t)^{a_1}, a_2 \dots a_n)\big) dt$$

where $c = c(a_i)$ is the *index* of component taking a_1, μ^c_q is the *delay density function* for component c to choose a delay t_i at q, and $\gamma^c_{q^t}$ is the *output probability function* for component c to choose an action a_i after q is delayed by t. The above nested integral reflects that the stochastic semantics of the network

is defined based on race among components. All components are independent in giving their delays which are decided by the given delay density functions. The player component who offers the minimum delay is the winner of the race, and takes the turn to make a transition and (probabilistically) choosing the action to output.

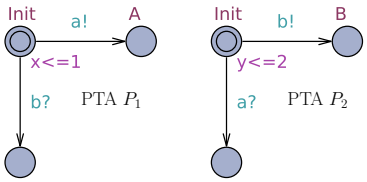

Fig. D.3: A Tiny Example

Fig. D.3 gives the intuition of the SMC semantics. Two PTAs P_1 and P_2 race to reach locations A or B. If P_1 enters A, it blocks P_2 to enter B, and vice versa. Furthermore, either PTA can delay uniformly within the invariants from its initial state before firing its output transition. We can use the SMC semantics to calculate the probability for P_1 to enter location A within 2 time units as:

$$\mathbb{P}(\pi(q_0, a)) = \int_{x=0}^{1} 1 \cdot \Big(\int_{y=x}^{2} \frac{1}{2} dy \Big) dx = \frac{1}{2} \int_{x=0}^{1} (2 - x) dx = \frac{3}{4}$$

where q_0 is the initial state of the network of P_1 and P_2, and the delay density functions for P_1 and P_2 at q_0 are 1 and $\frac{1}{2}$ respectively. P_1 can reach A only if it takes its transition before P_2.

4 Translating Strategies to Timed Automata

In this section, we provide a systematic way to translate a synthesized strategy of a timed game $\mathcal{G}$ produced by Uppaal-tiga into a controller timed automaton C. Once the controller is built, we can verify additional correctness properties or evaluate performance aspects of the closed system $C(\mathcal{G})$ in Uppaal.

4.1 The Method

We recall from Section 2.1 that strategies have the form $\hat{s}(\ell) = \{(Z_1, a_1), \ldots, (Z_n, a_n)\}$. Given a concrete state $q = (\ell, v)$, one can lookup which action a_i to take by finding Z_i such that $v \in Z_i$. Fig. D.4 illustrates how to translate the strategy from a location ℓ with the schematic zone representation (left) into a basic controller TA (right). The complete controller TA is obtained by repeating the same translation procedure for all locations and connecting all resulting basic controller TAs to the same initial state. The symbol "C" inside states indicates committed states. Time does not elapse in committed states, and the outgoing transitions are taken atomically. We use $\bar{Z}$ to denote the *closure* of the zone Z.

The small controller TA on the right is constructed as follows. For a given discrete state (ℓ) (location only), a transition from `Init` to a switch state `SW`

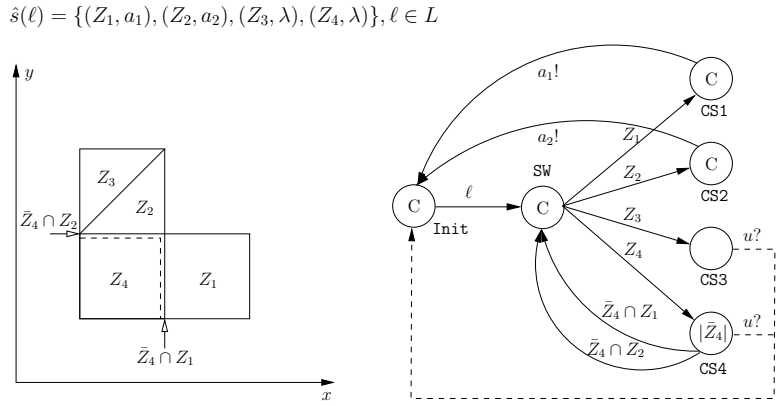

Fig. D.4: Translating the Strategy

is added with a guard encoding ℓ. From there we add transitions guarded by Z_i for each (Z_i, a_i) entry of $\hat{s}(\ell)$ to a choice state `CS`i. Then, we have tree basic cases: Either (1) a_i is a controllable action, (2) a_i is an unbounded delay, or (3) it is a bounded delay. In case (1), the controller takes a_i immediately with the synchronization $a_i!$ (e.g. from `CS1` and `CS2` in Fig. D.4). In case (2) corresponding to $a_i = \lambda$, the controller stays idle waiting for a move from the environment with the synchronization $u?$. Finally, case (3) is similar to case (2) except for the upper bound on the delay (encoded with an invariant) and additional transitions to go back to `SW` whenever the upper bound is reached and a controllable action is enabled.

4.2 The Running Example

We translate the strategy in Fig. D.2 into a controller TA C. Before translating, we need to synchronize C and $\mathcal{G}$ so that C can observe the state of $\mathcal{G}$ and control it. To observe the locations, we assign unique IDs and use global flags for each component to keep track of the current active location. Then we rename the local clocks to be global to make them visible. To monitor every uncontrollable transition in $\mathcal{G}$, we use a unique channel u and the synchronizations $u!$ in $\mathcal{G}$ and $u?$ in C. Similarly, to control $\mathcal{G}$, controllable actions a_i use the corresponding channel synchronizations $a_i!$ in C and $a_i?$ in $\mathcal{G}$.

In Fig. D.5, we define location IDs for `Main.L0` – `Main.L4` and `Main.goal` from 0 to 5. Then we use the global location flag *loc* to keep track of the current location of `Main`, and the global clock x to replace the local one, then the broadcast channels $u1, u2, a1$ – $a4$ to synchronize `Main` and its controller TA `MyCon` in Fig. D.6. In `MyCon`, by testing *loc* on the predefined location IDs, transitions from `Init` lead to the switch states `L0` – `L3` and `L5`, which correspond to the strategies at locations `Main.L0` – `Main.L3` and `Main.goal`.

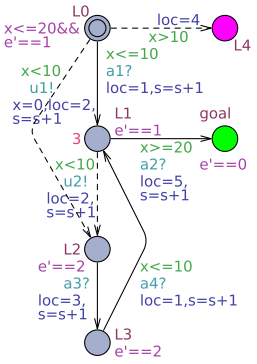

Fig. D.5: Decorated TGA `Main`

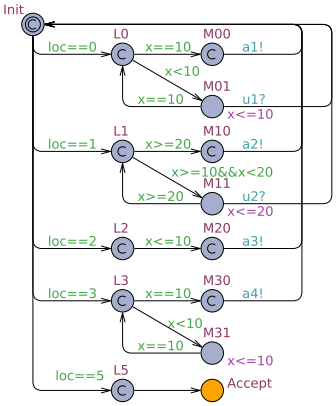

Fig. D.6: Controller TA `MyCon`

Choice states `M00`, `M10`, `M20` and `M30` depict case (1) in Fig. D.4. `Accept` corresponds to case (2). `M01`, `M11` and `M31` match case (3).

We also add price and a delay distribution to `Main` for performance evaluation in SMC. This essentially turns `Main` into a priced timed automaton. We use an integer s to count the number of transitions to reach `goal`, and a clock e to measure the energy consumption to reach `goal`. The rate of the clock e is specified at all locations as $e' == n, n \in \mathbb{N}$ except at `L4` because `L4` is not reachable under the strategy. e' is stopped at `goal` by setting to 0. Besides, an exponential rate of 3 is defined for the delay density function at `L1`. Now a closed system can be made from `Main` and `MyCon`. We can verify correctness properties and evaluate performance aspects of this strategy as shown in Table D.1.

Experiment 1 verifies the original control objective that is satisfied (`Yes`) for sure. Experiment 2 verifies if the strategy ensures `Main` to reach `goal` within 20 time units, where `time` is a global clock. The result is not satisfied (`No`). We evaluate reachability of `goal` within 30 time units under the strategy in experiment 3. The probability is `[0.902606,1] with confidence 0.95` if the probability uncertainty factor ϵ is 0.05. Besides, several kinds of statistical plots can be generated by UPPAAL-SMC such as probability distribution, probability density

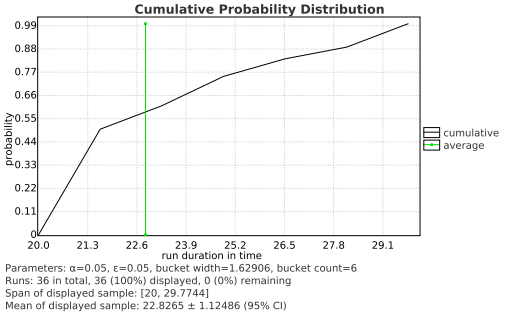

Fig. D.7: Distribution on Time to Reach `goal`

Table D.1: MC & SMC Experiments of the Running Example

	#	Queries	Results
MC	1	`A<> Main.goal`	Yes
	2	`A<> Main.goal and time<=20`	No
SMC	3	`Pr[<=30] (<> Main.goal)`	[0.902606,1]
	4	`E[<=30;200] (max: Main.s)`	3.05
	5	`E[<=30;200] (max: Main.e)`	27.5137

distribution, cumulative probability distribution, and frequency histogram. Fig. D.7 shows the cumulative probability distribution of 36 runs. The curve shows that over 55% of runs reach `goal` between 20.0 and 22.6 time units, and almost 90% runs can reach `goal` within 29.1 time units. The last two experiments report the expected number of steps and energy consumption to reach `goal` for 200 simulated runs within 30 time units.

5 MC and SMC under Strategies

Control-SMC is a new extension of UPPAAL. It automatically synthesizes a strategy of a timed game, keeps the strategy in memory, then verifies and evaluates the strategy on a number of SMC properties. We extended the semantics and algorithms of MC and SMC to apply the synthesized strategy when exploring the state space (for MC) and generating random runs (for SMC).

5.1 Extended Stochastic Semantics

Let $\mathbf{A} = (\mathcal{P}_1|\ldots|\mathcal{P}_n)$ be a network of priced timed automata modelling an environment to be controlled. That is $\mathbf{A}$ may be seen as a timed game with global state space $St = St_1 \times \cdots \times St_n$, and with sets Σ_c and Σ_u of controllable and uncontrollable actions, respectively. Now assume that – using UPPAAL-TIGA– we have synthesized a strategy $s : St \to (\mathbb{R} \times \Sigma_c) \cup \{\lambda\}$ for $\mathbf{A}$ ensuring some desired reachability or safety objective. That is $s(q) = (d, a)$ indicates that the strategy s in state q proposes to perform controllable action a after a delay of d; $s(q) = \lambda$ indicates that the strategy will delay indefinitely until the environment has performed an uncontrollable action. Now we may view the *extended* network:

$$\mathbf{A}^e = (\mathcal{P}_1|\ldots|\mathcal{P}_n|A_s)$$

as a closed *stochastic* network over $\Sigma_u \cup \Sigma_c$, where the components $\mathcal{P}_1, \ldots, \mathcal{P}_n$ have been given delay density functions $\mu^1, \ldots, \mu^n$ and output probability functions $\gamma^1, \ldots, \gamma^n$. Now A_s is a one-state component implementing the strategy s. That is s has delay density function $\mu^s_q = \delta_d$, when $s(q) = (d, a)$

and δ_d is the Dirac delta function with probability mass concentrated at time-point d[1]. Moreover the output probability function γ_q^s for s is given by:

$$\gamma_q^s(b) = \begin{cases} 1 & ; s(q) = (0, a), a = b \\ 0 & ; s(q) = (0, a), a \neq b \\ \bot & ; s(q) = (d, a), d > 0 \\ \bot & ; s(q) = \lambda \end{cases}$$

In this way $\mathbf{A}^e$ may be subject to statistical model checking provided. We extend the capability of UPPAAL-SMC to generate random runs for networks of environment components extended with control strategies.

5.2 Implementation

Fig. D.8 shows the work-flow of Control-SMC. The UPPAAL-TIGA engine receives the timed game model $\mathcal{G}$ and the control objective ϕ. It synthesizes a strategy that is kept in memory if $\mathcal{G}$ is controllable. The strategy can be printed out with the option `-w0`. If the option `-X` is used then subsequent MC or SMC queries ρ_i are checked under this strategy. For the purpose of evaluating performance, the model $\mathcal{G}$ can be extended with costs to $\mathcal{G}'$. These costs are modeled with clocks that must be declared as *hybrid clock*. They are ignored for the purpose of symbolic model-checking (synthesis or MC) and taken into account for SMC. Furthermore, floating-point variables can be used in the same way. These additional variables may not be *active* for the purpose of controlling the behavior.

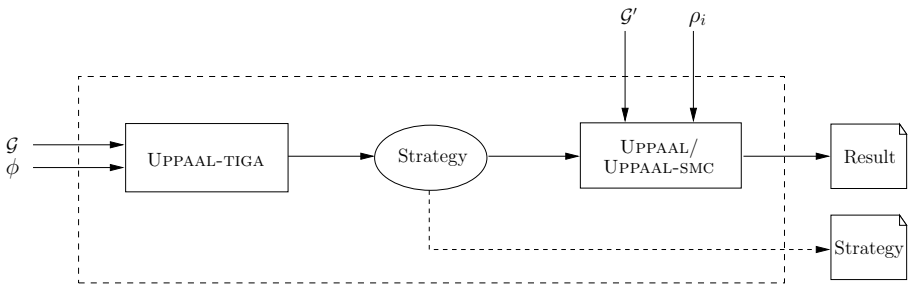

Fig. D.8: Workflow of Control-SMC

The exploration under a given strategy is similar to standard MC or SMC when considering uncontrollable transitions since they are played by an opponent. The opponent is stochastic for the purpose of SMC and when doing MC, all possible successors are tried. However, only the controllable transitions allowed by the strategy are allowed. In addition, delay is constrained by

[1] which should formally be treated as the limit of a sequence of delay density functions with decreasing, non-zero support around d.

the delays of the strategy, e.g., if a controllable transition is to be taken after 5 time units, UPPAAL will not delay more. For SMC, this is resolved naturally through the semantics with a race between components. For the symbolic exploration, the strategy specifies how much delay is allowed and this constrains the standard delay operation. Furthermore, we have to add the upper border of bounded delays to enable following transitions. More precisely UPPAAL-TIGA maintains a partition so we could have the case to wait while in $x \in [0,5[$ and take a transition at $x = 5$, but $x = 5$ is then unreachable. Therefor we have to wait while $x \in [0,5]$. Finally, when an action follows a delay it has an urgent semantics, i.e., the states in which such an action is enabled are not allowed to delay.

5.3 The Running Example

We demonstrates how to use Control-SMC on the running example described in Section 2.2 and 4.2 without the need to translate the strategy. We add prices and stochastic information directly on the TGA `Main` as shown in Fig. D.9. The clocks used for cost are declared as `hybrid clock` (e.g. e), while counters for SMC evaluation are declared as `double` (e.g. s).

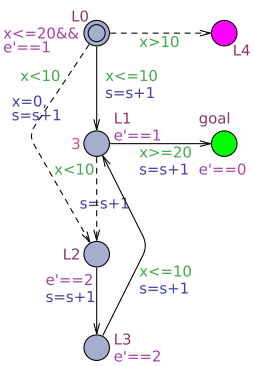

Fig. D.9: TGA Main with Prices

```
control: A<> Main.goal
A<> Main.goal
A<> Main.goal and time<=20
Pr[<=30] (<> Main.goal)
E[<=30;200] (max: Main.s)
E[<=30;200] (max: Main.e)
```

Fig. D.10: Combined Query File

Fig. D.10 shows the query file we use here. A control query that expresses the control objective starts on the first line with a list of MC and SMC queries on the following lines. For now, Control-SMC is available only from the command line checker `verifytga` and is enabled with the option `-X`. Given the model as in Fig. D.9 and the query file as in Fig. D.10 as inputs, it first synthesizes a strategy for the control query, then processes the rest MC and SMC queries in a batch fashion, and gives the same results as in Table D.1 in Section 4.2.

6 Experiments Results

We show the experiments of two case-studies by first using the Control-SMC method of Section 5, then using the strategy translation method in Section 4 as a cross-check. The two methods gave the same results for MC and SMC queries. We also measured the execution time of the queries for both methods, because we want to know the runtime benefit of applying a strategy in time and memory compared with using a translated controller from a strategy. All models in the experiments are available on our SMC web-page[2].

6.1 Case Study 1: Jobshop

The Jobshop problem is about scheduling a set of machines for a set of jobs, where each job needs to use those machines in a particular order for a particular time limit. This case-study involves two professors Kim and Jan who want to read a single piece of four-section newspaper. Each person has his own preferred order on sections, and can spend different times on different sections. The control objective, which is expressed as

```
control: A<> Kim.Done and Jan.Done and time <= 80
```

is to find a strategy that guarantees both people finish reading within 80 time units. UPPAAL-TIGA finds such a strategy. The full explanation about this model can be found on web-page of examples [14]. The model is down-sized for the purpose of the manual conversion to a controller automaton.

Fig. D.11 shows the TGA template with prices for each person for Control-SMC. The availability of four sections are maintained by four global boolean variables. During the initialization of Kim and Jan, the references to the boolean variables are assigned to `sec1` – `sec4` according to each person's preferred order of reading. The strategy tells a person when to acquire a section (controllable, solid edge), while a person can release a section at any time within a time bound (uncontrollable, dashed edge). We add respectively three stop-watches[3] wt, rt and t to measure the accumulated time on waiting, reading and finishing the newspaper respectively.

We obtain the same results when checking the MC and SMC queries in Control-SMC and UPPAAL. Thus in Table D.2 we use the single column Result to show the MC results (`Yes` for satisfied or `No` for not satisfied), and SMC results (probabilities or evaluations). The T (CS) column shows the execution time of a query in seconds by Control-SMC, while the T (M) column shows that by using a manually translated controller. We do not compare the runtime of MC queries because the size of this model is not big enough to make

[2]Section Control-SMC at `http://people.cs.aau.dk/~adavid/smc/cases.html`

[3]Stop-watches are clocks whose rates are reset to zero.

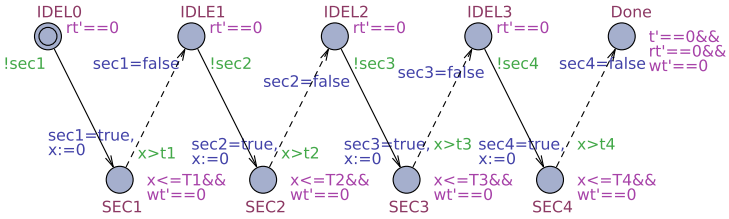

Fig. D.11: Job Template with Prices

Table D.2: MC & SMC Experiments of Jobshop

	#	Queries	Results	T (CS)	T (M)
MC	1	`A[] Jan.Done imply Kim.Done`	`Yes`	-	-
	2	`E<> Kim.Done and Jan.Done and time<=45`	`Yes`	-	-
	3	`E<> Kim.Done and Jan.Done and time<=44`	`No`	-	-
SMC	4	`Pr[<=80] (<> Kim.Done and Jan.Done)`	1^*	76.5	148.3
	5	`E[time<=80;2000000](max:Kim.wt)`	`5.40221`	62.1	132.5
	6	`E[time<=80;2000000](max:Kim.rt)`	`22.7469`	61.7	138.7
	7	`E[time<=80;2000000](max:Jan.wt)`	`11.5652`	60.9	136.8
	8	`E[time<=80;2000000](max:Jan.rt)`	`47.3951`	62.1	138.8

1^* `in [0.999998,1] with confidence 0.95.`

the runtime distinguishable. But we compare the runtime of SMC queries, because we can let the SMC engine to generate a large number of runs to make the runtime difference noticeable.

Experiment 1 shows that Kim always finishes reading before Jan. We get the shortest time (= 45 time units) for both to finish from experiments 2 and 3. Experiment 4 measures the probability for Kim and Jan to finish reading within 80 time units if the probability uncertainty $\epsilon = 0.000001$. In UPPAAL-SMC we can get the plot of probability distribution of this query as shown in Fig. D.12. The plot gives the mean value of around 59 time units. The remaining SMC experiments show the expected time for Kim and Jan to wait and read the newspaper individually. The strategy biases Kim because Kim waits less than Jan. The runtime experiments of SMC queries were carried out on a PC with Intel i7-2640M CPU @ 2.80GHz, 8GB main memory and Ubuntu 12.04

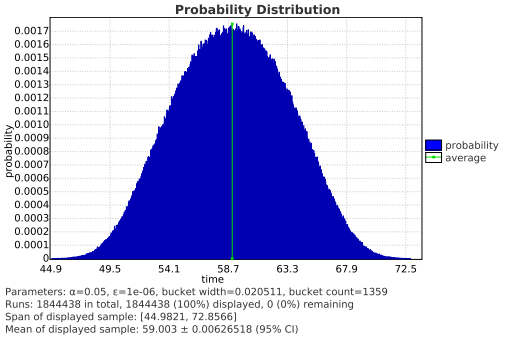

Fig. D.12: Distribution on Time to Finish Reading for Both People

x86_64 with the upcoming version 0.18 of UPPAAL-TIGA. Experiment 4 set $\epsilon = 0.000001$ to force the SMC engine to generate a large number of runs (1844438 runs). In experiments 5 – 8, we set the number of runs to 2000000. We can conclude that applying a strategy in memory improves the performance of SMC engine inside Control-SMC by a factor of two. This is due to the strategy look-up in a hash table instead of simulating it within the model.

6.2 Case Study 2: Train-Gate

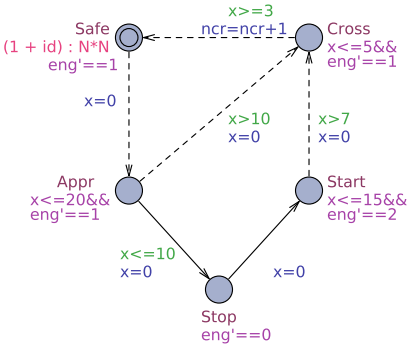

Fig. D.13: Train Template with Prices

Train-Gate is a classical case-study for real-time model checking. It is distributed with UPPAAL with an detailed explanation in [2]. Fig. D.13 shows the game version of it with prices and stochastic extensions. The control objective, which is expressed as

```
control: A[] forall (i : id_t)
forall (j : id_t) Train(i).Cross
and Train(j).Cross imply i == j
```

is finding a strategy to guarantee the exclusive access to `Cross` by two trains. If necessary, the strategy should stop a train at `Appr` in time ($x \leq 10$) by the controllable solid edge to `Stop`, otherwise the train goes to `Cross` directly by the uncontrollable dashed edge. The train can resume at `Stop` by the other controllable solid edge to `Start`. The exponential rate `((1+id):N*N)` appears at `Safe` for specifying the delay density function. A counter *ncr* records the throughput at `Cross`. A hybrid clock *e* measures the energy consumption of a train. The interesting point of this case-study is that we compare the behavior and performance of the synthesized strategy with a manually programmed queue-based controller available in the train-gate example provided in the distribution of UPPAAL.

Table D.3 shows the comparative experiments of the synthesized strategy `Syn` and the queue-based controller `Que`. Experiment 1 shows `Syn` allows `Train(1)` to approach `Cross` while `Train(0)` is still crossing. This is forbidden by `Que`. Experiment 2 measures the probability for `Train(0)` to reach `Cross` within 100 time units with the probability uncertainty $\epsilon = 0.000001$. Experiment 3 shows that `Syn` gives a bigger throughput from `Que`, because `Syn` allows different trains to approach `Cross` concurrently as witnessed by experiment 1. Experiment 4 gives the expected energy consumption for `Train(0)`. We compare the execution time of SMC queries in seconds by

Table D.3: MC & SMC Experiments of Train-Gate

	#	Queries	Result		T (CS)	T (M)
			Syn	Que		
MC	1	`E<>Train(0).Cross&&Train(1).Start`	`Yes`	`No`	-	-
SMC	2	`Pr[<=100](<>Train(0).Cross)`	1*	1*	45.9	88.5
	3	`E[<=100;1000000](max:ncr)`	`8.0665`	`5.8065`	72.3	173.3
	4	`E[<=100;1000000](max:Train(0).eng)`	`124.938`	`88.402`	69.3	169.5

1* in [0.999998,1] with confidence 0.95.

Control-SMC in the T (CS) column with that using a manually translated controller in the T (M) column. In experiment 2, we set $\epsilon = 0.000001$ to force the SMC engine to generate a large number of runs (1844438 runs). In experiments 3 and 4, we set the number of runs to 1000000. We can conclude that applying a strategy in memory improves the performance of SMC engine inside Control-SMC by a factor of two.

7 Future Work

The future work are in three directions. Our first goal is to merge UPPAAL and UPPAAL-TIGA, which will enable Control-SMC from the graphical interface with all its capabilities, in particular the plot composer. Next, we aim to make the clocks for measuring prices in Control-SMC to become real hybrid as in UPPAAL-SMC. The clock rates can be floating-point, negative, or in the form of ordinary differential equations (ODE). The third direction is exploring more potential use of the synthesized strategy in memory. We can try to refine or optimize the strategy using machine learning methods.

Acknowledgments

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

References

[1] K. G. Larsen, P. Pettersson, and W. Yi, "Model-checking for real-time systems," in *FCT*, ser. LNCS, vol. 965. Springer, 1995, pp. 62–88.

[2] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *SFM*, ser. LNCS, vol. 3185. Springer, 2004, pp. 200–236.

[3] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *CAV*, ser. LNCS, vol. 1427. Springer, 1998, pp. 546–550.

[4] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient on-the-fly algorithms for the analysis of timed games," in *CONCUR*, ser. LNCS, vol. 3653. Springer, 2005, pp. 66–80.

[5] H. L. S. Younes, "Planning and verification for stochastic processes with asynchronous events," in *AAAI*. AAAI Press / The MIT Press, 2004, pp. 1001–1002.

[6] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "UPPAAL-TIGA: Time for playing games!" in *CAV*, ser. LNCS, no. 4590. Springer, 2007, pp. 121–125.

[7] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and Z. Wang, "Time for statistical model checking of real-time systems," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 349–355.

[8] F. Cassez, J. J. Jessen, K. G. Larsen, J.-F. Raskin, and P.-A. Reynier, "Automatic synthesis of robust and optimal controllers - an industrial case study," in *HSCC*, ser. LNCS, vol. 5469. Springer, 2009, pp. 90–104.

[9] J. J. Jessen, J. I. Rasmussen, K. G. Larsen, and A. David, "Guided controller synthesis for climate controller using uppaal tiga," in *FORMATS*, ser. LNCS, vol. 4763. Springer, 2007, pp. 227–240.

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

[11] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems (an extended abstract)," in *STACS*, 1995, pp. 229–242.

[12] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets*, ser. LNCS, vol. 3098. Springer, 2003, pp. 87–124.

[13] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, J. van Vliet, and Z. Wang, "Stochastic semantics and statistical model checking for networks of priced timed automata," *CoRR*, vol. abs/1106.3961, 2011.

[14] K. G. Larsen, "Quantitative model checking exercise," http://people.cs.aau.dk/~kgl/QMC2010/exercises/, 2010, 28. Job Shop Scheduling.

ISSN (online): 2246-1248
ISBN (online): 978-87-7112-852-9

AALBORG UNIVERSITY PRESS