FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# Boolean Satisfiability Modulo Differential Equation Simulations

by

*Ing. Tomáš Kolárik*

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Doctoral study programme: Informatics
Department of Digital Design

Prague, January 2024

**Supervisor:**
doc. Dipl.-Ing. Dr. techn. Stefan Ratschan
Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2
182 00 Prague 8
Czech Republic

# Abstract and Contributions

SAT solvers are convenient tools to be used in the area of planning or formal verification. However, in addition to Boolean constraints, it is often important to model physical phenomena, where differential equations are of immense importance. In current industrial practice, the properties of the resulting models are checked by testing using simulation tools. These approaches, however, lack robust computational support of automatic analysis (e.g., verifying) of such models, and do not search the Boolean state space efficiently, as SAT solvers do.

Research on SAT solvers that can handle differential equations has aimed at replacing tests with correctness proofs. However, there are fundamental limitations to such approaches in the form of undecidability, and moreover, the resulting solvers do not scale to problems of the size commonly handled by simulation tools in industry. Also, in many applications, classical mathematical semantics of differential equations often does not correspond well to the actual intended semantics, and hence a correctness proof wrt. mathematical semantics does not ensure the correctness of the intended system.

We head at overcoming those limitations with an alternative approach to handling ordinary differential equations (ODEs) within SAT solvers. This approach is based on the semantics used by tests in simulation tools, but still may result in mathematically precise correctness proofs wrt. that semantics.

Computational experiments confirm the promise of such an approach. In particular, we present a railway scheduling problem that exhibits both non-trivial discrete and continuous behavior and where a number of timing and ordering constraints on the trains can appear. On the contrary, existing benchmark problems for SAT modulo ODE exhibit only fairly trivial discrete state space.

We also introduce a new approach to solving a multi-agent path-finding problem. We exploit conflict generalization techniques using an off-the-shelf SAT solver that also handles linear real arithmetic constraints. Differential equations do not appear in this model, but collision detection and avoidance of the agents are based on simulations where non-linear constraints appear as well.

**Contributions.** The main contributions of the dissertation thesis are the following:

- We propose a new method that exploits efficient SAT solvers and at the same time handles ordinary differential equations (ODEs). In contrast to state-of-the-art approaches to SAT modulo ODE, the semantics of ODEs is based on semantics that are used in simulation tools. This allows scaling the resulting solver to the size of benchmark problems that appear in the industry.
- We introduce a precisely defined Satisfiability Modulo Theories (SMT) language that is expressive and flexible compared to existing approaches.
- We introduce a benchmark problem that comes from the domain of railway scheduling. Unlike the benchmarks used in state-of-the-art approaches, the problem exhibits both non-trivial discrete and continuous behavior.
- We present an approach to a continuous-time version of the multi-agent pathfinding problem. Collision detection and avoidance of the agents yield both linear and non-linear arithmetic constraints. We efficiently handle the non-linear constraints based on simulations and the linear constraints using an off-the-shelf Satisfiability Modulo Theories (SMT) solver. Experiments show that the new approach scales better with computation time than state-of-the-art approaches.

**Keywords:** Boolean satisfiability (SAT), Satisfiability Modulo Theories (SMT), numerical methods for ordinary differential equations (ODEs), simulations, floating-point computation, planning, formal verification, embedded systems.

# Abstrakt

SAT řešiče se ukázaly jako vhodné nástroje v oblasti plánování a formální verifikace. Nicméně, kromě Booleovských omezení je často důležité modelovat také spojité jevy, k čemuž jsou nesmírně důležité diferenciální rovnice. V současnosti platí, že se vlastnosti průmyslových modelů ověřují prostřednictvím testování a simulačních nástrojů. Tyto přístupy však dostatečně neovládají automatickou analýzu takových modelů (např. verifikaci) a neprohledávají Booleovský prostor tak efektivně jako právě SAT řešiče.

Výzkum SAT řešičů, které dokáží zacházet s diferenciálními rovnicemi, se zaměřoval na nahrazení testování důkazy správnosti. Jenomže, takové přístupy trpí zásadními nedostatky, které plynou z nerozhodnutelnosti těchto problémů. Navíc, výsledné řešiče se nedokáží škálovat na problémy o takové velikosti, kterou běžně zvládají simulační nástroje v průmyslu. V mnoha aplikacích je také problém s tím, že klasická matematická sémantika diferenciálních rovnic často neodpovídá zamýšlené sémantice. Z toho však vyplývá, že důkaz správnosti vzhledem k matematické sémantice nezaručuje správnost zamýšleného systému.

Abychom překonali tyto nedostatky, představujeme alternativní přístup k zacházení s obyčejnými diferenciálními rovnicemi v rámci SAT řešičů, který je založen na stejné sémantice, která se používá v testování a v simulačních nástrojích. Stále však platí, že tato metoda může s ohledem na tuto simulační sémantiku dojít k matematicky přesným důkazům správnosti.

Výpočetní experimenty potvrzují, že takový přístup je slibný. To ukazujeme zejména na problému železničního plánování, který vykazuje jak netriviální diskrétní, tak také spojité chování. Rovněž je možné specifikovat řadu časových omezení vlaků a jejich řazení. Naproti tomu existující experimenty pro SAT modulo diferenciální rovnice vykazují jen poměrně triviální diskrétní stavový prostor.

Dále uvádíme nový způsob řešení problému multiagentního plánování. K zobecnění konfliktů agentů využíváme již existující SAT řešič, který také ovládá lineární omezení reálných čísel. Ačkoli se zde nevyskytují diferenciální rovnice, detekce a vyhýbání se kolizím agentů jsou založeny na simulacích, kde se vyskytují také nelineární omezení.

**Klíčová slova:** Booleovská splnitelnost, Satisfiability Modulo Theories (SMT), numerické metody pro obyčejné diferenciální rovnice, simulace, počítaní s pohyblivou řádovou čárkou, plánování, formální verifikace, vestavné systémy.

# Acknowledgements

I would like to express my deep gratitude to my supervisor Stefan Ratschan, for his professional and relentless guidance, and for being a patient and helpful advisor and mentor.

Special thanks go to the staff of the Department of Digital Design, who maintained a pleasant and flexible environment for my research and supported me with their advice and ideas. Another special thanks goes to the department and faculty management for providing funding for my research.

Finally, I would like to express my greatest thanks to my family, to my girlfriend, and to my friends, for their support, patience, care and understanding.

**Funding.** This work has been supported by:

- projects SGS20/211/OHK3/3T/18 and SGS23/208/OHK3/3T/18 of the Student Grant Competition of CTU in Prague,
- institutional support RVO:67985807,
- projects 21-09458S and 22-31346S of the Czech Science Foundation GA ČR,
- projects SVK 55/22/F8, SVK 56/21/F8, SVK 55/20/F8, SVK 50/19/F8 and SVK 62/23/F8 of IEEE and regarding to organization of the Prague Embedded Systems Workshop between years 2019–2023.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Mathematical Symbols

**Number Sets**

| | |
|---|---|
| $\mathbb{B}$ | Set of Booleans (i.e. $\mathbb{B} = \{\top, \bot\}$) |
| $\mathbb{Z}$ | Set of integer numbers |
| $\mathbb{Z}^{>0}$ | Set of positive integer numbers |
| $\mathbb{Z}^{\geq 0}$ | Set of non-negative integer numbers |
| $\mathbb{Q}$ | Set of rational numbers |
| $\mathbb{R}$ | Set of real numbers |
| $\mathbb{F}$ | Set of floating-point numbers |

**Typesetting of Common Mathematical Symbols**

| | |
|---|---|
| $\mathrm{var}$ | Variable $\mathrm{var}$ with a multiple-letter name |
| $\mathit{fun}$ | Function $\mathit{fun}$ with a multiple-letter name |
| $\dot{z}$ | First derivative of function $z$ |
| $(a, b, c)$ | Vector or list of elements $a$, $b$ and $c$ |
| $\boldsymbol{x}$ | Vector or list of particular elements $x^{[j]}$ |
| $\lvert\boldsymbol{x}\rvert$ | Number of elements in vector or list $\boldsymbol{x}$ (or in a set) |
| $x^{[j]}$ | $j$-th element of vector or list $\boldsymbol{x}$ (indexing from 1 to $\lvert\boldsymbol{x}\rvert$) |
| $\boldsymbol{x} \parallel \boldsymbol{y}$ | Concatenation of vectors or lists $\boldsymbol{x}$ and $\boldsymbol{y}$ |
| $\mathbf{var}$ | Vector or list of particular variables $\mathrm{var}^{[j]}$ with multiple-letter names |
| $\boldsymbol{\mathit{fun}}$ | Vector or list of particular functions $\mathit{fun}^{[j]}$ with multiple-letter names |
| $\dot{\boldsymbol{z}}$ | Vector or list of particular first derivative functions $\dot{z}^{[j]}$ |
| $\mathcal{S}$ | Set $S$ |
| $\mathcal{O}(x)$ | The big $\mathcal{O}$ notation |
| $\mathrm{coll.var}$ | Variable $\mathrm{var}$ in collection $\mathrm{coll}$ |

**Mathematical Abbreviations**

| | |
|---|---|
| $\mathcal{S}^n$ | $\mathcal{S} \times \mathcal{S} \times \ldots$ ($n$ times) |
| $[a, b]$ | $\{x \in \mathbb{R} \mid a \leq x \leq b\}$ |
| $(a, b)$ | $\{x \in \mathbb{R} \mid a < x < b\}$ |
| $\textsc{ite}\,(\mathrm{cond},\ a,\ b)$ | $(\mathrm{cond} \Rightarrow a) \land (\neg\mathrm{cond} \Rightarrow b)$ |

# List of Acronyms

**BDF** backward differentiation formulas

**BMC** bounded model checking

**CBS** Conflict-based Search

**CCBS** Continuous-time Conflict-based Search

**CDCL** conflict-driven clause learning

**CNF** conjunctive normal form

**CP** Constraint Programming

**DPLL** Davis–Putnam–Logemann–Loveland

**ETCS** European Train Control System

**ICP** Interval Constraint Propagation

**IVP** initial value problem

**LRA** linear real arithmetic

**MAPF** multi-agent path-finding

**MAPF**$_R$ multi-agent path-finding with continuous time

**MAX-SAT** maximum satisfiability

**NP** nondeterministic polynomial time

| | |
|---|---|
| **ODE** | ordinary differential equation |
| **OMT** | Optimization Modulo Theories |
| **SAT** | Boolean satisfiability |
| **SMT** | Satisfiability Modulo Theories |
| **UN/SOT** | UN/SAT modulo ODEs Not SOT |

CHAPTER 1

# Introduction

The design of cyber-physical systems is increasingly based on models that can be simulated before the actual system even exists. Here, the most natural way of modeling the physical part is based on differential equations. The resulting models can then be simulated using numerical solvers for ordinary differential equations (ODEs) that are used for instance in tools such as Simulink or Xcos. However, the computational support for automatically analyzing such models is still far from being satisfactory. For example, the analysis of industrial models is nowadays often based on exhaustive testing and validated by certification authorities. Still, such approaches often rely on statistical properties of the models, rather than on thorough constraints satisfaction techniques that are widely used within the community of formal methods, such as model checking and verification.

This has been addressed by SAT solvers [47, 58, 66] that do not only offer efficient discrete (i.e., Boolean) reasoning, but that can in addition handle differential equations by integrating interval ODE solvers [98]. These interval solvers are based on classical mathematical solutions of ODEs, not on numerical methods. However, handling ODEs in such a way is extremely difficult, and most related verification problems are undecidable [22]. The resulting SAT modulo ODE solvers can handle impressive benchmark examples, but their size is still quite far away from the size of the problems that may occur in industrial practice. Moreover, existing benchmark problems for SAT modulo ODE do not exhibit complex discrete state space.

A further reason why such tools may be a poor fit to the needs coming from industrial applications is the fact that classical mathematical solutions usually do *not* correctly represent the intended behavior of industrial models [95]. The reason is that the design process of the models is often *not* based on a mathematical analysis of the underlying differential equations, but on the results of numerical simulations, which stem from discretization and floating-point computation. Hence, the output of the simulation tool is the authoritative description of the behavior of the model, *not* traditional mathematical semantics. This holds even in cases when the model was designed based on ODEs corresponding to physical laws ("from first principles"), because even in such cases, the parameters of the model are estimated based on simulations. This may become even more important due to the increasing popularity of data-driven modeling approaches, for example, based on machine learning.

Therefore, the existing SAT modulo ODE approaches prove correctness wrt. semantics that differs from the notion of correctness used during simulation and testing. We overcome this mismatch by formalizing the semantics of SAT modulo ODE based on numerical simulations [75]. Moreover, we develop a benchmark problem [74] that combines a non-trivial propositional part with ODEs.

We also address another restriction of existing SAT modulo ODE approaches. Their support for differential equations has the form of monolithic building blocks that contain a full system of ODEs within which no Boolean reasoning is allowed. In contrast to that, we provide a direct integration of ODEs into a standard Satisfiability Modulo Theories (SMT) framework [14], which results in a tight integration of the syntax of the theory into Boolean formulas, as usual for theories in SMT-LIB [12]. We apply a corresponding algorithm that tightly integrates Boolean satisfiability (SAT) and numeric simulations of differential equations and support our observations by experiments using an implementation that is available online as open-source [72].

Our approach proves itself especially in the field of planning problems or the like, such as transport scheduling or robot path-finding. Precise planning may become to play an important role within upcoming autonomous traffic control systems. These are the problems where searching for a specific goal state is non-trivial, but the state is supposed to exist. Therefore, it is not necessary to explore the entire state space in order to arrive at the result. Still, the state space may be huge and complicated, and the constraints may exhibit a number of potential conflicts that must be avoided. An example is searching for a plan for multiple agents which must not collide with each other. Here, an algorithm such as SAT may make a fundamental difference in the efficiency of the search and of systematic avoidance of the conflicts.

As an example of a planning problem, we present a benchmark that comes from the domain of railway scheduling [74], where we simulate train networks at a low level and where a number of timing and ordering constraints can appear. The problem is inspired by an approach to railway design capacity analysis [91] that combines a SAT solver with a railway simulator, which is however not based on solving differential equations. We show that it is possible to handle specific tasks efficiently even with a general-purpose algorithm such as SAT modulo ODE. A major difficulty lies in modeling the fact that trains sometimes have to switch to a deceleration phase to obey velocity limits. Here, it is non-trivial to predict when such a switch must happen when modeling dynamics based on differential equations.

Railway route planning can also be viewed as a multi-agent path-finding (MAPF) problem [107], where trains are viewed as agents. MAPF [115, 106] is the problem of navigating agents from their start positions to given individual goal positions in a shared environment so that agents do not collide with each other. The environment is usually modeled using a graph. A generalization of standard MAPF which also considers continuous phenomena—multi-agent path-finding with continuous time

(MAPF$_R$) [4]—allows more accurate modeling of the target application problem without introducing denser and larger discretizations. Especially in applications, where agents correspond to robots, it is important to consider graph edges that interconnect vertices corresponding to more distant positions. However, in this area, usually much simpler models of continuous behavior are used [4] than in the case of our railway scheduling. Nonetheless, the resulting plans are often minimized wrt. a given objective function, for example, the sum of lengths of the paths, while in [74] we do not optimize at all. Furthermore, we employ a synchronous model of the trains that identifies each step of the unrolled planning problem with a fixed time period, resulting in too many steps of the unrolling.

We mitigate these drawbacks by solving the MAPF$_R$ problem which we directly translate to an SMT problem using the theory of quantifier-free linear real arithmetic (LRA), which allows us to reason about time in MAPF modeled in a continuous manner. In addition to this, collision detection and avoidance of the agents yields non-linear constraints which we handle based on simulations (i.e., floating-point computation). The simulations do not involve differential equations though and we handle them using an off-the-shelf SMT solver instead of our implementation of SAT modulo ODE. Still, the design of the model allows to increase the complexity of the simulations, for example with ODEs.

State-of-the-art approaches for MAPF$_R$ [4] search for optimal plans. However, in real-world applications, where the formalized MAPF problem results from an approximation of the original application problem, an overly strong emphasis on optimality is often pointless. Moreover, it may result in non-robust plans that are difficult to realize in practice [7]. Hence we aim for a sub-optimal method whose level of optimality can be adapted to the needs in the given application domain. We also differ from the state-of-the-art approaches in that we approach the optimum from above and iterate through collision-free plans. This has the advantage that—after finding its first plan—our method can be interrupted at any time, still producing a collision-free, and hence feasible plan. This anytime behavior is highly desirable in practice [82]. We did experiments comparing our method with the state-of-the-art approaches on three classes of benchmark problems and various numbers of agents. The results show that our method is typically able to solve more instances than existing approaches for high time-outs and less for lower time-outs. Future improvements in computer efficiency will consequently make the method even more competitive.

We also demonstrate that in a similar manner as in the case of planning problems, our SAT modulo ODE method is also efficient when finding a witness of unsafety of a model—provided that the model is indeed *not* safe, because proving safety is currently not our strong side. Among others, we present experiments based on a hybrid system model of inpatient glycemic control of a patient with type 1 diabetes [31], where it is critical to verify that the glycemic controller is safe. The original model contains a few mistakes, which we had to correct. The main problem is that the dynamics can block switching between adjacent modes, leading to unintended safe results, because finding an unsafe state was unsatisfiable. We however show that the corrected model is not safe, contradicting the original results [31] that proved the model to be safe, apparently only due to the mentioned modeling mistakes. To support our statement, we attach a concrete counterexample. We also compare the performance of our method with a state-of-the-art approach to SAT modulo ODE [58]. We show that the run-time of our approach that is based on simulations instead of validated ODE integration is in this particular case, not unexpectedly, much faster.

## 1.1 Related Work

Industrial models of cyber-physical systems can be simulated using numerical solvers for ODEs and by tools such as Simulink or Xcos. However, the computational support for automatically analyzing such models is still far from being satisfactory. Research on SAT solvers that can handle differential equations [47, 58, 66] has aimed at replacing tests with correctness proofs. However, there are fundamental limitations to such approaches in the form of undecidability, and moreover, the resulting solvers do not scale to problems of the size commonly handled by simulation tools in industry. Also, in many applications, classical mathematical semantics of ODEs often does not correspond well to the actual intended semantics, and hence a correctness proof wrt. mathematical semantics does not ensure the correctness of the intended system.

The problem of verifying differential equations wrt. simulation semantics has been addressed before [96, 21], but not in a SAT modulo theory context. Also, floating-point arithmetic has been realized to be an important domain for verification tools [24, 93], resulting in a floating point theory in SMT-LIB. However, this concentrates on the intricacies of floating point arithmetic, which we largely ignore here, and instead concentrate on the handling of ODEs.

The application of SAT and SMT solvers to planning problems is not new [104, 81, 30], usually in the context of temporal and numerical planning—extensions of the classical planning problem with numerical variables. However, in our approach we handle ODEs directly, without relying on those ODEs to have symbolic (or even polynomial) solutions.

**Railway Scheduling.** We are not aware of existing approaches to railway scheduling that are based on SMT with realistic modeling of continuous dynamics. An approach that builds on an ad-hoc combination of SAT and a railway simulator [91] solves the problem of design capacity analysis, which is a related problem to railway scheduling. However, we differ in that our model allows rich timing constraints, including their Boolean combinations, and that our dynamics of trains is an integral, but modifiable part of the model, instead of being hidden in a simulator.

Many other approaches dedicated to railway scheduling exist. Some support only limited precision or work only under certain assumptions, for example, fixed routes, or not taking into account limited track capacity. Some use networks that were transformed from a microscopic level to an aggregated, macroscopic level [108]. Also, probabilistic methods exist [109, 62]. Some approaches are quite accurate, but still ignore some constraints that we take into account. For example, not all combinations of possible train paths are considered [121], or bi-directional tracks are replaced by pairs of one-directional tracks, and simpler train dynamics is used [61].

**Multi-Agent Path-Finding.** State-of-the-art approaches for multi-agent path-finding with continuous time ($\text{MAPF}_R$) such as Continuous-time Conflict-based Search (CCBS) [4], a generalization of Conflict-based Search (CBS) [114] that represents one of the most popular algorithms for MAPF, search for optimal plans. Other existing methods for generalized variants of MAPF with continuous time include variants of Increasing Cost Tree Search (ICTS) [120] where durations of individual actions can be non-unit. The difference from our generalization is that agents do not have an opportunity to wait an arbitrary amount of time but wait times are predefined via discretization. In addition, a more accurate discretization often increases the number of actions, which can lead to an excessively large search space.

Our method for $\text{MAPF}_R$ comes from the stream of compilation-based methods for MAPF, where the MAPF instance is compiled to an instance in a different formalism for which an off-the-shelf efficient solver exists. Solvers based on formalisms such as SAT [118, 117], Constraint Programming (CP) [105, 55], or Mixed-integer Linear Programming (MILP) [79] exist. The advantage of these solvers is that any progress in the solver for the target formalism can be immediately reflected in the MAPF solver that it is based on.

## 1.2 Contributions

We list the main contributions of the dissertation thesis. In cases an item is directly related to a reviewed paper, we attach the corresponding reference.

- In Chapter 4, we propose a new way of incorporating ODEs into SAT followed by a corresponding solver in Chapter 6. The new method exploits efficient SAT solvers and at the same time handles ODEs based on semantics that are used in simulation tools. This allows the modeling of complex dynamic phenomena and scaling the resulting solver to the size of benchmark problems that appear in the industry—in contrast with state-of-the-art approaches to SAT modulo ODE. Moreover, since our solver is based on a SAT solver, this also allows the handling of benchmarks with a high number of Boolean constraints (e.g. planning or verification problems).

  Consecutively, we model particular problems in a precisely defined SMT language. We present the syntax of the theory in Section 4.2 and the input language of our solver in Section 6.3 which roughly follows SMT-LIB standard. The language is expressive and flexible compared to existing approaches. We show that it is possible to solve specific tasks efficiently even with our general-purpose algorithm. Also, in comparison with some ad-hoc methods for specific planning problems, it is possible to easily modify the underlying dynamics of the particular models.

Tomáš Kolárik and Stefan Ratschan. "SAT Modulo Differential Equation Simulations". In: *Tests and Proofs*. Ed. by W. Ahrendt and H. Wehrheim. Vol. 12165. LNCS. Springer, 2020. URL: https://doi.org/10.1007/978-3-030-50995-8_5.

- We implement[1] our method from Chapter 6 using a tight integration of the underlying SAT solver and the numerical simulator which also handles ODEs and invariants. The resulting algorithm is based on the principles of lazy online approaches to SMT with exhaustive theory propagation.

- In Chapter 8, we introduce a benchmark problem that, unlike the benchmarks used in state-of-the-art approaches, exhibits both non-trivial discrete and continuous phenomena. The resulting problem comes from the domain of railway scheduling, where we simulate train networks at a low level and where a number of timing and ordering constraints can appear.

  Tomáš Kolárik and Stefan Ratschan. "Railway Scheduling Using Boolean Satisfiability Modulo Simulations". In: *Formal Methods*. Ed. by Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker. Cham: Springer International Publishing, 2023, pp. 56–73. ISBN: 978-3-031-27481-7. URL: https://doi.org/10.1007/978-3-031-27481-7_5.

- In Chapter 9, we present an algorithm solving a continuous-time version of the multi-agent path-finding problem. We handle simple timing constraints on the agents using an off-the-shelf SMT solver with linear real arithmetic. On the other hand, collision detection and avoidance of the agents yields non-linear constraints which we handle based on simulations (i.e., floating-point computation), similarly to industrial simulation tools. Although the simulations do not involve differential equations, still the design of the model allows to increase the complexity of the simulations (e.g. with ODEs). Experiments show that the new approach scales better concerning the available computation time than state-of-the-art approaches.

  To appear in:
  Tomáš Kolárik, Stefan Ratschan, and Pavel Surynek. "Multi-Agent Path Finding with Continuous Time Using SAT Modulo Linear Real Arithmetic". In: *International Conference on Agents and Artificial Intelligence*. Ed. by Ana Paula Rocha, Luc Steels, and Jaap van den Herik. SCITEPRESS, 2024.
  On the shortlist for the Best Student Paper Award.

[1] https://gitlab.com/Tomaqa/unsot.

## 1.3 Structure of the Thesis

The dissertation thesis is organized into chapters as follows:

- Introduction.
- Chapter 2: *Theoretical Background* provides theoretical information on the problems that create the building blocks of the satisfiability problems that are studied in this thesis.
- Chapter 3: *Existing Algorithms* presents particular approaches to the problems presented in the previous theoretical chapter. However, our discussion of ODE solvers covers only methods used in classical simulation tools.
- Chapter 4: *SAT Modulo Differential Equation Simulations: Definition* presents the target problem of this thesis. Here we aim at a more general engineering audience compared to the original formalization [75]. The resulting formalism is able to model planning or verification tasks which involve complex systems such as cyber-physical systems.
- Chapter 5: *State of the Art Tools* contains an overview of current implementations of existing algorithms presented in Chapter 3 and of sophisticated state-of-the-art tools that may handle problems that are similar to SAT modulo ODE.
- Chapter 6: *SAT Modulo Differential Equation Simulations: Solver* proposes an alternative approach to SAT modulo ODE, based on the definitions in Chapter 4.
- Chapter 7: *Case Studies with ODEs* presents several models that stem from the language defined in Chapter 4, and provides experimental results of the related case studies.
- Chapter 8: *Railway Scheduling* presents another case study in SAT modulo ODE [74], where however the discrete part is non-trivial. The train networks are simulated at a low-level and the formulas may contain a number of timing and ordering constraints on the trains.
- Chapter 9: *Multi-Agent Path-Finding* presents an algorithm solving a continuous-time version of the MAPF problem [77] using an off-the-shelf SMT solver with linear real arithmetic. However, we handle collision detection and avoidance of the agents based on simulations.
- Conclusion.

# Chapter 2

# Theoretical Background

This chapter provides a theoretical background on the problems that create the building blocks of the satisfiability problems and of the case studies that we present in this dissertation thesis. Many of these formalisms and problems are well-known within the community of formal methods.

We start with definitions that involve ordinary differential equations (ODEs) in Section 2.1. In Section 2.2, we show a modeling formalism that concerns hybrid automata. Then, we switch to problems that are based on logical formulas: Boolean satisfiability (SAT) in Section 2.3 and Satisfiability Modulo Theories (SMT) in Section 2.4. Finally, in Section 2.5 we discuss bounded model checking (BMC).

## 2.1 Ordinary Differential Equation

Ordinary differential equations (ODEs) are widely used in natural science (not only) to model the progress of a phenomenon in time. Moreover, they often form a basis of the physical part of industrial models. The expressive power of differential equations is remarkable, but it may be non-trivial to realize or predict their execution.

There is a number of kinds of differential equations, but even the most ordinary ones are sufficient to model complex phenomena, in the form of *systems* (also called schemes) of differential equations, which all share the same independent variable.

**Definition 1.** *A* (first-order autonomous) system *of* $d$ ordinary differential equations (ODEs) *is an expression of the form*

$$\dot{\boldsymbol{f}} = \boldsymbol{G}(\boldsymbol{f})$$

*where* $\boldsymbol{f}$ *denotes* $d$ *unknown differentiable functions (dependent variables) of a shared independent variable* $t \in \mathbb{R}$ *(usually time) and* $\dot{\boldsymbol{f}}$ *their* first derivatives *in* $t$*, and* $\boldsymbol{G}\colon \mathbb{R}^d \to \mathbb{R}^d$ *are functions that are Lipschitz [22, 63] on* $\boldsymbol{f}$*.*

*Functions* $\boldsymbol{f}\colon \mathbb{R} \to \mathbb{R}^d$ *are a* solution *of the system if for all* $t$*,* $\dot{\boldsymbol{f}}(t) = \boldsymbol{G}(\boldsymbol{f}(t))$*.*

Hereafter, we will assume that all ODEs are of first order[1] and autonomous, and that the independent variable models *time*. Since the definition involves systems of ODEs, the fact that we require ODEs to be of first order is actually no qualitative restriction, because higher-order derivatives can be transformed into systems of first-order ODEs [63]. Autonomous ODEs restrict the equations s.t. they are independent of $t$, that is, functions $\boldsymbol{G}$ do not contain $t$ as an argument. Note that the definition also does not allow derivatives of functions to appear in $\boldsymbol{G}$. ODEs that forbid derivatives to appear as arguments of other functions are called *explicit* ODEs. The assumption that functions $\boldsymbol{G}$ are Lipschitz (i.e. they satisfy Lipschitz condition) is important as it guarantees that the solution exists and is unique[2]. For simplicity, we consider only cases with functions defined everywhere on $\mathbb{R}$.

*Remark* 1. The fact that we require ODEs to be autonomous (i.e. independent of time $t$) is not a restriction, because time can always be substituted by including an auxiliary function $u$ into $\boldsymbol{f}$ defined s.t. $\dot{u} = 1$. Then, for example, autonomous $\dot{f} = u$ is equivalent to non-autonomous $\dot{f} = t$.

Hence, if not stated otherwise, we will not distinguish between the independent variable and such an auxiliary function within our systems of ODEs, and will implicitly assume that function $t$ models time.

**Example 2.1.** We present an example of a simple system of (first order, autonomous, explicit) ODEs, but written in the following form:

$$\begin{aligned}\dot{x} &= v \\ \dot{v} &= -g\end{aligned} \tag{2.1}$$

where $x$ and $v$ model vertical position and velocity of a falling object, respectively, and $g$ is the constant of the gravity of Earth. Note that the left hand side of the equations contain just first derivatives (*first order* ODEs), and that the right hand side of the equations does not contain $t$ (*autonomous* ODEs) nor derivatives (*explicit* ODEs). Also, note that the system is equivalent to just a single second-order ODE $\ddot{x} = -g$.

Formula 2.1 can be formulated in terms of Definition 1 as follows: $\boldsymbol{f} = (x, v)$, that is, $f^{[1]} = x$ and $f^{[2]} = v$, and $\boldsymbol{G}(\boldsymbol{f}) = (v, -g)$, that is, $G^{[1]}(\boldsymbol{f}) = v$ and $G^{[2]}(\boldsymbol{f}) = -g$. The example shows that constants (here $g$) are not forbidden within $\boldsymbol{G}$, and also that functions and constants are not explicitly distinguished by using time $t$ as the argument value of the functions (e.g. $v(t)$).

[1]A higher-order ODE contains a higher-order derivative, which is a consecutive repetition of first derivatives of a function (since a derivative results in a function again, if the derivative exists). For example, $\ddot{f}$ stands for second derivative [63] of $f$ which corresponds to the first derivative of the first derivative of $f$.

[2]Proof of the theorem [22, 63] is based on Picard iterations.

It is usually necessary to provide initial values for ODEs. Initial values are typically required to be scalar numbers but can be, for example, also specified in the form of intervals [40]. Henceforth, we will assume only scalar initial values, unless stated otherwise.

**Definition 2.** *An* initial value problem (IVP) *[22] is a system of* ODEs *that in addition all have fixed initial values:*

$$\boldsymbol{f}(t_0) = \boldsymbol{f_0}$$

*where* $t_0 \in \mathbb{R}$ *is an* initial value of time *and* $\boldsymbol{f_0} \in \mathbb{R}^d$ *are* initial values *of the functions.*

*Remark* 2. If an IVP has a solution, then the solution satisfies the following integral equation:

$$\boldsymbol{f}(t) = \boldsymbol{f_0} + \int_{t_0}^{t} \boldsymbol{G}(\boldsymbol{f}(u))\ \mathrm{d}u.$$

---

**Example 2.2.** To make the system from Example 2.1 an IVP, one must also provide initial values of the functions and also of the independent variable, for example:

$$t_0 \mapsto 0, \qquad x(t_0) \mapsto 10, \quad v(t_0) \mapsto 0 \tag{2.2}$$

which states that the object starts falling from height 10, with no velocity at the beginning. In terms of Definition 2, this corresponds to $t_0 = 0$ and $\boldsymbol{f_0} = (10, 0)$.

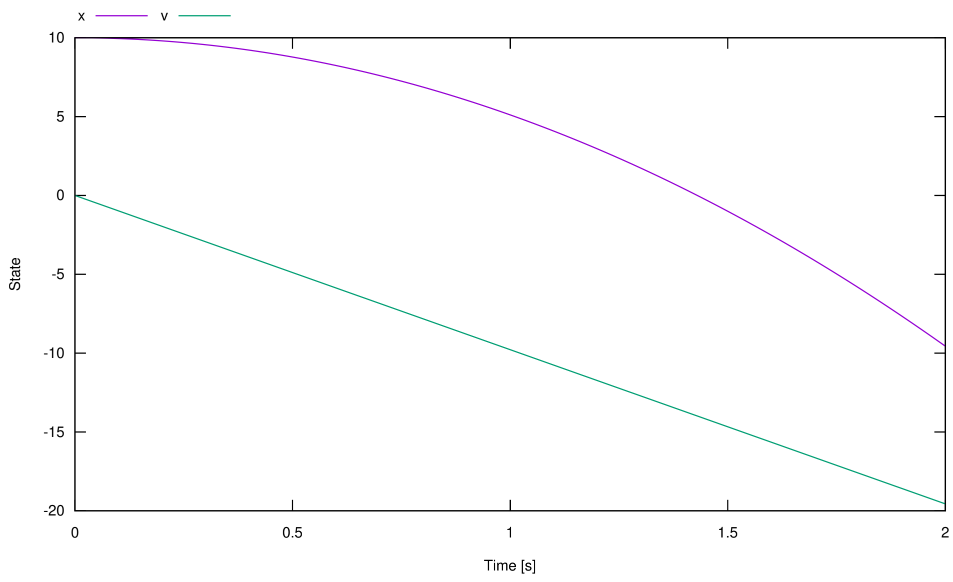


Figure 2.1: Solution of a system of ODEs modeling a falling object for $t \in [0, 2]$.

Note that the system does not define any final conditions, that is, the resulting functions would have infinite lengths. A typical approach is not only to fix the initial value of time, but also the final value, such as the time $t$ in Remark 2. An example of resulting trajectories of the system for $t \in [0, 2]$ is shown in Figure 2.1. Notice that the vertical position $x$ of the object is eventually negative, which may mean that it appears below the ground—which is probably not an intended behavior. However, it is generally non-trivial to estimate the proper value of time, say $\hat{t}$, such that $x(\hat{t}) = 0$ (in cases where Formula 2.1 would be more complicated).

Note that scalar variables can be used in the form of constant functions within $\boldsymbol{f}$ in Definition 1 and Definition 2. For example, constant $g$ in Formula 2.1 may be replaced by a function defined s.t. $\dot{g} = 0$, by extending $\boldsymbol{f}$ to $(x, v, g)$, and by adding e.g. $g(t_0) \mapsto 9.81$ into Formula 2.2.

---

### 2.1.1 Simulation Semantics of ODEs

Common semantics of differential equations are naturally based on mathematical analysis, that is, mathematical solutions of differential equations. We call such semantics *classical mathematical* semantics. An issue is that decision problems (like reachability of a region) related to such semantics are often undecidable[3] and hardly usable in practice.

According to the discussion presented in the introduction chapter, we rather focus on so-called *simulation* semantics, which solve the underlying equations *numerically*.

We treat the following terms as synonyms: simulation semantics of ODEs, ODE simulations, numerical solution of ODEs. Another synonym for solving ODEs numerically is *numerical integration*, where the motivation for the word integration refers to Remark 2.

A difference between classical mathematical solutions and simulations of ODEs lies in discretization and (usually) floating-point computation. There are several variants of floating-point arithmetic, like IEEE 754 arithmetic with a given bit precision (e.g. 64). Moreover, there are many numerical methods for simulating ODEs. Floating-point arithmetics often define special values, like infinities, or Not-A-Number (which represents e.g. $\frac{0}{0}$). Robust algorithms must handle such cases explicitly.

Simulation semantics, for example [96, 21], can result in a much higher performance compared to analytic solutions. However, floating-point operations usually aggregate rounding errors, which are propagated up to the solutions of the ODEs, which are likely to be different from the exact mathematical solutions. Another type of errors is related to the approximation of the integration itself, which is usually more significant than the rounding errors. Simulation methods usually do not guarantee precise bounds of the reached error, only its asymptotic convergence.

Nevertheless, simulation semantics often correspond to the behavior of industrial cyber-physical systems *better* than mathematical semantics do [95]. The reason is that the whole design process of such systems is usually tightly connected to their corresponding models that are based on the results of numerical simulations. The engineers

[3] A more rigorous discussion of this topic is out of the scope of this dissertation thesis, see [22] for more details.

have already created and thoroughly verified such models based on their experiments, and we can use the models and trust that they reflect realistic behavior. In the case of mathematical semantics, on the other hand, we do not have enough evidence that such models reflect real-world systems, unless they depend only on elementary physical laws—which is not the case of complex industrial systems.

Simulation semantics may be highly *parametric*, including a chosen arithmetic and an integration method, along with all corresponding parameters. Such methods can differ a lot, and there is no "best" method that dominates all the others—various problems may require different parameters. A deeper investigation of these methods follows in Chapter 3.

## 2.2 Hybrid Automata

ODEs are useful for modeling continuous phenomena. However, it may also be necessary to model discrete behavior of a real-world system. To achieve that, a possible way is to adapt classical discrete automata theory in the context of differential equations. There are many ways how to embed continuous time into discrete automata, for example timed automata, hybrid automata, etc. A *hybrid automaton* (e.g. [85, 90]) is a possible formalism that is expressive enough to also describe differential equations. Although hybrid automata are not the subject of this dissertation thesis, they serve as an illustrative example of an alternative approach to model cyber-physical systems. We present them only informally.

A hybrid automaton is represented by a directed graph which is extended with continuous variables. The graph has a finite number of vertices, called locations. There is always exactly one active location. Dynamics associated with the variables are described by ODEs and consists of guarded discrete transitions between locations of the automaton that can reset some variables. Each location is associated to a set of so-called *invariants*, which are constraints on continuous variables that must hold at all times, as long as the location is active. Continuous variables are actually functions of time, but are treated as first-order objects.

Initial conditions are given by a set of possible initial locations and by initial values of all the variables. A typical property of such an automaton that one may want to verify is reachability of a set of states [22], which is however an undecidable problem. We assume no explicit inputs or outputs of hybrid automata that would be related to the visited locations.

---

**Example 2.3.** A simplified example of a hybrid automaton is shown in Figure 2.2, with the same dynamics as in Example 2.1, but in addition with invariants and a reset of variable $v$. It models a bouncing ball where location *down* models fall of the ball and the location switches to *up* as soon as the ball reaches the ground. Then the location models bounce of the ball with possibly imperfect elasticity, which may reduce the vertical velocity, corresponding to the reset with $K \in [0, 1]$ which determines the elasticity.

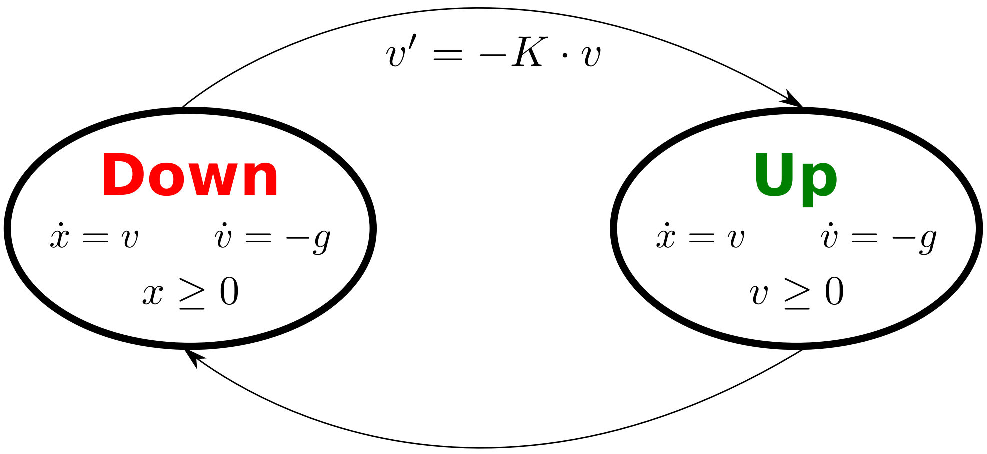


Figure 2.2: Simplified hybrid automaton of a bouncing ball model.

Invariant $x \geq 0$ guarantees that the ball does not appear below the ground (in contrast to Example 2.2 and Figure 2.1).

Initial conditions are missing in the figure: neither initial location nor initial values of variables $x$ and $v$ is provided. It is not unusual to provide both initial and so-called goal conditions separately from the model (i.e. the automaton). This way, such a model can be used in different contexts and can result in various verification (reachability) tasks. For example, given the automaton and initial conditions corresponding to Example 2.2, that is, initial value of $x$ maps to 10 and initial value of $v$ maps to 0, and starting from location *down*, a task can be to check if a state with active location *up* and with $x \geq 6$ is reachable.

One might also want to, for example, define a value for $K$. Actually, it may be more useful to use some small interval instead of concrete value. The same applies to the initial value of the continuous variables, which can also be desirable to be defined with some flexibility, that is, in the form of intervals, instead of exact values. However, such intervals are not discussed here.

Figure 2.3 shows an example of resulting trajectories of the automaton with the mentioned initial conditions after 2 transitions between the locations. The elasticity of the bounces of the ball is imperfect here ($K = 0.8$), so the height that is reached after the bounce is lower than the initial height. As a result, it is true that, for example, a state with active location *up* and with $x \geq 6$ is reachable—it goes up to $x = 6.4$ in the second stage, which corresponds to location *up*.

---

With hybrid automata, Boolean state space is represented in the form of a graph. Most verification tools that are based on hybrid automata (e.g. [53]) use graph algorithms for reachability analysis and the like. The model might be divided into multiple simpler automata connected in the form of *compositions*. Using compositions can enhance readability of the resulting model and also, with increasing number of Boolean constraints in the model, it can overcome the exponential growth of the number of nec-

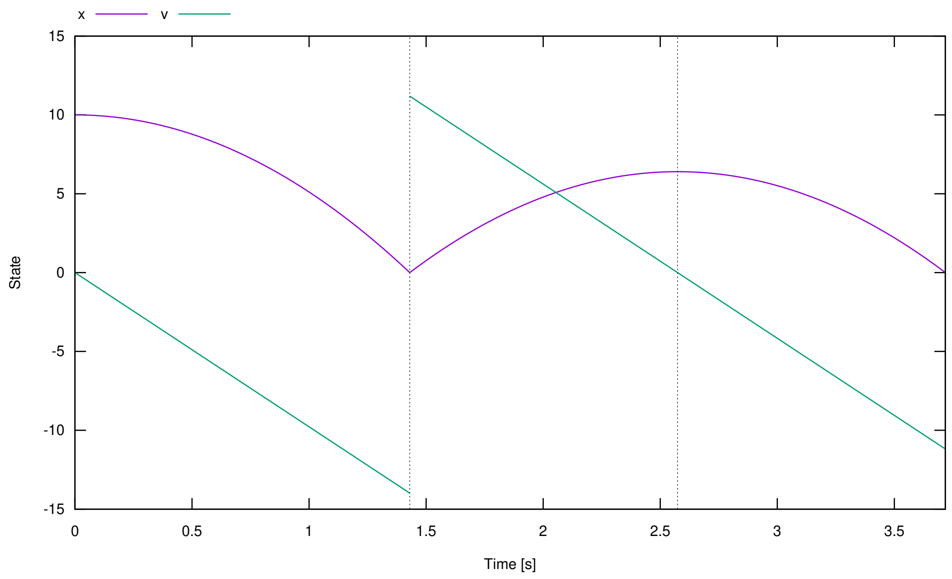


Figure 2.3: Solution of three stages of ODEs modeling a bounce of a ball with $K = 0.8$.

essary locations. For example, the bouncing ball model can be extended of horizontal movement, again with the possibility of bounces, analogously to how vertical dynamics is modeled. Instead of creating an automaton with four locations to cover all the possibilities (*down-left*, *down-right*, ...), it might be better to make a composition of the existing automaton and a new one with locations *left* and *right*, where the horizontal dynamics would be defined separately.

## 2.3 Boolean Satisfiability Problem

The Boolean satisfiability (SAT) problem is a well known nondeterministic polynomial time (NP) complete problem[4], deeply investigated and implemented in a number of very efficient solvers that are also used in practice. Although there is no known algorithm that solves this problem with an asymptotic worst-case complexity better than exponential, most practical instances are efficiently solvable by state-of-the-art algorithms because the hardest possible instances are rare.

**Definition 3.** Boolean satisfiability *is the problem of deciding whether there is an assignment of Boolean values to $n$ variables $\boldsymbol{b} = (b^{[1]}, \dots, b^{[n]})$ that* satisfies *a quantifier-free propositional (Boolean) formula $\phi$.*

A Boolean literal is either a Boolean variable or its negation. We will also respectively say that a Boolean literal is negative or positive iff it does or does not represent

[4]It is the first NP problem which was proved that any NP problem can be translated into it in polynomial time [37].

a negation. Formula $\phi$ is typically in conjunctive normal form (CNF), that is, conjunction of clauses which are disjunctions of Boolean literals. This may be an important fact for huge instances, because sometimes encoding a problem into such a formula can be a bit time-consuming.

A satisfiable assignment of variables $\boldsymbol{b}$ may be called a *model*, but we will not use this keyword to avoid confusion with another meaning of a model, which is related to modeling and is more relevant in this dissertation thesis. Note that computation of a satisfiable assignment might be generally more difficult than just checking satisfiability.

There is a plethora of algorithms and techniques that are used by state-of-the-art SAT solvers. Their basic overview follows in Chapter 3.

### 2.3.1 Maximum Satisfiability Problem

There is a generalization of SAT—maximum satisfiability (MAX-SAT), which searches for an assignment that satisfies not necessarily all clauses of a formula in CNF, but the maximum number of clauses. We also distinguish two kinds of clauses: *hard* and *soft* clauses. Then, the number of satisfied soft clauses is optimized, and all hard clauses must be satisfied in any way. This stands for so-called *partial* MAX-SAT. Classical SAT is then a special case where there are no soft clauses.

Another generalization is *weighted* (partial) MAX-SAT, where each soft clause is also assigned to a non-negative weight, and the goal is to arrive at a solution with maximum total weight. Non-weighted variant is then a special case where each soft clause is assigned to weight 1. We will henceforth assume the weighted partial variant of the problem.

Therefore, it is possible to handle some optimization problems too, not only decision problems.

## 2.4 Satisfiability Modulo Theories

Plain SAT is often effectively insufficient or inefficient for describing or deciding formulas that involve operations or variables that are more complex than the Boolean ones, like arithmetic. Satisfiability Modulo Theories (SMT) is a generalization of SAT where it is possible to decide more complex formulas. As the name states, it is based on various predicate logical *theories* [14, 23]. Some theories are focused on a certain number set (e.g. integers, reals), or a data structure (e.g. array). Some even allow quantifiers. As different theories can often be combined with each other, the expressive power offered by SMT instances can be impressive, and yet efficient.

Each theory has a syntactical and a semantical part. We will use simplified definitions from [14].

**Definition 4.** *A* theory *is defined in first-order logic by:*

- *a* signature*—a set of constant, function and predicate symbols without semantics (i.e. the syntax of the theory),*
- *an* interpretation*—a mapping of function and predicate symbols to semantically defined functions and relations, respectively.*

*A* fragment *of a theory is a syntactically-restricted subset of formulas of the theory.*

Note that other formalisms, such as [23], may define the semantics of theories in a different way, for example, based on axioms[5].

There is a number of standardized theories [12], covering real numbers, integer numbers, floating-point numbers, bit-vectors, or arrays. Recall that SAT is defined just on propositional logic, a subset of first-order logic.

### 2.4.1 Real Arithmetic

**Definition 5.** *The theory of* real arithmetic *[14] is defined by*

- *the signature with all rational constants, function symbols $\{+, -, \cdot\}$ and predicate symbols $\{=, \geq\}$, all with the usual arity,*
- *the interpretation that gives all the symbols the usual meaning, treating arguments as real numbers $\mathbb{R}$.*

The theory is decidable with double exponential asymptotic complexity [23], regardless whether the theory is fragmented with no quantifiers, or even if only conjunctions of literals are permitted.

Typical fragments of the theory are quantifier-free fragments and linear fragment (i.e. *linear real arithmetic*) that does not allow the multiplication of real variables (only with constants), which is in practice much easier to handle. Another common fragment is the difference logic (see below).

In the case of [23], the theory is defined as the Theory of Reals, using 17 axioms which cover the properties of real closed fields with a total order of $\geq$.

---

**Example 2.4.** An example [14] of a formula in linear real arithmetic is

$$x < y + z \quad \wedge \quad x - y = z - 2w \quad \wedge \quad y < 0 \quad \wedge \quad w < y.$$

After substitution of $x$, the formula looks as follows:

$$y + z - 2w < y + z \quad \wedge \quad y < 0 \quad \wedge \quad w < y$$

which is equivalent to

$$0 < w < y < 0$$

which is unsatisfiable due to $0 < 0$.

---

[5]Axioms are closed formulas in first-order logic that contain purely symbols from the syntax of the theory and define the semantics entirely.

**Difference Logic.** An interesting example of a fragment of the real arithmetic is the *real difference logic* [14], where all atomic theory formulas are of the form

$$x - y \bowtie c$$

where $x, y$ are variables, $c$ is constant, and $\bowtie \in \{=, \leq, \geq\}$. The background theory of a difference logic can also be other theories than the real arithmetic, for example the integer arithmetic.

Such formalism is too restrictive for many applications. However, very efficient algorithms exist [100] for problems that can be encoded using only these constraints.

### 2.4.2 Optimization Modulo Theories

An optimizing variant of the problem—Optimization Modulo Theories (OMT) [99, 110, 119]—generally combines the principles of MAX-SAT along with minimization of a cost function, which is for example a linear function of arithmetic variables (e.g. sum of given real variables in the case of the real arithmetic).

Such a framework already offers to model a wide range of problems, thanks to handling hard combinatorial constraints alongside the possibility of arithmetic optimization. Nevertheless, the resulting problems can of course be very difficult to solve, especially when a formula contains complicated theory constraints, such as non-linear arithmetic constraints.

An interesting aspect of optimization is *approximation*. With no approximation, the solution is required to be a precise optimum. This may or may not be important: the optimal solution guarantees the best possible result, which can on the other hand be time-consuming to compute. The thing is that a solution that is quite close to the optimal could be found much faster—the resources that would be necessary to arrive at just a little bit better solution are sometimes not worth the effort. In such a case, a possible strategy can be to search for an approximation of the optimum—a *suboptimal* solution. We can in addition guarantee that the result is not worse than, for example, a user-specified factor of the optimum, which we call a *bounded suboptimal* solution. Such an approach is not wide-spread within SMT community, but is for example common in the field of multi-agent path-finding [116, 83].

## 2.5 Bounded Model Checking

In the field of model checking (explicit or symbolic), bounded model checking (BMC) is a way of discovering undesired properties in a transition system by checking all possible paths from a set of initial states up to a bounded length of the paths. A transition system is defined by a triplet $(S, S_0, R)$ where $S$ is a set of all possible states, $S_0 \subseteq S$ is a set of initial states and $R \subseteq S \times S$ is a transition relation which defines possible successor states for each state of $S$. A path of a transition system is a sequence of states starting with an initial state and respecting the transition relation. Here we suppose that all paths of all transition systems are infinite, meaning that for all states at least one possible next state must be defined.

The idea of BMC is that any violation of a property within a *bounded* path implies that the property does not hold even for *infinite* paths [15]. In other words, if $BMC(T, \phi, n)$ stands for a BMC algorithm, which returns $\perp$ if a property $\phi$ can be violated in a transition system $T$ within any path of length $n$, and returns $\top$ otherwise, and a formula $\Phi_T$ holds iff $\phi$ holds for any infinite path of $T$, then

$$\neg BMC(T, \phi, n) \implies \neg \Phi_T. \tag{2.3}$$

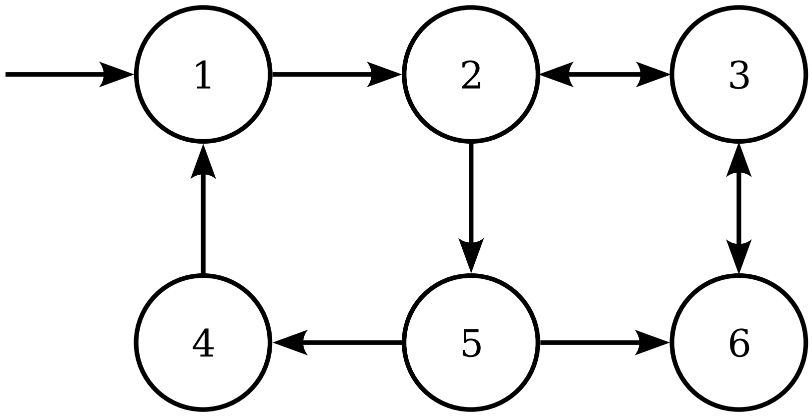


Figure 2.4: An example of a transition system.

**Example 2.5.** Figure 2.4 shows an example of a simple transition system $T = (S, S_0, R)$ with $S = \{1, 2, 3, 4, 5, 6\}$, $S_0 = \{1\}$ and $R = \{(1, 2), (2, 3), (2, 5), (3, 2), (3, 6), (4, 1), (5, 4), (5, 6), (6, 3)\}$. Let $V$ and $E$ be additional sets of states s.t. $V = \{1, 2, 4, 5\}$ and $E = \{6\}$. Let $\phi$ be a property of transition system $T$ that holds iff entering a state from $E$ implies that the previous state was from $V$.

$BMC(T, \phi, 3)$ holds, because all possible paths in $T$ of length 3 are $(1, 2, 3)$ and $(1, 2, 5)$, so set $E$ (i.e. state 6) cannot even be reached. Still, $BMC(T, \phi, 3)$ does *not* imply that property $\phi$ holds—the implication works only in the opposite way (Formula 2.3).

In the case of $BMC(T, \phi, 4)$, all possible paths of length 4 are $(1, 2, 3, 2), (1, 2, 3, 6), (1, 2, 5, 4), (1, 2, 5, 6)$. Only paths $(1, 2, 3, 6)$ and $(1, 2, 5, 6)$ reach state 6. Path $(1, 2, 5, 6)$ does not violate the property, but path $(1, 2, 3, 6)$ does, because the predecessor of state 6 is state 3 and $3 \notin V$. Therefore, $BMC(T, \phi, 4)$ does not hold, for which only a single path violating the property is sufficient. Following Formula 2.3, property $\phi$ does not hold even for infinite paths.

The example refers to explicit model checking, which is based on classical automata and graph algorithms. In such a case, BMC is a form of graph-searching procedures and usually relates to the reachability of a state.

Symbolic model checking does not model all the states explicitly, but uses symbolic abstraction of some (or all) states. SAT can be used as a possible way of symbolic model checking, where the transition system of a model is represented by a Boolean formula, instead of a graph. In this context, a BMC algorithm can be described as $BMC(T_n, \phi_n)$, where $T_n$ is the formula that represents the behavior of the model, including the transition system in the form of all initial states and the transition relation unrolled $n$ times; and $\phi_n$ is the property of our interest. Such an algorithm then forms $\psi_n \coloneqq (T_n \wedge \neg\phi_n)$ and returns $\top$ iff formula $\psi_n$ is *unsatisfiable*[6], which means that there is no path of length $n$ that satisfies $\neg\phi_n$ (no counterexample to $\phi_n$ was found).

---

**Example 2.6.** Let $T = \big(\{\boldsymbol{v} \in \mathbb{B}^2 \mid v^{[1]} \wedge v^{[2]}\}, \{(\boldsymbol{v}, \boldsymbol{v'}) \in \mathbb{B}^2 \times \mathbb{B}^2 \mid R(\boldsymbol{v}, \boldsymbol{v'})\}\big)$ be a simple transition system with an initial state corresponding to $(\top, \top)$ and with transition relation defined s.t.

$$R(\boldsymbol{v}, \boldsymbol{v'}) \Leftrightarrow \Bigg( \begin{array}{rlcl} & \big((v^{[1]} \wedge v^{[2]}) & \Rightarrow & (v'^{[1]} \Leftrightarrow \neg v'^{[2]})\big) \\ \wedge & \big((v^{[1]} \Leftrightarrow \neg v^{[2]}) & \Rightarrow & \neg(v'^{[1]} \wedge v'^{[2]})\big) \\ \wedge & \big((\neg v^{[1]} \wedge \neg v^{[2]}) & \Rightarrow & (\neg v'^{[1]} \wedge \neg v'^{[2]})\big) \end{array} \Bigg)$$

which means that $(\top, \top)$ has two equivalent successor states corresponding to $(\top, \bot)$ and $(\bot, \top)$, from where any transition except of back to $(\top, \top)$ can be taken, and then there is an absorbing state $(\bot, \bot)$. $T_1$ corresponds to $v_1^{[1]} \wedge v_1^{[2]}$, $T_2$ corresponds to $T_1 \wedge R(\boldsymbol{v}_1, \boldsymbol{v}_2)$, $T_3$ to $T_2 \wedge R(\boldsymbol{v}_2, \boldsymbol{v}_3)$, etc. Let the unreachability of $(\bot, \bot)$ after $n-1$ steps be the observed property, thus $\neg\phi_n \Leftrightarrow (\neg v_n^{[1]} \wedge \neg v_n^{[2]})$, and $\phi_n \Leftrightarrow (v_n^{[1]} \vee v_n^{[2]})$. Then, both $\psi_1$ and $\psi_2$ are unsatisfiable, and $BMC(T_1, \phi_1) \wedge BMC(T_2, \phi_2)$ holds, because $(\bot, \bot)$ cannot be reached after less than two transitions. However, $\psi_3$ is already satisfiable, $BMC(T_3, \phi_3)$ returns $\bot$, and a witness is $\{\boldsymbol{v}_1 \mapsto (\top, \top), \boldsymbol{v}_2 \mapsto (\bot, \top), \boldsymbol{v}_3 \mapsto (\bot, \bot)\}$. All $BMC(T_n, \phi_n)$, $n > 3$ return $\bot$ too, and as a result, it is not true, that state $(\bot, \bot)$ is unreachable.

---

Hybrid automaton (Section 2.2), or for example also timed automaton, is an interesting example of a modeling formalism in the field of model checking. Model checking algorithms that analyze such automata cannot simply rely on explicit model checking—this is sufficient for a graph representation of the locations, but they need to represent real variables symbolically. The resulting algorithms can be a combination of both explicit and symbolic model checking techniques.

Applying bounded model checking on a hybrid automaton can result in reachability analysis: The task is to decide whether certain states are reachable, where the states correspond to locations after $n$ transitions from the initial locations of the automaton.

[6]Note that $T_n$ alone is usually supposed to be satisfiable, otherwise it likely means that the model is corrupted and contains mistakes.

CHAPTER 3

# Existing Algorithms

This chapter presents existing algorithms that are related to some of the problems presented in Chapter 2. We will aim to describe those areas that are necessary for solving satisfiability problems where ordinary differential equation (ODE) appear as well. However, our discussion of ODE solvers covers only methods used in classical simulation tools.

First, we survey numerical methods for solving ordinary differential equations (ODEs) in Section 3.1. Then, sections related to satisfiability problems follow: Section 3.2 refers to Boolean satisfiability (SAT), and Section 3.3 refers to Satisfiability Modulo Theories (SMT).

## 3.1 Numeric Solving of ODEs

According to the discussion in the introduction and in Section 2.1.1, this dissertation thesis investigates only numeric approaches to solving ODEs (i.e. simulation semantics), and not the ones that are based on mathematical solutions of ODEs. As a result, scalars are of a sort $\mathbb{F}$, not $\mathbb{R}$.

Simulation methods are *stepping* methods, where there are *step sizes* between sample points of the discretized approximate computation [6]. This means that the functions of underlying ODEs are represented as *sequences*. These methods are *iterative*, where each sample point depends on a limited number of previous points. Generally, the shorter the step sizes are, the closer is the resulting trajectory to the mathematical solution of an ODE, but the higher computational complexity is required too.

*Notation.* $t_i$ is the value of time $t$ at $i$-th sample point (after $i-1$ steps from the initial point). $h_i \coloneqq t_{i+1} - t_i$ is the step size between $i$-th and $(i+1)$-th sample points, and it is abbreviated as just $h$ if it is constant (i.e. an equidistance, the same for all $i$). The last sample point is $n$-th sample point. Functions $\boldsymbol{f}$ and $\boldsymbol{G}$ correspond to Definition 1. $\boldsymbol{f}_i \approx \boldsymbol{f}(t_i)$ are the approximated function values at $i$-th sample point. $\boldsymbol{g}_i \coloneqq \boldsymbol{G}(\boldsymbol{f}_i)$ is an abbreviation for function values of $\boldsymbol{G}$.

Recall that $|\boldsymbol{f}| = |\boldsymbol{G}| = d$.

*Remark* 3. We assume autonomous systems of ODEs, which is not an unusual practice. Functions $\boldsymbol{f}$ and $\boldsymbol{G}$ are allowed to contain the time function $t$ (Remark 1) though. For example, if $f^{[1]} = t$ and $G^{[1]} = 1$, then $f_i^{[1]} \approx f^{[1]}(t_i) = t(t_i) = t_i$ and $g_i^{[1]} = G^{[1]}(t_i) = 1$. Such a setup works just fine for a number of stepping methods even for non-autonomous systems, but not in general. For simplicity, we will again ignore the time arguments of the functions, mainly because we do not consider non-autonomous systems to be important in the case of models that appear in this dissertation thesis.

A typical output of stepping methods is a list of pairs $(t_i, \boldsymbol{f}_i)$, $i \in \{0, \ldots, n\}$, $n \in \mathbb{Z}^{>0}$ which represents the resulting trajectory. In general, any of $n$, $t_n$ and $\boldsymbol{f}_n$ may be unknown before the computation starts—in contrary to many classical implementations of ODE solvers which do not support final conditions in a more general sense such as those that we will present in Section 4.1.

Stepping methods are either *explicit* or *implicit*. Explicit methods use only the already computed sample points. Implicit methods, on the other hand, depend also on the values that are not known so far, which can be resolved by solving an algebraic equation.

**Example 3.1.** The *explicit Euler's method* is in the form

$$\boldsymbol{f}_{i+1} = \boldsymbol{f}_i + h\boldsymbol{g}_i. \tag{3.1}$$

**Example 3.2.** The *implicit Euler's method* is in the form

$$\boldsymbol{f}_{i+1} = \boldsymbol{f}_i + h\boldsymbol{g}_{i+1}. \tag{3.2}$$

Another example of an implicit method is the *trapezoid* (or midpoint) method:

$$\boldsymbol{f}_{i+1} = \boldsymbol{f}_i + \frac{h}{2}\left(\boldsymbol{g}_{i+1} + \boldsymbol{g}_i\right). \tag{3.3}$$

*Remark* 4. If time was included in $\boldsymbol{f}_i$ (Remark 3), time-step constraint $t_{i+1} = t_i + h$ would still hold for all the examples above: in the case of Formula 3.1, it would be $f_{i+1}^{[1]} = f_i^{[1]} + hg_i^{[1]} \implies t_{i+1} = t_i + h \cdot 1$, in the case of Formula 3.2: $f_{i+1}^{[1]} = f_i^{[1]} + hg_{i+1}^{[1]} \implies t_{i+1} = t_i + h \cdot 1$, and in the case of Formula 3.3: $f_{i+1}^{[1]} = f_i^{[1]} + \frac{h}{2}\left(g_{i+1}^{[1]} + g_i^{[1]}\right) \implies t_{i+1} = t_i + \frac{h}{2} \cdot (1 + 1)$.

Explicit methods are usually faster than implicit, but are often not appropriate for solving so-called *stiff equations*[1], where the evaluation is *unstable*, in contrast to implicit methods [64]. A solution is unstable if small changes to the step size entail significant differences in resulting values.

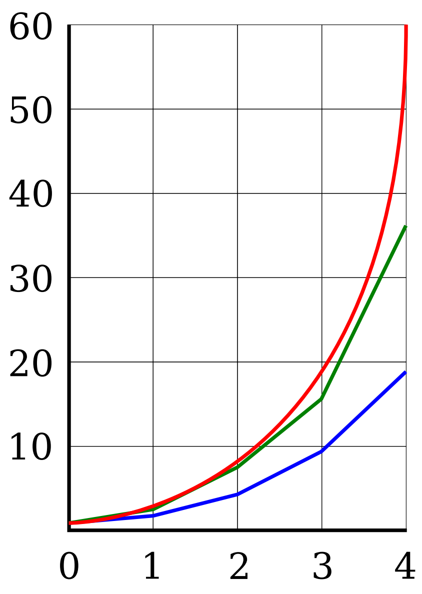


Figure 3.1: Comparison of the explicit Euler's method (blue) and the trapezoid method (green) to the exact solution (red).

The idea behind Euler's methods (whether the explicit or the implicit one) is based on the standard derivative approximation [6]:

$$\dot{f}(t) \approx \frac{f(t+h) - f(t)}{h}. \tag{3.4}$$

Most numeric methods, more sophisticated, arose from Euler's methods, which alone are very simple, but inaccurate, as can be seen in Figure 3.1. The red line shows the mathematical solution, the blue is the explicit Euler's method, and the green is the trapezoid method, both with step size 1 (which is quite wide and usually not too accurate). Observe that in these cases the total distance of the methods to the exact solution increases each step (the reached error is cumulative). Another Figure 3.2 illustrates a single step of the trapezoid method, where again red line stands for the mathematical solution.

Except for partitioning on explicit and implicit methods, we mention two families of the numeric methods: *linear multistep* methods, and *Runge–Kutta* methods. Both families include both explicit and implicit methods and are special cases of general linear

[1]There is no precise definition of stiff equations, but it is characteristic for them that explicit methods require to set a tiny step size, otherwise the solution is unstable; in contrast with implicit methods, which can be stable for an arbitrary step size [64]. Such equations often contain functions with very different time scales.

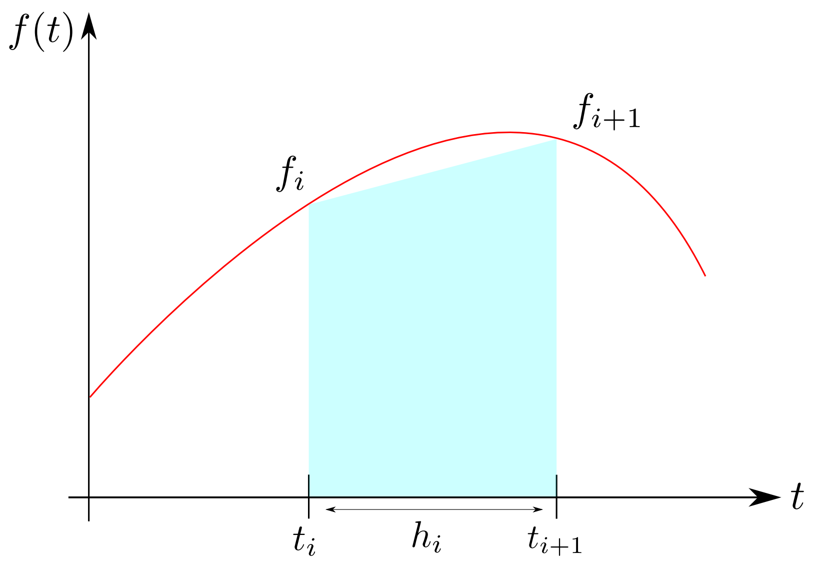


Figure 3.2: A single step of the trapezoid method.

methods [63]. This generalization will not be discussed within this dissertation thesis, and we will focus on the listed families instead.

Note that approximative errors in these methods are not necessarily a problem, since they can be an inherent part of the intended behavior of the models, as discussed in the introduction.

### 3.1.1 Linear Multistep Methods

Linear multistep methods are designed to compute a linear combination of multiple previous sample points [63].

**Definition 6.** *A* linear multistep method*, or a linear $k$-step method, is a stepping method where function values of sample points satisfy*

$$\sum_{j=0}^{k} \alpha_j \boldsymbol{f}_{i+j} = h \sum_{j=0}^{k} \beta_j \boldsymbol{g}_{i+j}$$

*with $\alpha_j \in \mathbb{F}$ and $\beta_j \in \mathbb{F}$, $\alpha_k \neq 0 \wedge (\alpha_0 \neq 0 \vee \beta_0 \neq 0)$.*

*The method is* explicit *iff $\beta_k = 0$, and* implicit *otherwise.*

Therefore, each $\boldsymbol{f}_i$ may depend on up to $k$ previous points. Also, each point is evaluated only once, so each $\boldsymbol{g}_l$, $l \in \{0, \ldots, n\}$ is evaluated only once as well, which can be important in cases when the evaluation of functions $\boldsymbol{G}$ is expensive. The step size $h$ is always constant (i.e. equidistant). A slight disadvantage of these methods is that it is necessary to compute the first $k - 1$ steps with a different method.

**Example 3.3.** Euler's methods are linear 1-step methods with $\alpha_1 = 1$, $\alpha_0 = -1$. The explicit variant (Formula 3.1) is with $\beta_1 = 0$, $\beta_0 = 1$:

$$\boldsymbol{f}_{i+1} - \boldsymbol{f}_i = h\left(\boldsymbol{g}_i\right) \implies \boldsymbol{f}_{i+1} = \boldsymbol{f}_i + h\boldsymbol{g}_i.$$

The implicit variant (Formula 3.2) is with $\beta_1 = 1$, $\beta_0 = 0$:

$$\boldsymbol{f}_{i+1} - \boldsymbol{f}_i = h\left(\boldsymbol{g}_{i+1}\right) \implies \boldsymbol{f}_{i+1} = \boldsymbol{f}_i + h\boldsymbol{g}_{i+1}.$$

**Example 3.4.** The trapezoid method (Formula 3.3) is a linear 1-step method with $\alpha_1 = 1$, $\alpha_0 = -1$, and $\beta_1 = \beta_0 = \frac{1}{2}$:

$$\boldsymbol{f}_{i+1} - \boldsymbol{f}_i = h\left(\frac{1}{2}\boldsymbol{g}_{i+1} + \frac{1}{2}\boldsymbol{g}_i\right) \implies \boldsymbol{f}_{i+1} = \boldsymbol{f}_i + \frac{h}{2}\left(\boldsymbol{g}_{i+1} + \boldsymbol{g}_i\right).$$

Now examples of more sophisticated linear multistep methods [6, 63] follow.

**Example 3.5.** An example of explicit linear multistep methods are *Adams–Bashforth* methods which are used for solving non-stiff equations. Their parameters are $\alpha_k = 1$, $\alpha_{k-1} = -1$, $\alpha_{k-2} = \cdots = \alpha_0 = \beta_k = 0$, and $\beta_j$, $j \neq k$ are based on linear polynomial interpolation of functions $\boldsymbol{G}$ at points $\{t_i, t_{i-1}, \ldots, t_{n-q}\}$, where $q$ is the degree of the interpolation polynomial, and $k = q + 1$. An example is the explicit Euler's method (Formula 3.1) for $q = 0$. Some examples with $q > 0$ are:

$$\begin{aligned}
q = 1:\ \boldsymbol{f}_{i+2} - \boldsymbol{f}_{i+1} &= \frac{h}{2}\Big( 3\boldsymbol{g}_{i+1} - \boldsymbol{g}_i\Big)\\
q = 2:\ \boldsymbol{f}_{i+3} - \boldsymbol{f}_{i+2} &= \frac{h}{12}\Big( 23\boldsymbol{g}_{i+2} - 16\boldsymbol{g}_{i+1} + 5\boldsymbol{g}_i\Big)\\
q = 3:\ \boldsymbol{f}_{i+4} - \boldsymbol{f}_{i+3} &= \frac{h}{24}\Big(55\boldsymbol{g}_{i+3} - 59\boldsymbol{g}_{i+2} + 37\boldsymbol{g}_{i+1} - 9\boldsymbol{g}_i\Big).
\end{aligned}$$

**Example 3.6.** An example of implicit linear multistep methods are *Adams–Moulton* methods that have similar parameters as the Adams–Bashforth methods, with the difference that $\beta_k \neq 0$ and $k = q$. The exception is the case when $q = 0$, for which $k = 1$, which is the case of the implicit Euler's method (Formula 3.2). Other examples of these methods are $q = 1$, which corresponds to the trapezoid method (Formula 3.3), and:

$$\begin{aligned}
q = 2:\quad \boldsymbol{f}_{i+2} - \boldsymbol{f}_{i+1} &= \frac{h}{12}\Big( 5\boldsymbol{g}_{i+2} + 8\boldsymbol{g}_{i+1} - \boldsymbol{g}_i\Big)\\
q = 3:\quad \boldsymbol{f}_{i+3} - \boldsymbol{f}_{i+2} &= \frac{h}{24}\Big(9\boldsymbol{g}_{i+3} + 19\boldsymbol{g}_{i+2} - 5\boldsymbol{g}_{i+1} + \boldsymbol{g}_i\Big).
\end{aligned}$$

**Example 3.7.** Other implicit linear multistep methods are *backward differentiation formulas* (BDF) that are appropriate for solving stiff equations thanks to their high stability properties for $k \leq 6$. Their parameters are $\beta_{k-1} = \cdots = \beta_0 = 0$, and $\beta_k$ and all $\alpha_j$ are, similarly to the Adams methods, based on linear polynomial interpolation, but not of functions $\boldsymbol{G}$, but of functions $\boldsymbol{f}$. Again, $q$ is the degree of the polynomial, and $k = q$. The case $q = 1$ corresponds to the implicit Euler's method. Other examples are:

$$\begin{aligned}
q = 2: &\quad & 3\boldsymbol{f}_{i+2} - 4\boldsymbol{f}_{i+1} + \boldsymbol{f}_i &= 2h\boldsymbol{g}_{i+2} \\
q = 3: &\quad & 11\boldsymbol{f}_{i+3} - 18\boldsymbol{f}_{i+2} + 9\boldsymbol{f}_{i+1} - 2\boldsymbol{f}_i &= 6h\boldsymbol{g}_{i+3} \\
q = 4: &\quad & 25\boldsymbol{f}_{i+4} - 48\boldsymbol{f}_{i+3} + 36\boldsymbol{f}_{i+2} - 16\boldsymbol{f}_{i+1} + 3\boldsymbol{f}_i &= 12h\boldsymbol{g}_{i+4}.
\end{aligned}$$

Consecutively to Remark 4, one can easily check that the time-step constraints would hold in Examples 3.5 and 3.6 as well. In the case of Example 3.7, we show the check for the case $q = 2$: $3t_{i+2} - 4t_{i+1} + t_i = 3(t_{i+2} - t_{i+1}) - (t_{i+1} - t_i) = 3h - h = 2h = 2hg^{[1]}_{i+2}$, and for the case $q = 3$: $11t_{i+3} - 18t_{i+2} + 9t_{i+1} - 2t_i = 11(t_{i+3} - t_{i+2}) - 7(t_{i+2} - t_{i+1}) + 2(t_{i+1} - t_i) = 11h - 7h + 2h = 6h = 6hg^{[1]}_{i+3}$. The case $q = 4$ can analogously be checked as well. Thus, all multistep methods listed in this subsection would work fine even for non-autonomous systems, referring to Remark 3.

### 3.1.2 Runge–Kutta Methods

Runge–Kutta methods stem from Taylor series approximations. Taylor series, however, need to evaluate higher-order derivatives, which is time-consuming. Runge–Kutta methods overcome this by evaluating functions $\boldsymbol{G}$ at multiple mid-points between sample points (i.e. from $[t_i, t_{i+1}]$) while attempting to retain the accuracy of Taylor approximation [6].

Runge–Kutta methods are *single-step* linear methods, but each step consists of multiple *stages*, according to the mentioned mid-points. Unlike multistep methods, function values of the sample mid-points generally cannot be reused, and it is necessary to compute them for each step separately. Thus, it is important that the evaluation of functions $\boldsymbol{G}$ is not too expensive. Another important difference can be that Runge–Kutta methods may use non-constant step sizes, which can in addition be controlled over time in order to increase accuracy—so-called *adaptive* step sizes $h_i$ are controlled by indirect error estimates after each step, which are out of the scope of this dissertation thesis. An example of such a method is Dormand–Prince [63]. We focus only on methods with constant step sizes here, although state-of-the-art methods with adaptive step sizes may be significantly faster and more accurate.

**Definition 7.** *A* Runge–Kutta method *with $s$ stages is a stepping method where function values of sample points correspond to*

$$\boldsymbol{f}_{i+1} = \boldsymbol{f}_i + h \sum_{j=1}^{s} b_j \boldsymbol{y}_j$$

$$\boldsymbol{y}_k = \boldsymbol{G}\left( \boldsymbol{f}_i + h \sum_{j=1}^{s} a_{k,j} \boldsymbol{y}_j \right)$$

*where $k \in \{1, \dots, s\}$, and all $b_j, a_{k,j}$ are* characteristic constants *of the method, all from $\mathbb{F}$. The values of the constants stem from a Taylor approximation.*

*The method is* explicit *iff for all $k \le j$, $a_{k,j} = 0$. Otherwise, it is* implicit.

For non-autonomous systems, it would be necessary to add the time argument into function $\boldsymbol{G}$ in the form $t_i + hc_k$, with $c_k = \sum_{j=1}^{k-1} a_{k,j}$. These constants are typically listed among the other characteristic constants of Runge–Kutta methods. For convenience, we will include constants $c_k$ as well.

**Definition 8.** *A* Butcher's tableau *is a visualization of characteristic constants of a Runge–Kutta method with $s$ stages in the following form:*

$$\begin{array}{c|cccc} c_1 & a_{1,1} & a_{1,2} & \cdots & a_{1,s} \\ c_2 & a_{2,1} & a_{2,2} & \cdots & a_{2,s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}.$$

In the case of explicit Runge–Kutta methods:

- Butcher's tableaux are strictly in lower triangular form with zeroes on the diagonal,
- $\forall j \,.\, a_{1,j} = 0 \implies c_1 = 0$,
- $\boldsymbol{y}_1 = \boldsymbol{G}\Big(\boldsymbol{f}_i + h \sum_{j=1}^{s} a_{1,j} \boldsymbol{y}_j\Big) = \boldsymbol{G}(\boldsymbol{f}_i) = \boldsymbol{g}_i$,
- all $\boldsymbol{y}_k$ depend just on their previous values (i.e. on $\boldsymbol{y}_j$, $j < k$).

**Example 3.8.** The explicit Euler's method (Formula 3.1) is an explicit Runge–Kutta method with $s = 1$, $b_1 = 1$ and $c_1 = a_{1,1} = 0$:

$$\boldsymbol{f}_{i+1} = \boldsymbol{f}_i + hb_1\boldsymbol{y}_1 = \boldsymbol{f}_i + h\boldsymbol{y}_1 = \boldsymbol{f}_i + h\boldsymbol{g}_i.$$

The coefficients correspond to Butcher's tableau

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}.$$

**Example 3.9.** The implicit Euler's method (Formula 3.2) is an implicit Runge–Kutta method with $s = 1$, $b_1 = 1$ and $c_1 = a_{1,1} = 1$:

$$\boldsymbol{f}_{i+1} = \boldsymbol{f}_i + h\boldsymbol{y}_1 = \boldsymbol{f}_i + h\boldsymbol{G}(\boldsymbol{f}_i + h\boldsymbol{y}_1) = \boldsymbol{f}_i + h\boldsymbol{G}\big(\boldsymbol{f}_{i+1}\big) = \boldsymbol{f}_i + h\boldsymbol{g}_{i+1}.$$

The coefficients correspond to Butcher's tableau

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}.$$

**Example 3.10.** The trapezoid method (Formula 3.3) is an implicit Runge–Kutta method with $s = 2$, corresponding to Butcher's tableau

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}.$$

**Example 3.11.** *Classical Runge–Kutta method (RK4)* is an explicit method with $s = 4$. It corresponds to sample points

$$\begin{aligned} \boldsymbol{f}_{i+1} &= \boldsymbol{f}_i + h\left(\frac{\boldsymbol{y}_1}{6} + \frac{\boldsymbol{y}_2}{3} + \frac{\boldsymbol{y}_3}{3} + \frac{\boldsymbol{y}_4}{6}\right) \\ \boldsymbol{y}_1 &= \boldsymbol{g}_i \\ \boldsymbol{y}_2 &= \boldsymbol{G}\left(\boldsymbol{f}_i + \frac{h}{2}\boldsymbol{y}_1\right) \\ \boldsymbol{y}_3 &= \boldsymbol{G}\left(\boldsymbol{f}_i + \frac{h}{2}\boldsymbol{y}_2\right) \\ \boldsymbol{y}_4 &= \boldsymbol{G}(\boldsymbol{f}_i + h\boldsymbol{y}_3) \end{aligned} \tag{3.5}$$

and to Butcher's tableau

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}.$$

Referring to Remarks 4 and 3, if time was included in $\boldsymbol{f}$ then it would satisfy the condition $t_{i+1} = t_i + h$ iff $\sum_{j=1}^{s} b_j = 1$, because $f_{i+1}^{[1]} = t_{i+1} = t_i + h\sum_{j=1}^{s} b_j y_j^{[1]}$ and $y_k^{[1]} = G^{[1]}(\star) = 1$ yields $t_{i+1} = t_i + h\sum_{j=1}^{s} b_j \cdot 1 = t_i + h$. Therefore, such methods can be properly used even for non-autonomous systems. Actually, all the listed Runge–Kutta methods satisfy this condition.

## 3.2 SAT Solving

SAT solvers refer to Boolean satisfiability (SAT), defined in Definition 3. The result of the solvers is either `sat`, in the case when the formula is satisfiable, or `unsat`, when it is unsatisfiable. In addition, the result may also be `unknown` in special cases like when the configured resource limits were exceeded (e.g. timeout). When a formula is `sat`, there may be an additional output with a satisfying assignment of the variables.

Algorithms used in SAT solvers usually exploit the fact that the input formula is in conjunctive normal form (CNF). Such a formula is usually represented in the form of a set of clauses. CNF format is useful in a sense that it is sometimes easy to discover (or avoid) a *conflict*—an assignment of a set of variables that would cause unsatisfiability—because just a single clause that evaluates to $\perp$ causes the whole formula to be unsatisfiable.

Most SAT solvers are nowadays based on the DPLL algorithm (Davis–Putnam–Logemann–Loveland) [100], which introduced basic techniques of the search procedure, in particular:

- *unit propagation*, which forces an assignment that avoids a conflict, in cases when there is a clause where only one literal is not assigned yet,
- *pure literal elimination*[2], which forces a variable, that is not assigned yet and all its literals appears in only one polarity throughout the whole formula in CNF, to a value s.t. all clauses, where the literals appear, are satisfied,
- *decide*, which guesses a value for a variable (usually based on some heuristics) that is not assigned yet, and usually only in cases when no other rule can be applied at the moment (especially propagations),
- *backtrack*, which reverts the recent decision when resolving a conflict.

Another technique that is implemented in most state-of-the-art SAT solvers is called *conflict-driven clause learning* (CDCL). After arriving at a conflict, CDCL remembers the incompatible decisions that caused the conflict in the form of a *conflict clause*, which is inserted into the clause set. The smaller a conflict clause, the more general constraints it represents. As a result, small conflict clause may prune the searched state space significantly.

Conflict clauses are often related to an enhancement of the backtrack rule called *backjump* rule [100], which returns directly back to a decision that caused the conflict, instead of reverting all the decisions that are not necessarily related to the conflict, one by one. Such an approach always terminates (e.g., no deadlock is possible) [100], and the discovered conflicts are efficiently resolved.

We do not discuss algorithms that are related to the optimizing variant of the problem—maximum satisfiability (MAX-SAT).

[2]While this rule is included in the original algorithm, it is usually used only as a preprocessing step, or not at all by modern SAT solvers, since it may be quite expensive for large formulas.

## 3.3 SMT Solving

In Section 2.4, we defined Satisfiability Modulo Theories (SMT)—a generalized problem of SAT. Approaches to SMT often encompass sophisticated algorithms and techniques that may be more-or-less specific to a given theory (or its fragment), but can also be usable in various contexts or for any theory at all.

There are the same possible outputs of SMT solvers as in the case of SAT solvers. In the case of a satisfying assignment, additional variables related to underlying theories might be included in the output as well. Output `unknown` can, among others, denote that the used implementation is not able to arrive at a definite decision of satisfiability (especially when the problem is not decidable), or that a formula does not belong to any supported fragment of a theory.

Most, if not all, SMT solvers are built upon an underlying SAT solver. However, the way the SAT solver is used can vary. The two most common families of SMT approaches, not necessarily disjoint, are *eager* and *lazy* approaches [100, 14].

### 3.3.1 Eager Approach

Eager approach is de-facto just translation of the original formula from first-order logic into propositional logic, including all theory-related semantics, which is then passed into the SAT solver that is used as black-box. This approach ultimately relies on the efficiency of the translation and on the SAT solver itself. However, the eagerness can result in congesting the SAT solver by too many constraints that are possibly not essential for arriving at the result.

The eager approach may be more efficient for some theories, for example, where the representation of variables is similar to binary encoding (e.g. the theory of bitvectors). However, in the rest cases, it is not used too much in state-of-the-art solvers. Eager approach will not be investigated anymore in this dissertation thesis.

### 3.3.2 Lazy Approach

The lazy approach is based on lazy evaluation, which means that the semantics of a theory are resolved as late as they are needed. The procedure is based on the cooperation of a SAT solver and a theory solver, so-called $\mathcal{T}$-solver. The SAT solver is based on DPLL with CDCL and computes only with *abstracted* formulas, meaning that atomic theory formulas are abstracted to pure Boolean variables. The $\mathcal{T}$-solver is a specialized solver that decides the satisfiability of conjunctions of theory literals—literals of atomic theory formulas.

The unsatisfiability of such abstract propositional formula implies unsatisfiability of the original formula, regardless of the underlying theory. On the other hand, satisfiability implies that it *can* be also *consistent* with the theory, which is when the $\mathcal{T}$-solver kicks in. If it is considered inconsistent, some explanation of the inconsistency is added into the propositional constraints set in the form of a conflict clause (as expected for a CDCL algorithm), and another Boolean assignment is sought. This process is repeated until the SAT solver finds a consistent assignment, or returns unsatisfiable. While the SAT solver is guaranteed not to, $\mathcal{T}$-solver can actually block (e.g. deadlock, infinite loop, ...), depending on the underlying theory.

Lazy SMT solvers are often not fully lazy and may encode some semantics into the propositional formula directly, like eager solvers do, but only in the case of the most fundamental semantics. There is always a trade-off of directly allowing the SAT solver to use theory constraints and of useless remembering of constraints that were actually not necessary for arriving at the solution.

There are several types of lazy approaches [100] and various related techniques. For starters, a naive lazy approach is called *offline* and uses a classical SAT solver that searches for a *full* propositional assignment, and whenever an inconsistency is found, and after explaining it, the whole procedure is restarted from scratch. Such SMT solvers are quite easy to implement, since the SAT solver can be practically used as black-box. However, such flexibility cannot overcome significant inefficiencies entailed by the full assignments, causing a lot of useless work to be wasted. An issue is that a number of inconsistencies may be discovered early, that is, after assigning just a small subset of the variables, meaning that all the rest did not have to be evaluated at all. State-of-the-art solvers implement much more efficient methods. Their overview is described in the rest of the subsection.

**Online Approach.** In contrast with the offline approach, *online* approach requires a specialized SAT solver that:

- supports *partial* propositional assignments,
- implements callbacks inserted into the most important parts of the DPLL algorithm, which allows the $\mathcal{T}$-solver to interfere the search procedure,
- (as a result) allows checking the consistency over time.

If, for example, consistency is being checked after setting each Boolean variable, then an inconsistency can be found early and no additional, useless assignments are done. However, in some theories, checking the consistency can actually be quite an expensive process, and therefore a suitable strategy is to do it only sometimes.

In any way, the $\mathcal{T}$-solver must keep track with the SAT solver and the currently assigned Boolean variables, and also wrt. to the theory model and the current state of the underlying variables (e.g., arithmetic variables). In the case of backtracking, it is especially important to keep track with the current assignment correctly.

Online approach may also allow to fully exploit backjumping of the SAT solver to a certain state where the assignment was still consistent, instead of restarting from scratch.

**Theory Propagation.** Experiments showed that even tighter integration of the SAT solver and the $\mathcal{T}$-solver can be very beneficial. *Theory propagation* guides the propositional search according to the theory semantics which avoids a large number of inconsistencies from hapenning and can massively improve the performance of the procedure. For example, when an abstracted theory predicate is assigned by the SAT solver, all other predicates that relate to the assigned one, according to the theory, are forced to a proper value, ensuring consistency. For instance, if the SAT solver assigns a predicate $x = 0$ to true in the real arithmetic, and another, still unassigned predicate $x = 1$ is present in the constraints set, then the predicate $x = 1$ is instantly forced to false, otherwise it would be inconsistent with $x = 0$. Without theory propagation, the SAT solver would have to do only unit propagations or worse, a decision whether to assign $x = 1$ to true or false, because it would have no clue which one to choose[3]. Consequently, it would be possible that inconsistency $x = 0 \wedge x = 1$ would appear.

If theory propagation is a cheap process (depending on the theory), then it can be applied literally after any Boolean assignment of an atomic theory formula by the SAT solver. Such a strategy is called *exhaustive theory propagation*, where actually the consistency is usually being checked exhaustively too. As a consequence, assignments of such a strategy may happen to be never inconsistent. For example, an algorithm that checks satisfiability in the difference logic from Section 2.4.1 is always consistent and very efficient [100].

**Decision Heuristics.** Without additional strategies on the decisions of the SAT solver, the decisions are completely blind in terms of the underlying theory. On one hand, the heuristics that a given SAT solver uses on its own might be very efficient with respect to the propositional part itself. Also, some theories do not really depend on decision heuristics and focus on techniques like thorough theory propagation. For example, linear real arithmetic defines a lot of semantical rules which themselves are robust enough and the procedure is not too sensitive to the order of decisions on Boolean variables.

On the other hand, for some theories, especially with a low number of semantical rules, such an order of decisions can play a crucial role to make the whole procedure efficient. For example, given an incomplete simple theory of equalities where an equality of a variable and a number causes assignment of the variable to the number, and given predicates $x = 0$ and $x = 1$, deciding predicate $x = 0$ to true is an example of a useful decision—such an assignment can activate a theory propagation which forces $x = 1$ to $\bot$, because variable $x$ already has value 0. However, in the opposite case, when deciding predicate $x = 0$ to $\bot$, the theory propagation is not possible, since the value of variable $x$ is left unassigned. And although continuing with the assignment of $x = 1$ to $\bot$ would not cause an inconsistency, the value of variable $x$ would still be unknown.

[3]In this particular case, an eager preprocessing like $x = 0 \implies \neg(x = 1)$ would help, but such exclusions are generally difficult to resolve within the preprocessing stage, if the right-hand side of the equations is more complicated. Also, again, this may produce a number of not-necessarily useful constraints.

Another example, with the same theory but with predicates $x = 0$ and $y = x$, $x = 0$ is the better candidate for a decision, because in the opposite case, $y = x$ causes no variable assignment.

Note that when a SAT solver selects a decision variable, sometimes it may be beneficial to suggest a specific variable according to the given encoding of an underlying problem. An example of such a case is traversing a graph encoded into propositional constraints, where shortest paths are preferred to be searched at first. Then, a specific strategy for the decisions should be probably used instead of general heuristics.

Moreover, many SMT theories, and even their fragments, are often too general even though they are being used in specific contexts where only specific instances can appear. Therefore, it might be convenient to adapt the search strategy according to a set of instances that are typical for a given problem.

Speaking of decision heuristics, an interesting option is to involve machine learning techniques or the like. Algorithms which are efficient "in most cases" can be useful, because even in the field of formal methods it is not necessarily an issue when such a learning-based technique suggests a wrong decision. The only possible consequence is that the search procedure may be slowed down, but decisions cannot cause the final result to be incorrect.

CHAPTER 4

# SAT Modulo Differential Equation Simulations: Definition

In this chapter, we present the target problem of this dissertation thesis. As a result, it is possible to model planning or verification tasks which involve complex systems such as cyber-physical systems. Unlike our original formalization [75] which directly defined a logical theory, as usual in the formal methods community, here we take a semi-formal presentation that aims at a more general engineering audience, introducing the parts of the formalism step by step. For completely formal definitions, we refer to the original publication [75].

We will start with defining systems of functions (Section 4.1), where we present a formalism on differential equations and so-called invariants. Then, we define SAT modulo differential equations (Section 4.2)—a satisfiability problem that involves differential equations with the semantics that are based on real numbers. Finally, we define the problem where the ODEs are handled using simulations (Section 4.3).

## 4.1 Systems of Functions

The motivation of this section is to define a formalism which allows specifications of systems of ODEs (Section 2.1) where in addition we allow more general final constraints on the equations and constraints that must hold at all times. The defined function and predicate symbols allow to treat such functions as variables and the constraints focus just on function values at all times or at the initial or final value of time.

Such a formalism corresponds to our paper [75], but here we explicitly split the definitions that are related to ODEs from those related to SAT. Using such a formalism, one can design a standalone and flexible ODE solver that handles rich continuous constraints but is still independent of any Boolean reasoning.

We first focus on isolated systems of functions. Then, we will also allow connecting of multiple systems into consecutive stages.

**Definition 9.** *A* system of functions $\mathcal{F}$ *is a type with an associated set of* functional variables.

*The elements of the type are functions, but we do not explicitly write their argument which models time over a shared* domain $\mathcal{D}$ *that starts at 0 and ends at a certain* length $\tau$. *The length is finite and does not have to be fixed. The functions are in the form* $f\colon \mathcal{D} \to \mathcal{R}$ *with an abstract* codomain $\mathcal{R}$.

For example, functional variable $f \in \mathcal{F}$ represents a function $f\colon \mathcal{D} \to \mathcal{R}$ with the argument $t \in \mathcal{D}$. However, we do not write the argument explicitly: instead of $f(t)$, we will write just $f$. In logic, this would mean that we treat functional variables as first-order objects.

The definition fixes the initial value of time to 0. The length $\tau$ will also be called *final value of time*. The length is not necessarily fixed and can actually depend on the progress of the functional variables. Intuitively, the functions start and end at the same time and progress synchronously.

For now, the codomain $\mathcal{R}$ corresponds to real numbers $\mathbb{R}$ and the domain $\mathcal{D}$ to interval $[0, \tau]$ with the length $\tau \in \mathbb{R}^{\geq 0}$, and the functional variables represent smooth functions. In this case, $\mathcal{D} \subseteq \mathcal{R}$, but not in general. Later, these symbols will also represent different sets, for instance the codomain $\mathcal{R}$ will represent a set of floating-point numbers $\mathbb{F}$ (such as 64-bit IEEE 754).

To allow some operations that are necessary to handle functional variables as functions, we introduce special function symbols.

**Definition 10.** Functional operators *over a system of functions $\mathcal{F}$ are unary functions defined as follows:*

- $\mathit{diff}\colon \mathcal{F} \to \mathcal{F}$ *models* differentiation *of $f \in \mathcal{F}$ s.t.* $\mathit{diff}(f) := \dot{f}$,
- $\mathit{init}\colon \mathcal{F} \to \mathcal{R}$ *models the* initial value *of $f \in \mathcal{F}$ s.t.* $\mathit{init}(f) := f(0)$,
- $\mathit{final}\colon \mathcal{F} \to \mathcal{R}$ *models the* final value *of $f \in \mathcal{F}$ s.t.* $\mathit{final}(f) := f(\tau)$,
- $\mathit{embed}\colon \mathcal{R} \to \mathcal{F}$ *converts a number $x \in \mathcal{R}$ to the corresponding constant function s.t.* $\mathit{embed}(x) := f$ *where $f \in \mathcal{F}$ and for all elements $t \in \mathcal{D}$, $f(t) = x$.*

*Here $\mathcal{R}$ is the codomain and $\mathcal{D}$ is the domain of the system of functions $\mathcal{F}$.*

Now, operator *diff* corresponds to the usual differential operator from mathematical analysis. We will usually abbreviate the expression $\mathit{diff}(f)$ as just $\dot{f}$ and implicitly assume operator *embed* whenever an argument from $\mathcal{F}$ is expected and an argument from $\mathcal{R}$ is present. We call constraints that contain *init* operators *initial conditions*, and similarly, constraints that contain *final* operators *final conditions*.

Before we define predicates on functional variables (such as differential equations), we first need a definition of the expressions that may appear within such predicates.

**Definition 11.** *A* functional expression *over a system of functions $\mathcal{F}$ is an expression that contains:*

- *functional variables $f \in \mathcal{F}$,*
- *function symbols $\{+, -, \cdot, /, \exp, \log, \sin, \cos, \tan\}$ with the usual arity and point-wise lifted to functional arguments,*
- *functional operator $embed$ whose argument is a constant from $\mathcal{R}$.*

*Such an expression denotes a function $G_d \colon \mathcal{F}^d \to \mathcal{F}$.*

For example, expression $x + y$ is a valid functional expression if $x, y \in \mathcal{F}$ and denotes a function in $\mathcal{F}^2 \to \mathcal{F}$ that assigns to each $x$ and $y$ in $\mathcal{F}$ a function that assigns to each $t \in \mathcal{D}$ the value $x(t) + y(t)$. Furthermore, using values from $\mathcal{R}$ is implicitly possible too thanks to the operator $embed$, which can even be omitted. For example, $x + c$ with constant $c \in \mathcal{R}$ is allowed because it denotes $x + embed(c)$.

Now we define predicates on functional variables. These cover differential equations, but in addition we also define constraints where no differentiation appears, which is new regarding Section 2.1.

**Definition 12.** *A* functional ODE *over a system of functions $\mathcal{F}$ is an expression of the form*

$$\dot{f} = G_d(\boldsymbol{g})$$

*where $f \in \mathcal{F}$ is a functional variable of $\mathcal{F}$, $G_d$ is a functional expression over $\mathcal{F}$ and $\boldsymbol{g} \in \mathcal{F}^d$.*

*An* invariant *over a system of functions $\mathcal{F}$ is an expression of the form*

$$F_c(\boldsymbol{f}) \bowtie G_d(\boldsymbol{g})$$

*where $\bowtie \in \{\geq, =\}$ is a binary predicate symbol point-wise lifted to functional arguments, $F_c$ and $G_d$ are functional expressions over $\mathcal{F}$, and $\boldsymbol{f} \in \mathcal{F}^c$ and $\boldsymbol{g} \in \mathcal{F}^d$ are vectors of functional variables of $\mathcal{F}$.*

*A* functional predicate *is either a functional ODE or an invariant.*

*A* solution *of a functional predicate is an assignment of functions to all functional variables $f \in \mathcal{F}$ that satisfies the expression denoting the predicate.*

We will allow writing invariants with symbol $\leq$ too since it is equivalent to symbol $\geq$ with swapped arguments.

The intuition behind functional predicates is that the functional variables progress synchronously and that we lift the usual meaning of the symbols over the set $\mathcal{R}$ to the whole domain $\mathcal{D}$ of the system of functions $\mathcal{F}$. Note that Definition 9 implies that the length $\tau$ of the functions of a solution must be the same and must be finite. For example, invariant $x \geq 0$ expresses the fact that functional variable $x$ is always greater or equal to the constant function 0, i.e. $x(t) \geq 0$ for all $t \in \mathcal{D}$, and functional ODE $\dot{x} = 1$ denotes the corresponding differential equation.

Hence, the defined function and predicate symbols allow to treat certain functions as variables where the constraints focus just on function values at all times or at the initial or final value of time.

If we compare functional ODEs to Definition 1, here we do not use vector form $\dot{\boldsymbol{f}}$ and $\boldsymbol{G}$, but we still restrict the arguments of functional expression $G_d$ to functional variables that belong to the same system of functions $\mathcal{F}$.

---

**Example 4.1.** Two examples of simple functional ODEs are

$$\dot{x} = v \tag{4.1}$$

and

$$\dot{v} = -g \tag{4.2}$$

with $x, v \in \mathcal{F}$ and the constant $g \in \mathcal{R}$. Note that $\dot{v} = -g$ is an abbreviation for $\mathit{diff}(v) = -\mathit{embed}(g)$. Such ODEs look the same as the particular ODEs in Examples 2.1 and 2.3.

We can also provide initial conditions for the particular ODEs, referring to Example 2.2. Since $t_0 = 0$, we can use $\mathit{init}(x) = 10$ in the place of $x(t_0) \mapsto 10$ and $\mathit{init}(v) = 0$ in the place of $v(t_0) \mapsto 0$.

---

**Example 4.2.** An example of a simple invariant is

$$x \geq 0 \tag{4.3}$$

which expresses the requirement that the falling object never appears below the ground, referring to the invariant of the hybrid automaton in Example 2.3.

Such an invariant requires $x(t) \geq 0$ for all $t$. This does not necessarily mean that at the end, $x(\tau) = 0$, though, resulting in nondeterminism. If we used a final condition instead of the invariant, for example $\mathit{final}(x) \geq 0$, then we would just require $x(\tau) \geq 0$, and would not constrain the previous values, which would yield even more nondeterminism. If we used the invariant in combination with final condition $\mathit{final}(x) = 0$, then we would require $x(t) \geq 0$ for all $t$ and also $x(\tau) = 0$, resulting in a deterministic final condition (which will be defined later). In the case when $x$ models a continuous function, then using final condition $\mathit{final}(x) \leq 0$ in combination with the invariant would be equivalent to $\mathit{final}(x) = 0$.

---

**Example 4.3.** An example of a more complex invariant is

$$x - z \leq 2x \cdot y \tag{4.4}$$

provided that $x, y, z \in \mathcal{F}$.

---

Referring to Remark 1, it is possible to use directly function $t$ in the place of the shared independent variable (e.g. $t$) of a system $\mathcal{F}$, because each system of functions $\mathcal{F}$ may contain time function $t \in \mathcal{F}$ s.t. $\dot{t} = 1$, $\mathit{init}(t) = 0$ and $\mathit{final}(t) = \tau$. Therefore, although there is no functional operator that obtains the value of the length $\tau$, one can simply use $\mathit{final}(t)$.

---

**Example 4.4.** A typical and straightforward constraint on ODEs is an explicit constraint on time (e.g. the value $t$ in Remark 2). We can provide such a constraint in the form of an invariant, for example

$$t \leq 10 \tag{4.5}$$

which constrains final time $\tau$ s.t. $0 \leq \tau \leq 10$. Note that we set just an upper bound on $\tau$. Such an invariant is in fact equivalent to final condition $\mathit{final}(t) \leq 10$. However, using the invariant in combination with final condition $\mathit{final}(t) = 10$ (or $\mathit{final}(t) \geq 10$) would explicitly require $\tau = 10$. In the case of $\mathit{final}(t) = 10$, the invariant is actually not necessary.

---

We defined a formalism on systems of functions including functional predicates using real numbers and smooth functions. However, the predicates themselves are separate from others and we still did not define how to, for example, solve a system of ODEs (Definition 1). Therefore, it is necessary to somehow group the predicates together and define how solutions of such groups of predicates look like.

**Functional Flows.** The goal is to allow solving systems of ODEs, but possibly with more complex stopping conditions, in contrast to Section 2.1. However, we still stick to initial value problems (IVPs), meaning that the initial conditions need to be fixed.

**Definition 13.** *A* simple initial condition *over a system of functions $\mathcal{F}$ is an expression in the form*

$$\mathit{init}(f) = \eta$$

*where $f \in \mathcal{F}$ is a functional variable of $\mathcal{F}$ and $\eta$ is an expression that evaluates to a number from the set $\mathcal{R}$.*

*Remark* 5. The definition can be extended to a form which, for example, allows intervals such as $\mathit{init}(f) \in [\eta, \theta]$. However, this would require to use appropriate methods which handle not only IVPs, but also interval initial (and consequently final) conditions.

Now we proceed to the definition of the desired systems.

**Definition 14.** *A* functional flow *is a set of simple initial conditions and functional predicates, all regarding a system of functions $\mathcal{F}$. For each $f \in \mathcal{F}$, there must be exactly one simple initial condition and one functional ODE.*

*A* solution *of a functional flow is an assignment of functions to all functional variables $f \in \mathcal{F}$ that is a solution of all functional predicates of the functional flow, and at the same time initial values must be fixed to the values that correspond to the simple initial conditions.*

This means that solutions of such systems must satisfy conjunction of all the participated constraints. Functional flows may express systems of ODEs along with initial conditions (referring to IVPs) where in addition invariants are allowed. Note that the definition excludes final conditions, which must be handled separately.

Every solution of a functional flow always has a finite length by definition. However, in some cases it may be unclear what is the longest possible length, meaning that there are no further solutions that are longer. Moreover, such an upper bound does not necessarily exist.

*Remark* 6. If any of the invariants of a functional flow is eventually violated, then the longest possible length of all solutions of the flow exists and corresponds to the time of the violation.

---

**Example 4.5.** Formulas from Example 4.1 can be put together into a functional flow in the form of an initial value problem, but there is no invariant. Together with Example 4.2, it can be rewritten into the following functional flow with two initial conditions:

$$\begin{aligned} init(x) &= 10 \\ init(v) &= 0 \end{aligned} \tag{4.6}$$

and three functional predicates:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -g \\ x &\geq 0. \end{aligned} \tag{4.7}$$

Such a system might correspond to the behavior of the hybrid automaton in Example 2.3, but only within the location *down*. Moreover, we have not specified final conditions and there are more possible solutions of the flow, resulting in nondeterminism (see Example 4.2 and the definition below).

In order to make the resulting trajectories look the same as in Figure 2.1 in Example 2.2, we may replace the invariant $x \geq 0$ with $t \leq 2$.

---

Functional flows do not include final conditions, so they must be expressed separately. In Example 4.2, we showed that with no final conditions there are usually more possible lengths of the system. We now define when final conditions constrain a functional flow in such a way that the resulting length is unique.

**Definition 15.** *Given a functional flow, a set of final conditions is* deterministic *iff all solutions of the flow that in addition satisfy all the final conditions have the same length. Otherwise, the final conditions are* nondeterministic. *Here, the arguments of operators $\mathit{final}$ in the final conditions must belong to the same system of functions as the functional flow.*

However, we did not show how to express a model such as in Example 2.3, even considering only deterministic final conditions. The reason is that it is necessary to allow multiple systems with possibly different lengths of the domains. Still, it is not possible to mix functional variables from different systems within functional predicates. The only permitted functional constraints on functional variables $x \in \mathcal{F}_i$ and $y \in \mathcal{F}_j$, $i \neq j$ are initial and final conditions, for example $\mathit{init}(y) = \mathit{final}(x)$. This allows to arrange functional flows into consecutive *stages*.

**Stages of Functional Flows.** A *stage* (of functional flows) is a functional flow where the initial conditions may depend on final values of other stages. Since functional flows (Definition 14) require that initial conditions are simple initial conditions, forming direct connections of stages is possible only in cases when solutions to the particular stages are unique. Moreover, here we want to stick just to functional flows which do not contain final conditions. Therefore, we require that the length of the solution of each stage corresponds *exactly* to the time when any invariant of the functional flow is violated. This would correspond to using deterministic final conditions, but here final conditions may appear only within simple initial conditions, which means that their values must already be fixed.

Particular parts of a stage (i.e. of a functional flow) always belong to just one system of functions. Furthermore, we require that the particular parts cannot be shared with other stages.

The length $\tau$ of a system of functions $\mathcal{F}$ is specific to the particular system and may differ from lengths of other systems. This is an important fact, since we will usually be interested in models which require more than just one functional flow. We use subscripts to distinguish different systems of functions, for example, we write $\mathcal{F}_1$, $\tau_1$ and $t_1$ in the case of one system and $\mathcal{F}_2$, $\tau_2$ and $t_2$ in the case of other, different system.

---

**Example 4.6.** An example of multiple stages (of functional flows) follows. We start with a stage that corresponds to Example 4.5 and add two more consecutive stages in order to model three deterministic steps in the hybrid automaton from Example 2.3:

$$\begin{array}{rl|rl|rl}
\mathit{init}(x_1) &= 10 & \mathit{init}(x_2) &= \mathit{final}(x_1) & \mathit{init}(x_3) &= \mathit{final}(x_2) \\
\mathit{init}(v_1) &= 0 & \mathit{init}(v_2) &= -K \cdot \mathit{final}(v_1) & \mathit{init}(v_3) &= \mathit{final}(v_2) \\
\dot{x}_1 &= v_1 & \dot{x}_2 &= v_2 & \dot{x}_3 &= v_3 \\
\dot{v}_1 &= -g & \dot{v}_2 &= -g & \dot{v}_3 &= -g \\
x_1 &\geq 0 & v_2 &\geq 0 & x_3 &\geq 0
\end{array} \tag{4.8}$$

where $x_1, v_1 \in \mathcal{F}_1$, $x_2, v_2 \in \mathcal{F}_2$ and $x_3, v_3 \in \mathcal{F}_3$, and $g, K$ are constants, $K \in [0, 1]$. The systems of functions may also include the time functions $t_1 \in \mathcal{F}_1$, $t_2 \in \mathcal{F}_2$ and $t_3 \in \mathcal{F}_3$, but these functional variables do not participate here.

The example models three consecutive stages of a bounce of a ball, where $K$ determines elasticity of the bounce. Note that, for example, $\mathit{init}(x_2) = \mathit{final}(x_1)$ is here equivalent to $\mathit{init}(x_2) = 0$. Switching between the stages is deterministic and happens exactly when $x_1 = 0$ and then when $v_2 = 0$.

The example is indeed similar to Example 2.3: the first stage corresponds to location *down*, the second stage corresponds to location *up*, and the third stage corresponds again to location *down*. A possible solution of the three consecutive stages is available in Figure 2.3, which corresponds to the case when $K = 0.8$.

In the end, it is possible to model multiple consecutive IVPs with stopping conditions expressed by invariants (Definition 12). However, such a formalism does not consider (nondeterministic) final conditions. It also does not consider any Boolean reasoning—it only supports conjunctions of certain constraints (Definition 14). In the next section, we will introduce a more sophisticated formalism that is based on formulas. Such a formalism can model the stages in a similar way, but at the same time it handles more robust constraints on top of particular functional flows, including (nondeterministic) final conditions.

**Relationship with Hybrid Automata.** As presented above, sometimes it is possible to express multiple steps of a hybrid automaton (Section 2.2) using stages of functional flows:

- Each stage relates to a certain location.
- The continuous dynamics and invariants associated to the location are represented by functional ODEs and invariants, respectively.
- Discrete transitions between locations can be modeled using operators $\mathit{init}$ and $\mathit{final}$ between the stages. By default, all continuous variables of hybrid automata remain unchanged when switching locations, which must be explicitly expressed within the stages. However, the presented stages alone cannot model nondeterministic transitions between locations, which would require to use final conditions.

### 4.1.1 Simulation Semantics

In this section, we formulate an interpretation of systems of functions that is not based on real numbers and continuous functions, but on a floating-point arithmetic and discretization. Referring to Section 2.1.1 and using methods presented in Section 3.1, we assume discretization of functional predicates with constant step size. We lift the meaning of all real arguments to floating-point numbers, resulting in the following definition.

**Definition 16.** *A* floating-point interpretation *of a system of functions $\mathcal{F}$ interprets the sets from Definition 9 such that $\mathcal{R} := \mathbb{F}$ and $\mathcal{D} := \{u\Delta \mid u \in \{0, \ldots, \frac{\tau}{\Delta}\}\}$, where $\Delta$ is* step size *and $\tau$ is the length that is required to be a multiple of $\Delta$. Set $\mathbb{F}$ is a set of floating-point numbers with the usual rounding to the nearest floating-point number.*

*The semantics of functional ODEs is given by a numerical method of solving ODEs, which is a parameter of the interpretation. In the case of invariants, we only require that they hold for $u \in \{0, \ldots, \frac{\tau}{\Delta} - 1\}$, that is,* without the final point.

We will also call such an interpretation *simulation semantics* of systems of functions.

For explaining why we exclude the final point from invariants, we refer to Example 4.6. The first stage uses invariant $x_1 \geq 0$, and the stopping condition that corresponds to violating the invariant can be expressed by final condition $\mathit{final}(x_1) \leq 0$. When interpreting $x_1$ as a continuous function, this makes perfect sense: the switch to the next stage occurs exactly when $x_1 = 0$. The fact that the intersection of the invariant and the final condition is just a point does not cause any trouble. However, this does not work in our approximate interpretation because it is highly unlikely that, after discretization, a point is reached for which precisely $x_1 = 0$. To circumvent this problem, we allow the invariant to be violated at the very final point of $x_1$ (i.e. when $u = \frac{\tau_1}{\Delta}$, where $\tau_1$ is the length of system $\mathcal{F}_1$) which at the same time is the first point that allows switching to a next stage.

Note that the above is also a reason why using final conditions that are based on equalities instead of inequalities is usually not a good idea, at least in the case of using an approximate semantics. For instance, $\mathit{final}(x_1) = 0$ is likely to be unsatisfiable wrt. chosen simulation semantics, while $\mathit{final}(x_1) \leq 0$ works fine.

Also note that in the case of the semantics with continuous functions, $\mathit{final}(x_1) \leq 0$ corresponds to a deterministic final condition, but *not* in the case of simulation semantics. The reason is that we allow that the invariant is violated at the last point, but we do not *require* it. For example, we can arrive at a point where $\mathit{final}(x_1) = 0$ (although this is unlikely), or a point where $\mathit{final}(x_1) < 0$. Therefore, the proper deterministic final condition here is $\mathit{final}(x_1) < 0$, which explicitly states that invariant $x_1 \geq 0$ must be violated. But again, this is not the case of the semantics with continuous functions, where this would not work.

*Remark* 7. In the case of simulation semantics, deterministic final conditions for a functional flow actually correspond to the negation of the conjunction of its invariants, while applying operator *final* on all functional variables. For instance, in the case of Example 4.5 extended of an additional invariant $v \leq 0$, the corresponding deterministic final conditions are

$$\mathit{final}(x) < 0 \lor \mathit{final}(v) > 0. \tag{4.9}$$

## 4.2 Satisfiability Modulo Differential Equations

According to the fact that the dissertation thesis is focused on simulations of ODEs, a whole separate section is dedicated to satisfiability modulo differential equations, an interesting special case of SMT.

SMT traditionally uses *first*-order predicate logic as its basis, while here we want to reason about functions (the solutions of ODEs), which are often viewed as objects of second order. We overcome this seeming mismatch by handling those functions as first-order objects[1]. Moreover, some theories such as the theory of real arithmetic or the theory of integer arithmetic assume only one kind of variables that can appear in formulas. However, this is not our case, because we need to include both real numbers and functions altogether. Therefore, we will use the definitions from Section 4.1 where two types of values are distinguished.

**Definition 17.** *A* numerical variable *ranges over the set $\mathcal{R}$ from Definition 9.*

Hence we will use two types of variables based on Definition 9: numerical variables and functional variables. Note that in Section 4.1, elements from the set $\mathcal{R}$ appear only in the form of constants, while from now on we allow variables as well. Although numerical variables may correspond to real numbers $\mathbb{R}$ and real variables in the theory of real arithmetic (Section 2.4), we stick to Definition 9 and the symbol $\mathcal{R}$ because we will need to mix these variables with functional variables which use the same set $\mathcal{R}$.

For explaining the intuition behind the structure of formulas we expect to handle here, we first describe an illustrative example, before introducing further formal definitions.

---

**Example 4.7.** We present an extended version of Example 4.6 which shows a bouncing ball model that is based on functional flows (Definition 14). Here, in addition, we model linear drag of the ball. The resulting formula represents deterministic constraints, but the constraints can be easily extended to nondeterministic. The example is similar to Example 2.3. Figure 4.1 shows possible trajectories as solutions of the formula which is to be described. (The figure is a bit similar to Figure 2.3).

In order to model multiple consecutive stages of functional flows, we unroll formulas in a similar way as in bounded model checking (Section 2.5). The particular stages include functional variables $x_j$ and $v_j$ that belong to system of functions $\mathcal{F}_j$ and describe the vertical position and velocity of the ball, respectively, in $j$-th stage of the unrolling, $j \in \{1, \dots, J\}$, $J \in \mathbb{Z}^{>0}$. Next, the stages include Boolean variables $\mathrm{up}^{[j]}$ which model the discrete state of the ball—whether it is bouncing up or falling down, similarly to the locations of the hybrid automaton in Example 2.3. Finally, numerical variables $t^{[j]}$

[1]While this is new in the context of SAT modulo ODE, this is quite common in mathematics. For example, Zermelo-Fraenkel set theory uses such an approach to define sets, relations, etc. within *first-order* predicate logic.

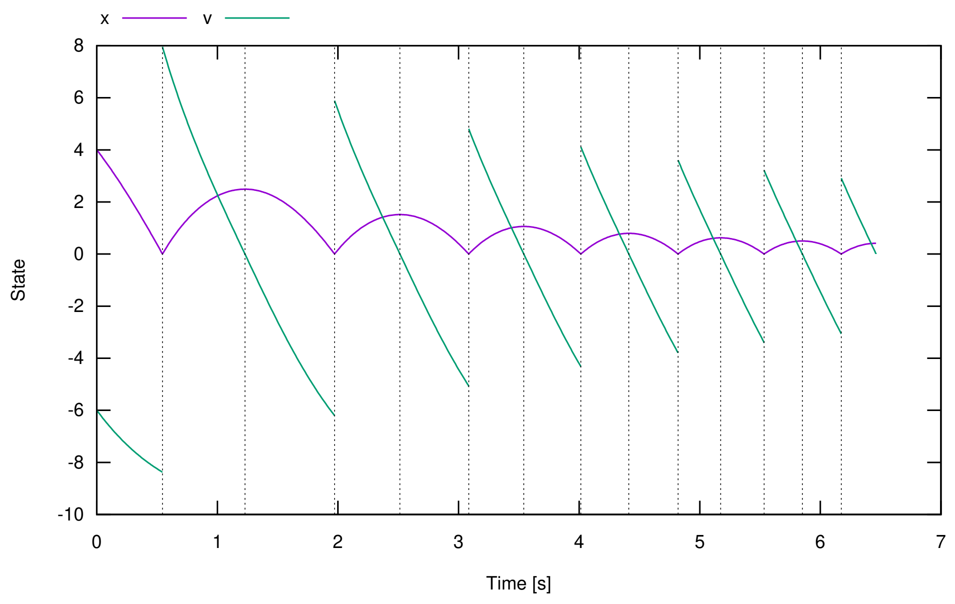


Figure 4.1: A solution of the bouncing ball model with $J = 14$, $K = 0.95$, $D = 10$.

range from $\mathcal{R}$ and model time at the beginning of the stage (not to be confused with time functions $t_j \in \mathcal{F}_j$).

An example of a formula that contain the particular stages $j$ and represents the whole bouncing ball model is

$$
\begin{aligned}
& g = 9.81 \land K = 0.95 \land D = 10 \\
& \land \neg \mathrm{up}^{[1]} \land t^{[1]} = T_0 \land \mathit{init}(x_1) = X_0 \land \mathit{init}(v_1) = V_0 \\
& \land \bigwedge_{j=1}^{J} \Bigg( \dot{x}_j = v_j \land x_j \geq 0 \\
& \qquad \land \mathrm{ITE}\left(\mathrm{up}^{[j]},\ \dot{v}_j = -g - \frac{v_j}{D},\ \dot{v}_j = -g + \frac{v_j}{D}\right) \\
& \qquad \land \mathrm{ITE}\left(\mathrm{up}^{[j]},\ v_j \geq 0,\ v_j \leq 0\right) \\
& \qquad \land \mathrm{ITE}\left(\mathrm{up}^{[j]},\ \mathit{final}(v_j) \leq 0,\ \mathit{final}(x_j) \leq 0\right) \Bigg) \\
& \land \bigwedge_{j=1}^{J-1} \Bigg( \left(\mathrm{up}^{[j]} \Rightarrow \neg \mathrm{up}^{[j+1]}\right) \land \left(\neg \mathrm{up}^{[j]} \Rightarrow \mathrm{up}^{[j+1]}\right) \\
& \qquad \land t^{[j+1]} = t^{[j]} + \mathit{final}(t_j) \\
& \qquad \land \left(\ \mathrm{up}^{[j]} \Rightarrow \left(\mathit{init}(x_{j+1}) = \mathit{final}(x_j) \land \mathit{init}(v_{j+1}) = 0\right)\right) \\
& \qquad \land \left(\neg \mathrm{up}^{[j]} \Rightarrow \left(\mathit{init}(x_{j+1}) = 0 \land \mathit{init}(v_{j+1}) = -K \cdot \mathit{final}(v_j)\right)\right) \Bigg) \\
& \land t^{[J]} \leq T^* \land \mathit{final}(x_J) \geq X^*.
\end{aligned}
\tag{4.10}
$$

Constants $g$ and $K$ relate to Example 4.6. Linear drag of the ball is modeled using coefficient $D$. Constants $T_0$, $X_0$ and $V_0$ are arbitrary initial numerical values, and $T^*$ and $X^*$ are final numerical values, which are used within so-called goal conditions. Predicate ITE is an abbreviation for if-then-else (see the list of mathematical abbreviations). The formula may be divided into three parts: initial conditions (i.e. the first two rows), the model (i.e. the two big conjunctions with stages), and the goal conditions.

Note that, for example, $x_j$ and $x_k$, $j \neq k$ model the same physical quantity, but do not necessarily share domains, because $\mathcal{F}_j = \mathcal{F}_k$ is not necessarily true. This is the reason why here we use the notation $x_j$ instead of $x^{[j]}$ which would mean that there is a vector $\boldsymbol{x} = (x^{[1]}, x^{[2]}, \dots)$, but particular elements $x^{[j]}$ would belong to different sets: $x^{[1]} \in \mathcal{F}_1, x^{[2]} \in \mathcal{F}_2, \dots$, therefore $\boldsymbol{x} \in \mathcal{F}_1 \times \mathcal{F}_2 \times \dots$, which would be confusing. Nonetheless, variables $x_j$ and $v_j$ can be grouped for example into $\boldsymbol{f}_j = (x_j, v_j)$, $\boldsymbol{f}_j \in \mathcal{F}_j^2$ (similarly to Example 2.1).

The only possible dependencies of functional variables that belong to different systems of functions (e.g. $x_j$ and $x_k$, $j \neq k$) are dependencies between their initial and final values, referring to stages of functional flows in Section 4.1. For example, $\mathit{init}(x_{j+1}) = \mathit{final}(x_j)$ is allowed, but $x_{j+1} \geq x_j$ would be invalid. Nevertheless, functional variables from different systems do not have to be necessarily dependent on each other. For example, in the case of $\mathit{init}(x_{j+1}) = 0$, there is no dependency on $x_j$.

Each stage of the formula includes deterministic final conditions (Definition 15) for two possible functional flows that depend on the value of variable $\mathrm{up}^{[j]}$: $\mathrm{ITE}(\mathrm{up}^{[j]}, \mathit{final}(v_j) \leq 0, \mathit{final}(x_j) \leq 0)$. The final conditions always constrain only one of the functional variables, which however refers to the invariant that is actually violated at that stage: $\mathit{final}(v_j) \leq 0$ in stages where $\mathrm{up}^{[j]}$, and $\mathit{final}(x_j) \leq 0$ in stages where $\neg\mathrm{up}^{[j]}$. This means that switching between the stages is indeed deterministic. Note that in the case of simulation semantics, it would be necessary to replace the nonstrict inequalities of the final conditions by strict inequalities, in order to make the conditions indeed deterministic (following Remark 7): $\mathrm{ITE}(\mathrm{up}^{[j]}, \mathit{final}(v_j) < 0, \mathit{final}(x_j) < 0)$.

---

Finally, following Definition 4 we provide the *theory of ODEs* [75] which is here based on systems of functions from Section 4.1.

**Definition 18.** *The* signature *of the* theory of ODEs *contains:*

- *sort symbols $\mathcal{R}$ and $(\mathcal{F}_j)_{j \in \mathcal{J}}$ for a finite index set $\mathcal{J}$, where symbol $\mathcal{R}$ corresponds to numerical variables (Definition 17) and symbols $\mathcal{F}_j$ correspond to a system of functions (Definition 9),*
- *all rational constants,*
- *function symbols $\{+, -, \cdot, /, \exp, \log, \sin, \cos, \tan\}$ and binary predicate symbols $\{\geq, =\}$ with the usual arity, defined on the sort $\mathcal{R}$ and on each of the sorts $(\mathcal{F}_j)_{j \in \mathcal{J}}$,*
- *functional operator symbols (Definition 10).*

*An* atomic formula *is either an atomic theory formula or a Boolean variable. An* atomic theory formula *is either:*

- *a* numerical predicate *which is an atomic formula of the form $\eta \bowtie \theta$ where $\bowtie$ is a predicate symbol from $\mathcal{R}$ and $\eta, \theta$ are terms built in the usual way using function symbols from $\mathcal{R}$ and the functional operators $init$ and $final$ whose argument is allowed to be a functional variable,*

- *or a functional predicate (Definition 12).*

*A* numerical literal *is either a numerical predicate or its negation.* Functional literals *are defined analogously. A* theory literal *is either a numerical literal or a functional literal. A* literal *is either a theory literal or a Boolean literal. A* formula *is an arbitrary Boolean combination of literals.*

We will also respectively say that a literal is negative or positive iff it does or does not represent a negation.

The resulting formulas have the usual mathematical semantics where we interpret the sorts $\mathcal{R}$ and $\mathcal{F}_j$, $j \in \mathcal{J}$ and functional predicates according to the real continuous semantics presented in Section 4.1. Here numerical variables are also called *real variables*.

Note that due to the hidden $\forall$-quantification over time within functional predicates, it is not true that, for example, $x \geq 0$ is equivalent to $\neg\,(x < 0)$: the former means all the time $x$ is greater or equal to zero, whereas the latter means not all the time $x$ is lower than zero. Moreover, predicate symbol $<$ is not defined.

Using the defined language, it is possible to express functional flows (Definition 14) or stages such as in Example 4.6. Furthermore, now we can express disjunctions and negations of the particular constraints (Example 4.7). It is also possible to denote non-deterministic final conditions.

We define the notion of variable assignment, satisfiability and unsatisfiability as usual. Satisfiability modulo differential equations then corresponds to checking satisfiability of a (quantifier-free) formula of the theory of ODEs.

The theory of ODEs cannot be generally decidable, not only because of the exponential and trigonometric functions, but especially because of the solutions of ODEs [22].

## 4.3 SAT Modulo Differential Equation Simulations

Finally, satisfiability modulo differential equations from Section 4.2 can be defined based on numerical simulations of the ODEs (Section 2.1.1), that is, with an alternative semantics of the ODEs. For this we will use specific semantics of functional variables that we introduced in Section 4.1.1 with numeric methods from Section 3.1. The resulting theory differs from the theory of ODEs only in its semantics.

**Definition 19.** *A* floating-point interpretation *of the* theory of ODEs *interprets the sorts and functional predicates based on Definition 16, and interprets numerical predicates in the obvious floating-point analogue to the semantics presented in Section 4.2.*

*Here numerical variables are also called* floating-point variables*.*

We will also call the interpretation as *simulation semantics* of the theory of ODEs.

As a result, for example, the domains of the functional variables are sets of discrete points, as stated in Definition 16, and importantly, invariants may be violated in the final point. According to Section 2.1.1, the resulting interpretation is parametrized not only by a chosen set $\mathbb{F}$, but also by a numerical method that solves the ODEs (which can further depend on a number of parameters).

Due to the numerical simulations, it is likely that an implementation of deciding satisfiability of formulas of the theory will require all ODEs to be initialized using operator $init$, although it is not directly required by the theory.

A fragment of the theory with no functional variables, that is, just with floating-point variables, would result in something like a theory of floating-point numbers. An example is [24], which however concentrates on the intricacies of floating point arithmetic and on completeness and soundness of the corresponding satisfiability problem. We, on the other hand, ignore these aspects and instead focus on the handling of ODEs which implies some significant consequences. For example, as already mentioned, ODEs need to be initialized and it is likely that most of simple constraints on floating point variables depend on the solutions of ODEs. Therefore, it might be reasonable to assume that most of these constraints will depend on some explicit initial values too, since it is hard to predict the outputs of ODEs. As a result, one can view even simple floating point constraints as numeric simulations.

Using this semantics, the resulting theory is not only decidable in the theoretical sense [75], but also efficiently decidable for formulas of the type occurring in this dissertation thesis. We present our approach to the defined problem in Chapter 6.

**Strong Satisfiability.** We also provide a refinement of the satisfiability problem where we only care about final values of functional flows that stem from a violation of an invariant. The same applies for final values of stages of functional flows in Section 4.1 (see also Example 4.6). This way, the lengths of functional flows are the longest possible. This reduces the search space for solvers. The intuition is to ensure that functional flows only end with the violation of an invariant. This corresponds to hybrid systems where jumps are only enabled in situations where they actually *must* be taken.

**Definition 20.** *A variable assignment $\alpha$* strongly satisfies *a formula $\phi$ of the theory of ODEs with the simulation semantics iff it satisfies $\phi$ and in addition the following holds: for all systems of functions $\mathcal{F}_j$, there is an invariant $I$ in $\phi$ belonging to $\mathcal{F}_j$ such that $\alpha$ satisfies $I$ but the final values of $\alpha$ do not satisfy $I$.*

*A formula $\phi$ of the theory of ODEs with the simulation semantics is* strongly satisfiable *iff a variable assignment that strongly satisfies $\phi$ exists.*

Using final conditions, it is possible to ensure syntactically that strong satisfaction and satisfaction are the same (such as in Example 4.7). Consider the following example:

---

**Example 4.8.** Let $\phi$ be a formula

$$\mathit{init}(x) = 0 \land \dot{x} = 1 \land x \leq 10 \land \mathit{final}(x) \bowtie \xi \tag{4.11}$$

where $\bowtie \in \{\leq, =, \geq, >\}$ and $\xi > 0$ is a constant. We will discuss several concrete cases of $\bowtie$ and $\xi$, which correspond to single functional flows.

If $\bowtie$ is $\leq$ or $\geq$ and $\xi < 10$, $\phi$ is not strongly satisfiable because the final condition is nondeterministic: with $\bowtie$ being $\leq$, $\mathit{final}(x) \in [0, \xi]$, and with $\bowtie$ being $\geq$, $\mathit{final}(x) \in [\xi, 10]$, meaning that the length of the flow is not unique.

If $\bowtie$ is $=$ and $\xi < 10$, the final condition may be deterministic, because it may constrain the length of the flow s.t. it is unique[2]: $\mathit{final}(x) = \xi$. Nonetheless, even if the resulting length is unique, it is definitely not the longest possible length and invariant $x \leq 10$ is *not* violated at the final value: $\mathit{final}(x) = \xi < 10 \leq 10$. Therefore, an assignment that satisfies such a formula does *not* strongly satisfy the formula.

If $\bowtie$ is $\geq$ and $\xi = 10$, then $\mathit{final}(x) \geq 10$ in combination with invariant $x \leq 10$ still does not result in a unique length of the flow. Following the discussion in Section 4.1.1, the final value may or may not violate the invariant, resulting in nondeterminism. Such a formula is still not strongly satisfiable.

If $\bowtie$ is $>$ and $\xi = 10$, then $\mathit{final}(x) > 10$ negates invariant $x \leq 10$ and therefore explicitly states that the invariant must be violated (Remark 7) and results in a unique length of the flow that is also the longest possible. Therefore, in this case $\phi$ is strongly satisfiable.

---

### 4.3.1 SAT Modulo Black-Box Simulations

Simulations of ODEs can be generalized in such a way that systems of functions are not restricted to model only differential equations, but to more general simulations such as black-box functions that somehow transform a vector of floating point values to another vector. Examples of such black-box functions are Simulink models, railway simulators, solvers of systems of linear equations, neural networks, etc.

For such functions, the SMT solver has no implicit information about the relationship between the outputs and the inputs and must always blindly simulate every particular situation. This can be similar in the case of a non-robust algorithm for SAT modulo ODE simulations. However, given a specific problem or an algorithm, some theory information can be provided explicitly. For example, in the case of a lazy online approach

[2]Here, an issue with the uniqueness is a possible ambiguity of the equation due to various possible choices of rounding modes of the floating-point numbers. We already mentioned in Section 4.1.1 that using final conditions that are based on an equality is discouraged.

to a problem where railway simulators are employed as well, the simulators can explicitly provide the fact that the progress of the involved variables that describe the trains is always monotonic (provided that the trains are not reversing). The resulting SMT solver can then be able to learn more general facts when it arrives at an inconsistency.

CHAPTER 5

# State of the Art Tools

In Sections 5.1–5.3 we focus on implementations of algorithms presented in Chapter 3. Importantly, in Section 5.4 we also discuss complex and sophisticated state-of-the-art tools that may handle problems that are similar to SAT modulo ordinary differential equations (ODEs) from Section 4.2. Some of the presented tools are based on reachability analysis of hybrid automata (Section 2.2), instead of SAT. Note that some of the algorithms that are used within these tools were not covered in Chapter 3, because we focus on satisfiability problems and on simulations of ODEs.

## 5.1 Numeric ODE Solvers

Some of the methods mentioned in Section 3.1, and many more, are implemented in a plethora of ordinary differential equation (ODE) solvers (or more complex tools). We present a few examples of robust and efficient standalone solvers that are implemented in C and C++ and support parallel variants of the algorithms:

- SUNDIALS [65] is a complex toolkit that has been developed for decades originally in Fortran, now implemented in C with an OCaml API available. It implements implicit methods, namely Adams–Moulton methods (Example 3.6) for non-stiff equations, and backward differentiation formulas (BDF) (Example 3.7) for the stiff ones. It also implements some Runge–Kutta methods, for example, it combines explicit methods with implicit methods, resulting in methods with adaptive step sizes.

  The toolkit also supports a form of root-finding, meaning that it is possible to use constraints on function values as final conditions of the simulations, not only constraints on the time.

- Odeint [1] is a part of Boost C++ libraries. In contrast to SUNDIALS, Odeint is a relative novice in the field of ODE solvers. It implements a number of methods, both explicit and implicit, including Adams methods (Examples 3.5 and 3.6) and explicit Runge–Kutta methods, for example RK4 (Example 3.11), but also Dormand–Prince method with adaptive step sizes.

Odeint library is well customizable. Since it is based on C++ template programming and concepts, one can configure various data types, stepping methods, or even interfere each step with an external function, which for example allows to implement checking of complex invariants after each step.

## 5.2 SAT Solvers

There is a bunch of SAT solvers that implement the DPLL algorithm with conflict-driven clause learning (CDCL), as presented in Section 3.2. These solvers are nowadays extremely efficient for many applications and are used in industry. Each year, many SAT solvers participate in the international SAT solver competition [67], where one can observe an enormous progress of the performance of state-of-the-art solvers across the whole community—there is not just one solver which dominates the others over years.

We explicitly mention the MiniSat2 solver [45, 46], which has won several competitions. It is no longer being developed and its performance is actually already quite far away from the current state of the art, but it became a basis of many of these successor solvers. One of advantages of MiniSat2 is that its interface and even implementation is quite simple, therefore easy to understand, yet still relatively efficient.

If a solver is used as black-box, it is possible to use a standard text format of the input and also output: DIMACS, which is supported by most of SAT solvers. However, SAT solvers are often used via API, which sometimes offer various callbacks to other routines that interfere the search procedure.

We omit discussion of optimizing SAT solvers (which refer to maximum satisfiability (MAX-SAT)).

## 5.3 SMT Solvers

As indicated in Section 3.3, there is a plethora of possible approaches to Satisfiability Modulo Theories (SMT), regarding the support and handling various theories and their fragments. Still, most of state-of-the-art solvers conform to SMT-LIB standard which allows to compare the solvers against each other within given theories. Inspired by SAT solvers, there is an international competition of SMT solvers [11] that is based on SMT-LIB (see below for more details). Notable competitors are for example CVC5 [10, 5], Yices2 [43, 44], Z3 [39, 97], OpenSMT [26, 27], MathSAT [33, 60], Barcelogic [18], and lately also Z3++ [29, 28] which is based on Z3. CVC5 and OpenSMT, for instance, are built on top of MiniSat2. We do not list SMT solvers that handle differential equations here, they are mentioned later.

Handling optimization in SMT, that is, Optimization Modulo Theories (OMT), is still an area under active research, with no widely used standards and benchmark libraries available. For example, authors of CVC5 mentioned that OMT is their future work [10]. Still, some comparisons of existing solvers, and also of CP implementations (Constraint Programming), exist [36]. The OMT solvers include OptiMathSAT [112, 111] which is based on MathSAT, $\nu$Z [17] which is developed within the Z3 project, and Barcelogic also handles some optimization [99].

### 5.3.1 SMT-LIB

SMT-LIB is an international initiative aimed at facilitating research and development in SMT [12]. Most importantly, it develops and promotes common input and output languages [35, 13] for SMT solvers, and provides a large library of benchmarks. It forms a basis of the competition of SMT solvers.

SMT-LIB introduces a new terminology for theory fragments, which they call *logics*. As for theories, various logics can be combined together, which is important for many solvers. Some logics are subsets of others, resulting in a hierarchy. Solving more constrained logics can be much more efficient, especially in cases without quantifiers. Solvers implement only logics, theories serve as a theoretical background.

Each logic defines syntax (using core functions) and semantics, but new functions can be introduced too. The library uses sorts to distinguish different types of variables (e.g. `Bool`, `Real`), or return types of functions. New sorts can also be introduced. Functions must be defined for a fixed number of arguments of possibly different sorts.

Forming a formula is done via *assertions*, which add subformulas into assertion stack. Then, all the formulas from the stack are put into conjunction. It is also possible to revert some assertions (i.e. remove them from the stack).

Some OMT solvers define optimization extensions of the SMT-LIB language, but they are not unified.

## 5.4 Tools for Handling Hybrid Systems

In the case of models of cyber-physical systems and the like (such as Example 2.3), typically both non-trivial discrete and especially continuous reasoning are needed for reachability, verification and/or planning tasks. The underlying dynamics of the corresponding models may be difficult or impossible to specify without ODEs. In this section, we offer an overview of tools that handle such models.

Some of the tools are based on the formalism of hybrid automata (Section 2.2) and build on reachability analysis. Next, many tools use *hybrid systems* as a more general name for models with both discrete and continuous behavior, where the discrete behavior is described for example by discrete modes, and the continuous behavior is described using differential equations. Still, various formalisms are often interchangeable. For example, in the case of problems related to Boolean satisfiability (SAT) modulo ODEs described in Section 4.2 (such as Example 4.7), the formulas can be transformed into a corresponding hybrid automaton (although the number of locations of the resulting automaton that would be necessary to model all possible discrete states may be high). Therefore, for the sake of this section where we survey many tools that are in some sense similar, we will use the name hybrid systems for the underlying models.

In the case of hybrid automata, we covered theoretical background on the formalism but we did not cover algorithms that concern the corresponding reachability analysis, which is in general very complex topic and is out of scope of this dissertation thesis. The same applies also for the other formalisms that are not based on SAT, and for approaches that handle ODEs in a different way than based on simulations.

The resulting approaches can vary not only in the underlying formalism, but also in the way how constraints are handled. For example, there are several specific approaches where classical mathematical enclosures of ODEs, using an interval arithmetic, are applied: Taylor model approximation [32], set-based reachability [20], SMT solving [66], Constraint Programming [59], Interval Constraint Propagation (ICP) [47], or a combination of more ways, like SAT modulo ICP [58]. An effort of these tools, within particular (systems of) ODEs, is to prove that a unique solution to an initial value problem (IVP) (Definition 2) exists. Then, they compute theoretical bounds that contain this solution [88, 98], which is an honourable property applicable for the purposes of precise mathematical proofs. They also support more-or-less non-linear constraints. An issue with such approaches is, however, that solving ODEs this way is extremely difficult.

Some interval ODE solvers are based on a numerical method, like Runge–Kutta, but in a validated context with intervals, where the bounds of errors are carefully controlled [40]. Most importantly, the underlying semantics of ODEs still does not rely on approximation and floating-point computation. An example of such a solver is DynIBEX [41].

In addition, a model that is to be analyzed, typically of an embedded system, is often designed *not* based on mathematical analysis (e.g. on physical laws), but based on simulations. As the design of the model evolves, it is usually being parametrized, and such parameters are often estimated again based on the simulations s.t. the ODEs behave as closely to the real system as possible. Still, some models can be unrelated to simulations, for example some statistic and biochemic models, where using these mathematical approaches can be a reasonable choice.

We address another limitation of most of these state-of-the-art approaches regarding the supported constraints with differential equations, which are required to be in the form of monolithic building blocks that contain a full system of ODEs within which no Boolean reasoning is allowed. For example, in the case of hybrid automata, a location defines a full system of ODEs where all continuous variables are involved, and switching to another location means changing the whole system of ODEs.

There is a number of tools that focus on reachability analysis of hybrid automata or hybrid systems, where usually the main intended application is verification and model checking. An effort of a performance comparison of such tools on unified benchmarks exists [52, 101] in the form of a friendly competition. Notably, the presented benchmarks exhibit only trivial discrete state spaces, which is typical for a number of such tools, while we focus on models where also complex discrete state space appears. Examples of tools that have competed there through the years are (in alphabetic order) Ariadne [25], CORA [3], C2E2 [49, 48], dReal3 [58], DynIBEX [40], Flow* [32], HySon [21], JuliaReach [20], Kaa [42], KeYmaera X [54] and Verse [84]. Some of the tools, however, support only linear continuous dynamics, for example HyLAA [9] and SpaceEx [53].

DynIBEX [40, 41] is a plug-in of IBEX [113], a library for constraint processing over real numbers. DynIBEX provides a set of validated numeric integration methods that are based on Runge–Kutta schemes (including e.g. RK4—Example 3.11). They also implement affine arithmetic to allow modeling with intervals. The presented benchmarks, such as [80], show interesting case studies where for example discrete state space is modeled in the form of so-called sampled switched systems. Here the changes of discrete modes occur periodically at a constant sampling period and the problem consists in finding a switching rule in order to satisfy a given specification. The problem is called control synthesis and is based on state-space bisection. However, either the number of different discrete modes is low, or the performance degrades significantly in the case of a higher number of modes.

There is another tool, HSolver [103], that refers to verification of hybrid systems where however the time is not bounded. On the other hand, again, the discrete part of the models is trivial, and also the dimension of the continuous part is quite low.

Now we take a closer look into more specific tools that are more related to the problems defined in Sections 4.2 and 4.3.

### 5.4.1 SAT Modulo ODE Solvers

Here we focus on approaches that are based on or close to SAT-modulo-theory solving. All such state-of-the-art approaches are based on classical mathematical semantics of ODEs. However, differential equations are not included within standard SMT logics. An augmentation of non-linear real arithmetic with trigonometric and exponential transcendental functions exists [86], but this in general is still not enough to model complex dynamic phenomena.

Solvers dedicated specifically to ODEs exist. However, such methods often exhibit significant differences with each other, even between the underlying problem statements. Hence we will ignore the differences and focus on handling formulas of the theory of ODEs (Definition 18). Since these methods do not allow Boolean reasoning inside the full systems of ODEs, as mentioned above, they actually support only a subset of the theory of ODEs.

We now list the representatives of these tools:

- Hydlogic [66] implements lazy online SMT solving with theory propagation (see Section 3.3), with an interval-based theory solver which deals with real constraints involving ODEs. It is designed to solve problems in the area of bounded model checking (BMC) (Section 2.5).

- iSAT-ODE [47] is a SAT modulo ODE solver that is based on the iSAT algorithm [51]. However, the approach substantially deviates from traditional lazy SMT solving, because the propositional SAT solver directly manipulates Interval Constraint Propagations (ICPs), resulting in a much tighter integration. It exploits the algorithmic similarities between constraint solving and propositional solving that is based on Davis–Putnam–Logemann–Loveland (DPLL). However, constraints cannot only be satisfied or unsatisfied for all valuations from an interval box, but can also contain a mixture of points satisfying or violating a constraint [51].

  The verified formula is required to be in restricted form related to BMC, starting from an initial state, followed by unwinding of the transition system and finally leading to a target state satisfying a property of interest.

  Interval solutions of ODEs are similar to the approach used in Hydlogic. In addition, they are combined with a second layer of reasoning about ODEs, which is only applicable under certain conditions, but may yield tighter enclosures. This additional layer generates so-called bracketing systems.

- dReal3 [58, 56, 68] is based on OpenSMT. It supports lazy online SMT solving with theory propagation, but with an added theory that is based on ICP. Validated interval enclosures of ODEs are computed by the CAPD solver [57], which builds on decades of research. The dReal3 solver decides $\delta$-satisfiability of a $\delta$-perturbated formula, where $\delta$ is a numerical error bound specified by the user. Unsatisfiable results do not involve any numerical approximation.

  The input language [8] stems from SMT-LIB and served us as an inspiration while designing our language. The tool also provides a preprocessor called dReach [78] which translates a BMC problem, encoded using a specific syntax, into a formula of the input language.

  Some additional features are provided, like heuristics (e.g. for BMC), an additional support for parallel compositions of models, etc. Newer version of the tool, dReal4, is available as well, but dReach is not provided for the new version.

In Chapter 7 we will present experimental comparison of our approach with dReal3 on a few simple case studies. We will not use dReal4 since the case studies are based on bounded model checking.

### 5.4.2 Simulation Tools

The problem of verifying differential equations wrt. simulation semantics, along with other constraints, has been addressed before, but not in a SAT modulo theory context. Simulation-based solvers do analyze the underlying systems of ODEs according to the methods presented in Section 3.1.

The discussion of particular tools follows.

- Simulink is a part of proprietary extensive computational software pack MATLAB®. It offers modeling of hybrid systems in the form of compositions of blocks. Differential equations are represented by continuous-time integrator blocks.

A stream-based approach is adopted to formalize variable-step solver semantics and to establish a computational model of time that supports discrete-time and discrete-event behavior [96]. However, the support of models with a large number of discrete modes is limited.

Simulink is widely used in industry: the design process of complex cyber-physical systems is tightly connected to the models which are periodically being updated based on real experiments with the corresponding devices. Therefore, such systems indeed emerge from simulations. However, the verification part of the design process is still based on classical testing, not on automatic analysis of the underlying models.

There are also free alternatives to Simulink, for example Xcos and Ptolemy.

- HySon [21] is actually an enhancement over the Simulink tool. The algorithm performs set-based simulation of hybrid systems with uncertain parameters, expressed in Simulink. The uncertain parameters are expressed as intervals, and HySon computes a good approximation of the set of all possible Simulink executions at once—and this approximation also takes into account rounding errors of the floating-point arithmetic. Instead of plain interval arithmetic, it uses affine arithmetic which tracks linear dependencies between the underlying variables.

  However, there are no significant changes to Simulink in the discrete part of the procedure.

Although we do not consider the listed approaches to be well suitable for verification and planning tasks for cyber-physical systems, due to the drawbacks mentioned above, some techniques seem to be relevant to our approach and are considered to be a future work, for example supporting intervals using affine arithmetic.

Chapter 6

# SAT Modulo Differential Equation Simulations: Solver

In Section 5.4, we presented state-of-the-art tools and accented two groups of them:

- Tools that are based on SMT solving (Section 5.4.1). Such methods can efficiently handle discrete constraints. However, the semantics of ODEs are based on classical mathematical analysis, which may have disadvantages that were discussed in the section.

- Tools that simulate the underlying ODEs (Section 5.4.2), but are not efficient in discrete reasoning. Insufficient discrete reasoning is actually also the case of all the other mentioned tools that fit into neither of these groups.

In this chapter, we propose an alternative approach to SAT modulo ODEs. We defined the problem in Section 4.2 and alternative semantics of ODEs that are based on numerical simulations in Section 4.3. Notably, we focus on a simplified problem and check strong satisfiability of a formula (Definition 20). Within the simulations we apply some numerical methods from Section 3.1. Here we focus on algorithmic and implementation details of the approach. To explain the intuition behind the structure of formulas that we expect to handle, we refer to Example 4.7.

Given a formula of the theory of ODEs (Definition 18), the result of our solver is `sat` if the formula is (strongly) satisfiable and `unsat` if it is not strongly satisfiable. However, the result may also be `unknown` in cases when the solver arrives at certain final states where it still did not assign some variables to any value. Therefore, the solver is not complete in general. However, in [75] we provide a syntactical characterization of the kind of inputs for which our solver provides an efficient solution. We support this through experiments presented in Chapter 7 and Chapter 8.

Section 6.1 concentrates on the design of a standalone numerical ODE solver. The ODE solver does not depend on the SMT framework and Boolean reasoning. Still, it allows computation of selected functional predicates of particular functional flows from

Section 4.1, with parameters such as a stepping method and step size. Section 6.2 provides the design of a lazy SMT solver that is based on simulations of ODEs. The section includes a description of the theory solver and integration of this solver with an underlying SAT solver. Finally, Section 6.3 presents the input language of our solver and Section 6.4 provides some implementation details.

## 6.1 ODE Solver

Here we build on the definitions of systems of functions and functional flows presented in Section 4.1, and design an ODE solver as an independent software component such that:

- It solves functional flows (Definition 14), which are similar to IVPs (Definition 2)—systems of ODEs with initial values. In addition to classical systems of ODEs, invariants (Definition 12) are allowed in the flows.
- The solver is based on stepping of differential equations, and can be parametrized with a numerical method (Section 3.1) along with the corresponding parameters, for example step size.
- Stopping conditions of the stepping stem from a violation of an invariant of the flow.
- The solver has no explicit support of Boolean reasoning, but at the same time the interface allows to select various possible variants of constraints regarding a certain functional variable—which will be useful for the SMT solver that will be described in Section 6.2.

**Stepping of Differential Equations.** The ODE solver accepts systems of ODEs such as Example 4.5, that is, systems that are based on functional flows (Definition 14). According to Section 4.1.1, we currently support only constant step-size integration methods, some of which were presented in Section 3.1, for example RK4. Also, we currently do not support invariants with the predicate symbol $=$ (Definition 12).

*Notation.* We will separate particular components of a functional flow as follows: vector $\boldsymbol{x_0}$ with the initial values of the corresponding simple initial conditions, vector $\mathbf{odes}$ with the ODEs, and set $\mathit{invs}$ with the invariants. The position of particular elements of the vectors $\boldsymbol{x_0}$ and $\mathbf{odes}$ correspond to an arbitrary order of the particular functional variables in a system of functions $\mathcal{F}$.

Every solution of a functional flow can be parametrized by a set of parameters $\mathcal{P}$, especially the chosen numerical method, and step size $\Delta$ which we explicitly exclude from the set $\mathcal{P}$.

The numerical methods are based on stepping and typically use matrix operations, where each step might be implemented using a standalone function DO_STEP that transforms a vector of floating-point values to another, possibly depending also on some previous values. Repetitive calls to function DO_STEP results in an output vector.

The particular calls to function DO_STEP can be interleaved by callbacks, for example with checking of invariants. This way, we can implement stopping conditions of such a stepping based on invariants. We ensure that all invariants $invs$ hold in such a way that before each step we simply numerically check whether all the invariants evaluate to true. Then, we keep on stepping until *any* invariant is violated, resulting in Algorithm 6.1 which describes function DO_STEPS. The function returns not only the last step, but also the previous steps in the form of a list of all the steps—a trajectory, as described in Section 3.1.

**Algorithm 6.1:** Function DO_STEPS.

```
DO_STEPS($\boldsymbol{x_0}$, odes, $invs$, $\mathcal{P}$, $\Delta$) $\longrightarrow$ List[Vector[$\mathbb{F}$]] :
$\boldsymbol{x} \leftarrow \boldsymbol{x_0}$
$\boldsymbol{T} \leftarrow (\boldsymbol{x})$
while ALL_INVARIANTS_HOLD($\boldsymbol{x}$, $invs$) do
    $\boldsymbol{x} \leftarrow$ DO_STEP($\boldsymbol{x}$, odes, $\mathcal{P}$, $\Delta$)
    $\boldsymbol{T} \leftarrow \boldsymbol{T} \parallel (\boldsymbol{x})$            ▷ append the computed step to the trajectory
return $\boldsymbol{T}$
```

Importantly, we assume that the invariants will be eventually violated. Therefore, following the observation in Remark 6 (Section 4.1), such a stepping is guaranteed to terminate, provided that function DO_STEP terminates, which is a reasonable assumption referring to Section 3.1. To support the assumption, we also require that there is at least one invariant in functional flows.

Recalling Section 4.1.1, invariants must hold only for all steps in $\{0, \ldots, \frac{\tau_j}{\Delta} - 1\}$. Therefore, at the end of the algorithm, the last step of the trajectory corresponds to the final sample point $\frac{\tau_j}{\Delta}$ and violates the invariants. Note that such a trajectory is never empty.

Here we ignore the corner cases where the function might fail, e.g. due to floating-point overflows. Function DO_STEP depends on the implementation of the selected numerical method.

The ODE solver currently requires exact initial values and does not explicitly support nondeterminism here—the only way to achieve that so far is to use Boolean constraints in a supervising algorithm, e.g. an SMT solver. For example, in the case of initial conditions in the form of intervals (Remark 5), it is possible to discretize the interval into sample points using disjunction of multiple equalities (i.e. simple initial conditions) and solve these cases separately. A future work is to implement an interval arithmetic (e.g. affine arithmetic) which would allow such nondeterminism of initial values.

**Stopping Conditions.** An issue with classical numerical ODE solvers is that they support only evaluations for a fixed time period (such as in Remark 2) which however does not depend on function values of the underlying functions. Some solvers (e.g. SUNDIALS) implement a form of root-finding, but only for some simple cases such as Examples 4.5 and 4.6. We apply more general stopping conditions of stepping methods which are based on invariants (Definition 12).

In Algorithm 6.1 we already showed how to check invariants, while we assume that they are always eventually violated, which is a reliable stopping condition (as a result of Remark 6). Moreover, the stopping happens exactly at the moment of a violation of any invariant, resulting in the longest possible lengths of functional flows—in the same way as for stages of functional flows (Section 4.1). Such stopping conditions are sufficient for implementing strong satisfaction of a formula (Definition 20). This way it is possible to use Algorithm 6.1 with no modifications, only checking the invariants within the ODE solver. The algorithm returns a list of vectors where the last one violates the invariants, and the responsibility of checking whether the last step satisfies final conditions lies on a supervising algorithm such as an SMT solver.

It is typical that subsequent stages of functional flows reuse the final values of functional variables, for instance in Example 4.6. The final values correspond to the point where invariants are already violated. However, it would require to use a very small step size $\Delta$ in order to achieve that the final values are accurate. Too small step sizes might cause significant performance drop, even though we just want to increase accuracy in the final phase of simulations. Therefore, once we reach a point where invariants are violated, we apply iterative bisection resulting in Algorithm 6.2. It searches for another point where invariants are violated too, but the point is close enough to the previous point where the invariants still held. The bisection is similar to the root-finding in SUNDIALS, but in our case the stopping conditions may be more general due to the usage of invariants.

The algorithm keeps searching with lower step sizes until a threshold $\epsilon$ is reached. Note that lower step sizes imply that the particular steps in the resulting trajectory $\boldsymbol{T}$ are no more equidistant wrt. time, but at the end of the list the sample points are more granular. We additionally check if invariants are violated at the beginning of the algorithm in which case the bisection is not desirable. Algorithm 6.2 terminates because Algorithm 6.1 terminates and $\delta$ converges to zero[1]. The resulting trajectory is unique wrt. the parameters of the algorithm.

We currently implement only numeric methods with a constant step size, but both Algorithms 6.1 and 6.2 can be upgraded to also handle methods with adaptive step sizes—it will require to control the invariants and the iterative bisection more carefully, though. Furthermore, it can happen during the adaptive stepping that a careless choice of a large step size causes skipping a region where an invariant would have been violated using a lower step size.

[1]We exclude the corner cases where $C$ is very close to the bounds of the interval $(0, 1)$, when it may round to the boundary values, wrt. a floating-point arithmetic.

**Algorithm 6.2:** Function ODE_SOLVE.

```
ODE_SOLVE($\boldsymbol{x_0}$, odes, $invs$, $\mathcal{P}$, $\Delta$) $\longrightarrow$ List[Vector[$\mathbb{F}$]] :
if $\neg$ALL_INVARIANTS_HOLD($\boldsymbol{x_0}$, $invs$) then
    return ($\boldsymbol{x_0}$)
$\boldsymbol{x} \leftarrow \boldsymbol{x_0}$
$\boldsymbol{T} \leftarrow ()$
$\delta \leftarrow \Delta$
loop
    $\boldsymbol{T} \leftarrow \boldsymbol{T} \parallel$ DO_STEPS($\boldsymbol{x}$, odes, $invs$, $\mathcal{P}$, $\delta$)          ▷ Algorithm 6.1
    if $\delta \leq \epsilon$ then                                                    ▷ $\epsilon > 0$
        break
    $\boldsymbol{x} \leftarrow T^{[|\boldsymbol{T}|-1]}$                ▷ forget the last step ($\boldsymbol{T}$ has at least two elements)
    $\boldsymbol{T} \leftarrow (T^{[1]}, \dots, T^{[|\boldsymbol{T}|-2]})$       ▷ forget the last step and avoid duplication of $\boldsymbol{x}$
    $\delta \leftarrow \delta \cdot C$                                                 ▷ $C \in (0, 1)$
return $\boldsymbol{T}$
```

## 6.2 SMT Solver

In this section, we present a standalone lazy SMT solver (see Sections 2.4 and 3.3) which decides formulas in the theory of ODEs (Section 4.2, Definition 18) with simulation semantics (Section 4.3, Definition 19). Deeper theoretical study of the target problem and definition of a corresponding SMT solver is available in our paper [75]. Here, we focus on an implementation of a solver that decides strong satisfiability (Definition 20) of formulas.

The SMT solver consists of a SAT solver and a theory solver. The particular solvers are more-or-less independent based on how tight is their integration. A part of the theory solver is the ODE solver from Section 6.1. We will describe the theory solver in Section 6.2.1, and Section 6.2.2 will follow with the discussion on integration of the theory solver with the underlying SAT solver.

We start with arguments why we built a brand new SMT solver from scratch instead of using an existing state-of-the-art solver with a real or a floating-point arithmetic from SMT-LIB (Section 5.3.1).

**Motivation.** Our ultimate goal is numerical solving (i.e. simulation) of IVPs (referring to Section 3.1) within an SMT framework, which implies that:

1. The underlying ODE solver computes with raw floating-point numbers, using cheap numeric computation and matrix operations.
2. Initial values of all ODEs must be defined before each simulation is executed.
3. The outputs of ODE solvers are approximations, as was discussed earlier.

A floating-point theory which concentrates on the intricacies of floating-point arithmetic might seem to be a reasonable choice. However, most of floating-point computation occurs in the ODE solver, referring to Item 1. ODE solvers are unaware of the precise formalization of the theory and use raw floating-point numbers. Hence we focus on the handling of ODEs using the native machine-friendly representation of floating-point numbers and we largely ignore the corner cases of the arithmetic. In the future, we will aim to support the corner cases with a rigorous explanation wrt. simulation semantics and the underlying numerical method.

Using an SMT solver with real arithmetic or alike is not suitable either because it represents real values symbolically or uses precise rational numbers which are not easily interchangeable with floating-point numbers. Moreover, in the case where most of the constraints are directly or transitively related to initial or final values of ODEs, then the necessity of algebraic solutions of constraints is limited, as a consequence of Items 2 and 3. In such cases, (nonlinear) real arithmetic is just too universal and hard to analyze.

Therefore, we focus on formulas where all terms transitively depend on the initial conditions based on simulations.

### 6.2.1 Theory Solver

Generally, $\mathcal{T}$-solvers, that is, theory solvers, decide satisfiability of a conjunction of theory literals of a given underlying theory $\mathcal{T}$. Here the theory is the theory of ODEs and the theory literals refer to Definition 18, Some theory literals may form functional flows which we handle using the ODE solver that we described in Section 6.1. We check strong satisfiability, but we may also return `unknown`, although not in cases that are needed to solve benchmark problems of the type found in the result of this thesis.

We will define the state space of the $\mathcal{T}$-solver which will stem from theory variables and define how to proceed from one state to another depending on the theory literals and the involved variables.

**Variables.** Our $\mathcal{T}$-solver uses variables and sorts that we defined in Section 4.2—numerical variables and functional variables (Definition 18). Since we also use simulation semantics (Section 4.3), numerical variables are also called floating-point variables (Definition 19) which we will assume that are based on a machine-friendly floating-point arithmetic such as IEEE 754.

In addition, for each functional variable $f$, the corresponding initial value $\mathit{init}(f)$ is not treated as just an expression but represents an additional fresh floating-point variable.

All the listed variables can in addition have a special value `undef` which is the default value at the beginning of the algorithm.

**Literals.** The $\mathcal{T}$-solver uses decides strong satisfiability (Definition 20) of a conjunction of theory literals of the theory of ODEs (Definition 18). The literals are passed to the $\mathcal{T}$-

solver as an input. They do not necessarily correspond to the original formula, but rather to their truth values according to the assignment of the SAT solver. For example, given numerical predicate $x > 0$ abstracted to a Boolean variable $p$, and given a Boolean variable $a$, formula $a \vee \neg(x > 0)$ is abstracted to $a \vee \neg p$. With assignment of the SAT solver $\{a \mapsto \bot, p \mapsto \bot\}$, the $\mathcal{T}$-solver receives negative theory literal $\neg p$ and checks its strong satisfiability. But with assignment $\{a \mapsto \top, p \mapsto \top\}$, the $\mathcal{T}$-solver receives positive literal $p$.

In practically reasonable formulas (such as Example 4.7), functional predicates occur only positively, without a negation. Hence, the $\mathcal{T}$-solver may ignore negative functional literals, instead of checking that the negation holds, and focus on positive functional literals.

**State.** A *state* of the $\mathcal{T}$-solver is an assignment to all theory variables defined above and occurring in the input formula. The *initial state* of the $\mathcal{T}$-solver is the state where all these variables have the value `undef`.

Next, we will define what is a consistent state which stems from the evaluation of numerical predicates.

**Definition 21.** *A numerical predicate is $\mathcal{T}$-*evaluable *iff none of the arguments that occur in the predicate have the value* `undef`*.*

**Definition 22.** *A state of the $\mathcal{T}$-solver is $\mathcal{T}$-*consistent *iff for all numerical literals that are $\mathcal{T}$-evaluable, the Boolean value of the literal is the same as the resulting Boolean value of numerical evaluation of the corresponding atomic theory formula, using the values of the variable assignment of the state. A state that is not $\mathcal{T}$-consistent is $\mathcal{T}$-*inconsistent.

*A numerical literal is $\mathcal{T}$-consistent wrt. a state of the $\mathcal{T}$-solver iff the state remains $\mathcal{T}$-consistent with the literal included in the assignment. Otherwise the literal is $\mathcal{T}$-inconsistent.*

According to the nature of functional literals and the way how invariants are defined, functional predicates themselves do not affect consistency.

**Inference Rules.** We apply *inference rules* on the states of the $\mathcal{T}$-solver in order to assign values to the variables of the solver. There are two kinds of inference rules depending on the sort of the involved variables. We start with inference rules on numerical variables where we exploit the fact that both sides of an equality have to evaluate to the same value.

**Definition 23.** *A* numerical inference rule *can be applied if the input formula contains a positive numerical literal of the form $x = \eta$ or $\eta = x$, where $x$ is a floating-point variable with the value* `undef` *and $\eta$ is a term where none of the arguments have the value* `undef`*.*

*The inference rule changes the current state s.t. it assigns variable $x$ to the value that corresponds to the numerical evaluation of term $\eta$.*

For example in $x^{[j]} = x^{[j-1]} + \mathit{final}(y_{j-1})$, if the value of variable $x^{[j-1]}$ and final value of variable $y_{j-1} \in \mathcal{F}_{j-1}$ is already fixed, then we can infer the value of variable $x^{[j]}$.

**Definition 24.** *A* functional inference rule *can be applied if the set of functional predicates and simple initial conditions corresponding to a system of functions $\mathcal{F}$ and occurring in positive literals of the input formula forms a functional flow such that:*

- *the flow contains at least one invariant,*
- *all functional variables $f \in \mathcal{F}$ have the value* `undef`*,*
- *for all $f \in \mathcal{F}$ the corresponding numerical variable $init(f)$ does not have the value* `undef`*,*
- *every numerical variable that appears within any of the functional predicates of the flow does not have the value* `undef`*.*

*The resulting state after applying such an inference rule is identical to the previous one except that all variables $f \in \mathcal{F}$ are assigned to the resulting trajectories of the ODE solver as described in Algorithm 6.2.*

For example, if the initial value of variable $y_j \in \mathcal{F}_j$ (i.e. $init(y_j)$) and the value of variable $x_j$ is fixed, then we can use $\dot{y}_j = x^{[j]}$ and $y_j \leq 10$ to infer the value of the variable $y_j$, that is, the corresponding trajectory with $final(y_j) > 10$ (a value that is close to 10 but strictly greater). Here we assume that $x^{[j]} > 0$.

Note that this inference rule differs from the one that we defined in [75] where the rule fixes the number of steps. Using that inference rule would require tighter integration of the SMT solver and the ODE solver. On the other hand, in [75] we do not require that invariants must be violated and the length of the systems does not have to be the longest. For instance, in the example above, the final value of trajectories of that solver would satisfy $final(y_j) \geq init(y_j)$.

Both numerical and functional inference rules yield unique successor states of the $\mathcal{T}$-solver. We can apply several inference rules in a row, starting from the initial state. This always terminates, since every inference rule creates a state with less undefined elements and the number of variables is finite.

**Algorithm and Result.** The $\mathcal{T}$-solver accepts a formula $\Phi$ as an input that is a conjunction of theory literals. We show a simple algorithm of the solver in Algorithm 6.3 which uses the fact that the order of the applications of inference rules does not affect a discovery of a $\mathcal{T}$-inconsistent state.

It must always check all literals in order to avoid missing a check of a literal that is currently not $\mathcal{T}$-evaluable. The variables of the $\mathcal{T}$-solver are hidden here.

The algorithm returns `sat` if $\Phi$ is (strongly) satisfiable and `unsat` if $\Phi$ is not strongly satisfiable. However, it may return `unknown` in cases when at the end there are some variables that have the value `undef`. Therefore, the theory solver is not complete in general. However, in [75] we provide a syntactical characterization of the kind of inputs for which our solver provides an efficient solution. We support this through experiments presented in Chapter 7 and Chapter 8.

**Algorithm 6.3:** Function $\mathcal{T}$::SOLVER.

```
$\mathcal{T}$::SOLVER($\Phi$) $\longrightarrow$ {sat, unsat, unknown} :

let $\sigma$ be the initial state for $\Phi$
while $\sigma$ is $\mathcal{T}$-consistent $\wedge$ an inference rule can be applied on $\sigma$ wrt. $\Phi$ do
    let $\sigma'$ be the state after applying the inference rule on $\sigma$ wrt. $\Phi$
    $\sigma \leftarrow \sigma'$
if $\sigma$ is $\mathcal{T}$-inconsistent then
    return unsat
if $\sigma$ contains a variable with the value undef then
    return unknown
return sat
```

If the result is `sat`, then formula $\Phi$ is indeed strongly satisfiable (Definition 20), not only satisfiable. The reason is that here all variables must be assigned to a value other than `undef`, including functional variables. The only way how to assign a value to a functional variable is to use a functional inference rule, which requires at least one positive literal of an invariant. In addition, in the ODE solver we assume that some invariant must always be eventually violated. Therefore, for each system of functions, a functional inference rule must have been used where an invariant must have been violated. Moreover, since there is a satisfying variable assignment $\alpha$, it also means that $\alpha$ must satisfy all selected invariants. Therefore, given these assumptions and given that $\mathcal{T}$-solver returns `sat`, every such variable assignment that satisfies $\Phi$ also strongly satisfies $\Phi$.

If the result is `unsat`, then the input formula $\Phi$ is not strongly satisfiable. If $\Phi$ is of a form that ensures that satisfiability and strong satisfiability coincide, this implies that it is not satisfiable. See the discussion after Definition 20 including Example 4.8 for details.

## 6.2.2 Solver Integration

Now we discuss how to build the SMT solver based on an integration of the presented theory solver (i.e. $\mathcal{T}$-solver) in Section 6.2.1 and an underlying SAT solver. Following Section 3.3.2, we use the lazy approach to SMT where atomic theory formulas are substituted with fresh Boolean variables. The SMT solver checks strong satisfiability of a formula in a similar way as the $\mathcal{T}$-solver, also having the same result. But importantly, now also Boolean variables and disjunctions of constraints are allowed, for which we use the SAT solver.

The SAT solver controls the core algorithm, but by itself, it does not have the theory-related information on these Boolean variables and must communicate with the theory solver. How frequent is the communication between the solvers depends on how tightly they are integrated. We described the principles of lazy offline and online approaches. We first describe a so-called dependency graph that will be useful in the algorithms. Next, we proceed to the naive offline approach which is useful for illustrating some techniques of lazy solving. Finally, we present our online approach.

**Dependency Graph.** Atomic theory formulas form vertices of a directed *dependency graph*, where an edge means that the source vertex may allow an inference rule that assigns a value to a variable that is shared with the target vertex. Inference rules that may be allowed by vertices that have no input edges are *initial* inference rules. By definition, initial inference rules can be only numerical inference rules.

The dependency graph is useful for decision heuristics of the SMT solver. For example, it is usually beneficial to use initial inference rules before other inference rules, since they do not require any variables to be assigned. Then, it is often efficient to proceed via the edges of the dependency graph that lead from the already processed vertices to those that are not yet processed. The graph is also useful for construction of conflict clauses, as we will see below.

**Offline Approach.** A naive, shallow integration of the underlying solvers uses the solvers as independent components with no callbacks or interruptions: the SAT solver searches for a full propositional assignment and the $\mathcal{T}$-solver checks it for strong satisfiability. The approach results in an unnecessarily high number of discovered inconsistencies, but it is easy to implement and also allows flexible selections of particular implementations of the underlying solvers.

*Notation.* Boolean variables are represented by set $\mathit{bools}$, numerical (i.e. floating point) variables by set $\mathit{floats}$, and atomic theory formulas (abstracted to Boolean variables) by sets $\mathit{flt\_preds}$ and $\mathit{fun\_preds}$, referring respectively to numerical predicates and functional predicates. Each system of functions $\mathcal{F}_j$ is ordered into a vector of functional variables $\boldsymbol{F}_j$, where $j \in \mathcal{J}$ and $\mathcal{J}$ is the index set from Definition 18. All such systems are grouped altogether such that $\boldsymbol{F} := \{\boldsymbol{F}_j \mid j \in \mathcal{J}\}$. Variables that represent initial values of functional variables are included in the set $\mathit{floats}$ as well.

Algorithm 6.4 shows an outline of such an approach which decides whether a formula $\Phi$ is strongly satisfiable. In CHECK_SAT, it firstly checks propositional satisfiability of $\Phi$. If $\Phi$ is unsatisfiable, the algorithm terminates, otherwise it searches for a full propositional assignment. This part corresponds purely to the SAT solver. The rest of the algorithm corresponds to the $\mathcal{T}$-solver. After getting the assignment, it tries to evaluate all floating-point variables and functional variables inside function TRY_EVAL_-FLOATS, where it also checks consistency of the current assignment.

If checking of consistency in TRY_EVAL_FLOATS fails, a conflict clause $\chi$ is constructed using CONFLICT_CLAUSE and appended to formula $\Phi$. Then, the procedure restarts from scratch. The algorithm keeps looping until it arrives at a definite result, referring to the possible results described in Section 6.2.1.

**Algorithm 6.4:** Function NAIVE_SAT.

```
NAIVE_SAT($\Phi$, $bools$, $floats$, $flt\_preds$, $\boldsymbol{F}$, $fun\_preds$) $\longrightarrow$
  {sat, unsat, unknown} :
loop
    UNSET_VALUES($bools \cup floats \cup flt\_preds \cup fun\_preds$, $\boldsymbol{F}$)
                                                  ▷ set all variables to undef
    ▷ SAT solver
    if ¬CHECK_SAT($\Phi$, $bools$, $flt\_preds$, $fun\_preds$) then
        return unsat                              ▷ failure: $\Phi$ is unsatisfiable
    $\boldsymbol{A}$ ← GET_ASSIGNMENT($\Phi$, $bools$, $flt\_preds$, $fun\_preds$)
    ▷ $\mathcal{T}$-solver
    if ¬TRY_EVAL_FLOATS($\boldsymbol{A}$, $floats$, $flt\_preds$, $\boldsymbol{F}$, $fun\_preds$) then ▷ Algorithm 6.5
        $\chi$ ← CONFLICT_CLAUSE($\boldsymbol{A}$, $floats$, $flt\_preds$, $\boldsymbol{F}$, $fun\_preds$)
        $\Phi \leftarrow \Phi \land \chi$                         ▷ asserts the conflict clause
        continue
    if ¬ALL_SET($floats$, $\boldsymbol{F}$) then
        return unknown                     ▷ failure: some variables not set
    return sat                              ▷ success: all variables are set!
```

**Algorithm 6.5:** Function TRY_EVAL_FLOATS.

```
TRY_EVAL_FLOATS($\boldsymbol{A}$, $floats$, $flt\_preds$, $\boldsymbol{F}$, $fun\_preds$) $\longrightarrow \mathbb{B}$ :
repeat
    forall $p \in flt\_preds$ do                  ▷ no functional predicates here
        if ¬NUM_CONSISTENT_TRY_INFER($p$, $\boldsymbol{A}$, $floats$) then     ▷ Algorithm 6.6
            return $\bot$                              ▷ found a $\mathcal{T}$-inconsistency
    forall $\boldsymbol{F}_j \in \boldsymbol{F}$ do
        TRY_FUN_INFERENCE_RULE($\boldsymbol{F}_j$, $\boldsymbol{A}$, $fun\_preds$, $floats$)     ▷ Algorithm 6.7
until NO_PROGRESS($floats$, $\boldsymbol{F}$)
return $\top$
```

Algorithm 6.5 describes function TRY_EVAL_FLOATS, i.e. evaluation of the non-Boolean part: numerical variables *floats* and functional variables $\boldsymbol{F}$, which either succeeds or not. Checking of $\mathcal{T}$-consistency and at the same time the evaluation of numerical variables is done by function NUM_CONSISTENT_TRY_INFER, using numerical inference rules. In the case of an inconsistency, the function returns false. The evaluation of functional variables is done by procedure TRY_FUN_INFERENCE_RULE. The loop tries to assign as many variables as possible according to the corresponding inference rules, as long as there is a progress, that is, at least one inference rule must be used every time.

**Algorithm 6.6:** Function NUM_CONSISTENT_TRY_INFER.

```
NUM_CONSISTENT_TRY_INFER$(p, \boldsymbol{A}, floats) \longrightarrow \mathbb{B}$ :
if $\neg$ANY_UNDEF_ARG$(p, floats)$ then
    $val \leftarrow$ SAT_VALUE$(p, \boldsymbol{A})$   ▷ truth value of the corresponding Boolean variable
    $eval \leftarrow$ EVAL$(p, floats)$   ▷ Boolean value of the evaluation of the predicate
    return $val \Leftrightarrow eval$
if SYMBOL$(p) \in \{=\}$ then
    TRY_NUM_INFERENCE_RULE$(p, \boldsymbol{A}, floats)$   ▷ Definition 23
return $\top$
```

Function NUM_CONSISTENT_TRY_INFER is described in Algorithm 6.6. According to propositional assignment $\boldsymbol{A}$, it either checks $\mathcal{T}$-consistency of numerical predicate $p$, or tries to apply the numerical inference rule from Definition 23. Checking of inconsistency is possible only for predicates where all floating point values are already fixed. Then it compares the Boolean values of assignment $\boldsymbol{A}$ and of the numerical evaluation of the predicate, and returns true iff the values match. If not checking the consistency, it returns true because a predicate that is not evaluable or used in an inference rule cannot become $\mathcal{T}$-inconsistent.

Algorithm 6.7 describes procedure TRY_FUN_INFERENCE_RULE which tries to evaluate a functional flow using the functional inference rule from Definition 24 and by solving the flow using function ODE_SOLVE (Algorithm 6.2). It basically just checks if it is possible to apply the inference rule and arranges initial values and the appropriate positive functional literals based on propositional assignment $\boldsymbol{A}$ for the function ODE_SOLVE. Still, an adaptor function ODE_SOLVE* is used instead of ODE_SOLVE in order to also find floating point variables that appear inside **odes** and *invs* and to use their values—because the ODE solver does not use numerical variables, only constant values.

Going back to function NAIVE_SAT (Algorithm 6.4), function TRY_EVAL_FLOATS can discover a $\mathcal{T}$-inconsistency. Construction of conflict clause $\chi$ is done via CONFLICT_-CLAUSE based on the dependency graph: the clause is filled with the inconsistent literal and also all theory literals that correspond to the inference rules that affected the values

**Algorithm 6.7:** Procedure TRY_FUN_INFERENCE_RULE.

```
TRY_FUN_INFERENCE_RULE($\boldsymbol{F}_j, \boldsymbol{A}, \mathit{fun\_preds}, \mathit{floats}$) :

$\boldsymbol{x_0} \leftarrow ()$
odes $\leftarrow ()$
forall $f_j$ in $\boldsymbol{F}_j$ do
    if VALUE($f_j$) $\neq$ undef then return
    if VALUE($\mathit{init}(f_j)$) $=$ undef then return

    pos_odes $\leftarrow$ POSITIVE_ODE_LITERALS($f_j, \boldsymbol{A}, \mathit{fun\_preds}$)
    if $|$pos_odes$| \neq 1$ then return                      ▷ can handle only exactly one ODE
    if ANY_UNDEF_ARG(pos_odes$^{[1]}$, $\mathit{floats}$) then return   ▷ use just the one ODE

    $\boldsymbol{x_0} \leftarrow \boldsymbol{x_0} \parallel (\mathit{init}(f_j))$                  ▷ append the initial value
    odes $\leftarrow$ odes $\parallel$ pos_odes                            ▷ append the ODE

$\mathit{invs} \leftarrow$ POSITIVE_INV_LITERALS($\boldsymbol{F}_j, \boldsymbol{A}, \mathit{fun\_preds}$)
if $|\mathit{invs}| = 0$ then return                   ▷ would not terminate with no invariants
forall $\mathit{inv} \in \mathit{invs}$ do                                    ▷ order does not matter
    if ANY_UNDEF_ARG($\mathit{inv}, \mathit{floats}$) then return

$\boldsymbol{F}_j \leftarrow$ ODE_SOLVE*($\boldsymbol{F}_j, \boldsymbol{x_0}$, odes, $\mathit{invs}, \mathit{floats}$)
 ▷ using ODE_SOLVE (Algorithm 6.2)
```

of arguments of the inconsistency. This can be done by traversing from the vertex in the dependency graph that corresponds to the inconsistency back to initial inference rules, using the fact that the edges of the graph connect vertices to predicates that allow inference rules. The resulting conflict clause $\chi$ forms negation of the current assignment of the SAT solver, but only selecting the atomic theory formulas that directly participated in the inconsistency.

**Example 6.1.** Let $p_0^=, p_1^=, p_2^=, p_2^\leq$ be atomic theory formulas s.t.

$$\begin{aligned} p_0^= &:= x^{[0]} = 0, \\ p_1^= &:= x^{[1]} = x^{[0]} + 1, \\ p_2^= &:= x^{[2]} = x^{[1]} + 1, \\ p_2^\leq &:= x^{[2]} \leq 0 \end{aligned} \tag{6.1}$$

where $x^{[0]}, x^{[1]}, x^{[2]}$ are numerical variables. An assignment $\{p_0^= \mapsto \top, p_1^= \mapsto \top, p_2^= \mapsto \top, p_2^\leq \mapsto \top\}$ results in a $\mathcal{T}$-inconsistency of the theory literal $p_2^\leq$, because $x^{[2]} = 2 \nleq 0$. The corresponding conflict clause $\chi$ is

$$\neg p_0^= \vee \neg p_1^= \vee \neg p_2^= \vee \neg p_2^\leq \tag{6.2}$$

which is equivalent to $\neg \left(p_0^= \wedge p_1^= \wedge p_2^= \wedge p_2^\leq\right)$.

**Analysis of the Offline Approach.** We experimented with such an offline approach, but it is very difficult to avoid the great number of undesirable $\mathcal{T}$-inconsistencies, which was mentioned above. A possible improvement is to eagerly preprocess the formula and append additional propositional constraints that are related to existing atomic theory formulas. For example, one can add pairwise conflict clauses with numerical predicates which are equalities where one of the sides evaluates to a constant, such as $(x \neq 1 \vee x \neq 2) \wedge (x \neq 1 \vee x \neq 3) \wedge (x \neq 2 \vee x \neq 3)$. However, this works only if one of the terms are somehow easily computable within the preprocessing stage. Also, even in the case when there is a lot of such equalities with constants, such eager approach is still less efficient than a lazy online approach with theory propagation: For $n$ such equalities, the number of conflict clauses in the case of the preprocessing would be $\frac{n(n-1)}{2}$, and many of the clauses would likely not be useful. With an online approach, after the SAT solver assigns one of these equalities to true, it is sufficient to just lookup somehow the rest $n-1$ equalities and assign them to $\perp$.

We also observed that the offline approach arrives at a lot of unnecessary $\mathcal{T}$-inconsistencies in the case of formulas with many if-then-else constraints, such as in Example 4.7. Consider an example of a formula with an unrolling similar to BMC, where each stage contains $\text{ITE}(a^{[j]},\ x^{[j]} = y^{[j]},\ x^{[j]} = z^{[j]})$. Suppose that usually $y^{[j]} \neq z^{[j]}$, but not in general (so we cannot use preprocessing). This however means that the inconsistency $x^{[j]} = y^{[j]} \wedge x^{[j]} = z^{[j]} \wedge y^{[j]} \neq z^{[j]}$ will appear quite often, because of the way how constraints with an implication work, where the SAT solver is naturally free to make such assignments. With increasing number of steps of the unrolling, such an issue becomes significant even in the case of fairly trivial formulas. The discouraging fact here is that there is no suitable way how to prevent this from hapenning within the offline approach, because it requires to do it within the preprocessing stage.

**Example 6.2.** In order to demonstrate the inefficiency of the offline approach, we present an example similar to the model of bouncing ball in Example 4.7, which is again deterministic but in addition without any differential equations:

$$\begin{aligned}
& X = 5 \wedge \xi = 0.35X \\
& \wedge \neg \text{up}^{[1]} \wedge x^{[1]} = X \\
& \wedge \bigwedge_{j=1}^{J} \text{ITE}\left(\text{up}^{[j]},\ x'^{[j]} = x^{[j]} + \xi,\ x'^{[j]} = x^{[j]} - \xi\right) \\
\wedge \bigwedge_{j=1}^{J-1} \text{ITE}\Big( & \text{up}^{[j]}, \text{ITE}\left(x'^{[j]} < X,\ \text{up}^{[j+1]} \wedge x^{[j+1]} = x'^{[j]},\ \neg\text{up}^{[j+1]} \wedge x^{[j+1]} = X\right), \\
& \text{ITE}\left(x'^{[j]} > 0,\ \neg\text{up}^{[j+1]} \wedge x^{[j+1]} = x'^{[j]},\ \text{up}^{[j+1]} \wedge x^{[j+1]} = 0\right)\Big)
\end{aligned} \tag{6.3}$$

where $\mathrm{up}^{[j]}$ are again the Boolean variables, $x^{[j]}$ and $x'^{[j]}$ are numerical variables that model the height of the ball at the beginning and at the end of stage $j$, respectively, and $X, \xi$ are rational constants. Even in this simplified case, the modeled ball keeps bouncing up and falling down with constant velocity.

The deterministic formula is easily satisfiable and can be simulated in a straightforward way with no Boolean search. Therefore, the procedure should not arrive at any $\mathcal{T}$-inconsistency at all. This can be done with an *online* approach with theory propagation, where *only* a sequence of unit propagation and theory propagation rules are sufficient to arrive at the result, that is, with no decisions of the SAT solver. However, this is not the case of the offline approach: the underlying SAT solver also uses unit propagations, but without the guidance from theory propagations it always arrives at a point where it has no clue which way to go within particular if-then-else branches, and has to decide some variables, which inevitably leads to conflicts. Let's consider at least some preprocessing, for instance $\neg(x'^{[j]} = x^{[j]} + \xi \wedge x'^{[j]} = x^{[j]} - \xi)$. Then, the example will be processed as follows:

1. Unit propagation: $\mathrm{up}^{[1]} \mapsto \bot$, $(x^{[1]} = X) \mapsto \top$.
2. Unit propagation: $\neg\mathrm{up}^{[1]} \Rightarrow x'^{[1]} = x^{[1]} - \xi$ implies $(x'^{[1]} = x^{[1]} - \xi) \mapsto \top$.
3. Unit propagation: $x'^{[1]} = x^{[1]} - \xi$ implies $(x'^{[1]} = x^{[1]} + \xi) \mapsto \bot$.
4. (No inference rules nor theory propagations were applied—the SAT solver searches for a full propositional assignment with no interruptions.)
5. No other rules can be applied on the abstracted model. Furthermore, the value of variable $x'^{[1]}$ is still `undef`. Thus, the SAT solver does not know which branch of the condition $x'^{[j]} > 0$ to select and must make a decision.

Until now we were still inside the first call of function CHECK_SAT in Algorithm 6.4, but it is already clear that most likely there will be a number of inconsistencies. In our experiments, the number of conflicts (i.e. runs of the SAT solver) for $J = 100$ was 904. An example of such a conflict is $x^{[1]} = X \wedge x'^{[1]} = x^{[1]} - \xi \wedge \neg(x'^{[1]} < X)$.

Using an online approach with theory propagation (without preprocessing), the corresponding run can look as follows:

1. Unit propagation: $\mathrm{up}^{[1]} \mapsto \bot$, $(x^{[1]} = X) \mapsto \top$.
2. Theory propagation: $x^{[1]} = X$ allows inference rule $x^{[1]} \mapsto X$.
3. Unit propagation: $\neg\mathrm{up}^{[1]} \Rightarrow x'^{[1]} = x^{[1]} - \xi$ implies $(x'^{[1]} = x^{[1]} - \xi) \mapsto \top$.
4. Theory propagation: $x'^{[1]} = x^{[1]} - \xi$ allows inference rule $x'^{[1]} \mapsto x^{[1]} - \xi = X - \xi$. Consequently, it ensures $\mathcal{T}$-consistency by $(x'^{[1]} = x^{[1]} + \xi) \mapsto \bot$, $(x'^{[1]} < X) \mapsto \top$ and $(x'^{[1]} > 0) \mapsto \top$.

5. Unit propagation: $\neg\mathrm{up}^{[1]} \Rightarrow (x'^{[1]} > 0 \Rightarrow \neg\mathrm{up}^{[2]} \wedge x^{[2]} = x'^{[1]})$ implies $\mathrm{up}^{[2]} \mapsto \bot$ and $(x^{[2]} = x'^{[1]}) \mapsto \top$.

6. Theory propagation: $x^{[2]} = x'^{[1]}$ allows inference rule $x^{[2]} \mapsto x'^{[1]} = X - \xi$. Consequently, it ensures $\mathcal{T}$-consistency by $(x^{[2]} = X) \mapsto \bot$ and $(x^{[2]} = 0) \mapsto \bot$.

7. Unit propagation of $\neg\mathrm{up}^{[2]} \Rightarrow x'^{[2]} = x^{[2]} - \xi \ldots$

This way, the steps above ensures that not only all variables of the first stage (i.e. $\mathrm{up}^{[1]}$, $x^{[1]}$ and $x'^{[1]}$) are set, but also all numerical predicates. Therefore, such an approach covers all variables, including the abstracted Boolean variables, in a deterministic way and proceeds efficiently up to a final stage $J$.

---

**Online Approach.** Following Section 3.3.2, and according to the discussion above, we apply (lazy) online approach with exhaustive theory propagation along with consistency checks. In the case of our theory which is based on simulations, it is indeed suitable to propagate and check theory constraints exhaustively, because numerical evaluations are cheap operations[2]. Moreover, the propagations and the checking are sufficient to perform only in connection with inference rules, which introduce new values. After introducing new values, in the dependency graph, it is sufficient to traverse all neighbors of the predicates that enabled the inference rule.

For example, given numerical predicates $x > 0$ and $x = 0$, and with $x$ still being `undef`, assignment $(x = 0) \mapsto \top$ will allow the inference rule from Definition 23. Consequently, the assignment will either result in theory propagation $(x > 0) \mapsto \bot$, if $x > 0$ is still unassigned by the SAT solver, ensuring $\mathcal{T}$-consistency. Or, if $x > 0$ is already assigned, then it will be checked for consistency (as in Algorithm 6.6).

Still, constraints that are currently not evaluable cannot be propagated nor checked for consistency. That is, constraints where an argument with the value `undef` appears cannot be checked for consistency, and constraints where no inference rule can be applied cannot be propagated. For example, the predicate $x > 0$ does not allow any inference rule. And if the value of $x$ is still `undef`, the predicate will not be checked for consistency neither. This corresponds to the observation above that the checks are sufficient to perform in connection with an application of an inference rule.

Therefore, it is also important to allow specific suggestions for decisions. For example, to prefer decisions on predicates that allow inference rules. Note that such a case requires not only to select an appropriate decision variable, but also the value of the decision, because in inference rules it is necessary that the truth value of the involved predicates is true.

[2]Simulations of ODEs are actually not that cheap, but we currently do not have evidence that postponing them within theory propagation would be beneficial.

In contrast to the offline approach, now the algorithm is controled entirely by the SAT solver, but at the same time the interface allows some callbacks to the $\mathcal{T}$-solver. The outline of the procedure is presented in Algorithm 6.8.

**Algorithm 6.8:** Function ONLINE_SAT.

```
ONLINE_SAT($\Phi$, $bools$) $\longrightarrow$ {sat, unsat, unknown} :
$\boldsymbol{A} \leftarrow ()$                                    ▷ start with an empty assignment
$d \leftarrow 0$                                    ▷ start with zero decision level
loop
    $(ok, \boldsymbol{P}) \leftarrow$ PROPAGATE($\Phi$, $bools$, $\boldsymbol{A}$)        ▷ Algorithm 6.9, $\mathcal{T}$::PROPAGATE
    if $ok$ then
        $\boldsymbol{A} \leftarrow \boldsymbol{A} \parallel \boldsymbol{P}$        ▷ append the propagations into the assignment
        if $|\boldsymbol{A}| = |bools|$ then break
        $\boldsymbol{A} \leftarrow \boldsymbol{A} \parallel$ (DECIDE($\Phi$, $bools$, $\boldsymbol{A}$))        ▷ $\mathcal{T}$::SUGGEST
        $d \leftarrow d + 1$
        continue
    if $d = 0$ then return unsat                ▷ no decisions have been made
    $(\chi, n) \leftarrow$ ANALYZE_CONFLICT($\Phi$, $bools$, $\boldsymbol{A}$, $\boldsymbol{P}$)        ▷ maybe $\mathcal{T}$::EXPLAIN
    $\Phi \leftarrow \Phi \wedge \chi$
    $\boldsymbol{A} \leftarrow$ BACKJUMP($\boldsymbol{A}$, $\boldsymbol{P}$, $d$, $n$)        ▷ $\mathcal{T}$::BACKJUMP
    $d \leftarrow d - n$
if $\mathcal{T}$::UNKNOWN($\boldsymbol{A}$) then return unknown
return sat
```

The algorithm indeed looks similar to an algorithm of a regular SAT solver—it only concentrates on propositional constraints and variables, but the most important parts of the procedure are extended of callbacks to the $\mathcal{T}$-solver:

- $\mathcal{T}$::PROPAGATE is inserted into function PROPAGATE.
- $\mathcal{T}$::SUGGEST allows theory suggestions for decisions of Boolean variables.
- $\mathcal{T}$::EXPLAIN serves for propositional explanations of theory conflicts within function ANALYZE_CONFLICT. Similarly, $\mathcal{T}$::LEARN serves for explaining and learning a theory fact, but within function $\mathcal{T}$::PROPAGATE.
- $\mathcal{T}$::BACKJUMP keeps the $\mathcal{T}$-solver synchronized with the SAT solver in case it is backtracking after a conflict, using function BACKJUMP.
- $\mathcal{T}$::UNKNOWN checks whether all theory variables (i.e., *floats* and $\boldsymbol{F}_j$) are assigned to a value in the case of a satisfiable Boolean assignment, otherwise the result is `unknown`.

The variables *floats* and $\boldsymbol{F}_j$ are hidden from the SAT solver and are handled solely by the $\mathcal{T}$-solver. We will discuss functions PROPAGATE and $\mathcal{T}$::PROPAGATE (Algorithm 6.9 and 6.10) in more detail, and then also functions $\mathcal{T}$::SUGGEST, $\mathcal{T}$::EXPLAIN and $\mathcal{T}$::-LEARN. The principles of online lazy solvers were already discussed in Section 3.3.2. Our theory solver must keep track with the current assignment of the SAT solver, including backtracking. We arrange all assignments into decision levels $d$, because non-deterministic choices stem only from decisions, not propagations. Therefore, when the solver arrives at a conflict (which will be discussed later), it must backtrack to a previous point that corresponds to a lower decision level and revert all decisions and propagations down to this point. This is done by BACKJUMP with a callback to $\mathcal{T}$::BACKJUMP. There, it must remember which inference rule infered the value for each theory variable at the corresponding decision levels in order to correctly reconstruct the previous states.

**Algorithm 6.9:** Function PROPAGATE.

```
PROPAGATE($\Phi$, $bools$, $\boldsymbol{A}$) $\longrightarrow \mathbb{B} \times$ List[Assignment($\mathbb{B}$)] :
loop
    $(ok, \boldsymbol{P}) \leftarrow$ SAT::PROPAGATE($\Phi, bools, \boldsymbol{A}$)        ▷ e.g. unit propagations
    if $\neg ok$ then return $(\bot, \boldsymbol{P})$        ▷ encountered a propositional conflict
    $(ok, \boldsymbol{P}_\mathcal{T}) \leftarrow \mathcal{T}$::PROPAGATE($bools, \boldsymbol{A}, \boldsymbol{P}$)        ▷ Algorithm 6.10
    $\boldsymbol{P} \leftarrow \boldsymbol{P} \parallel \boldsymbol{P}_\mathcal{T}$
    if $\neg ok$ then return $(\bot, \boldsymbol{P})$        ▷ encountered a $\mathcal{T}$-inconsistency
    if $|\boldsymbol{P}_\mathcal{T}| = 0$ then return $(\top, \boldsymbol{P})$        ▷ no more propagations possible with $\boldsymbol{A}$
```

Function PROPAGATE is an enhancement of propositional propagations which are represented by SAT::PROPAGATE. It uses for example unit propagations, which is an important rule of DPLL algorithms, but modern SAT solvers use also many further efficient propositional techniques. After this, propagations $\boldsymbol{P}$ are sent into the $\mathcal{T}$-solver via $\mathcal{T}$::PROPAGATE, which can add further propagations that are related to the theory. However, at the same time it also checks whether the propositional assignment is $\mathcal{T}$-consistent, which may fail. Theory propagations $\boldsymbol{P}_\mathcal{T}$ may also enable more propositional propagations, and so on, which is the reason why the loop is important in the algorithm. The function returns a success flag and all propagations $\boldsymbol{P}$, that is, a list of assignments to Boolean variables that stem from the current Boolean assignment $\boldsymbol{A}$.

**Theory Propagation.** Theory propagations along with $\mathcal{T}$-consistency checks are handled in function $\mathcal{T}$::PROPAGATE. Propositional propagations $\boldsymbol{P}$ that correspond to atomic theory formulas are checked for consistency and in the case of an applied inference rule also theory-propagated. The function returns a success flag and all theory propagations $\boldsymbol{P}_\mathcal{T}$ (i.e. a list of propositional assignments to theory predicates) that stem from the current assignment $\boldsymbol{A}$ and propagations $\boldsymbol{P}$.

**Algorithm 6.10:** Function $\mathcal{T}$::PROPAGATE.

```
$\mathcal{T}$::PROPAGATE$(bools, \boldsymbol{A}, \boldsymbol{P}) \longrightarrow \mathbb{B} \times$ List[Assignment($\mathbb{B}$)] :
$\boldsymbol{P}_{\mathcal{T}} \leftarrow ()$
forall $p$ in $\boldsymbol{P}$ do
    if $\neg$IS_PRED$(p)$ then          ▷ only theory predicates can be theory-propagated
        continue
    $(ok, \boldsymbol{P}_{\mathcal{T}}{}^*) \leftarrow \mathcal{T}$::CONSISTENT_TRY_INFER$(p)$          ▷ maybe $\mathcal{T}$::LEARN
        ▷ similar to Algorithm 6.6, but also considering functional inference rules
    $\boldsymbol{P}_{\mathcal{T}} \leftarrow \boldsymbol{P}_{\mathcal{T}} \parallel \boldsymbol{P}_{\mathcal{T}}{}^*$
    if $\neg ok$ then return $(\bot, \boldsymbol{P}_{\mathcal{T}})$
    if $\neg$INFERENCED$(p)$ then
                    ▷ did $\mathcal{T}$::CONSISTENT_TRY_INFER use an inference rule on $p$?
        continue
    $(ok, \boldsymbol{P}_{\mathcal{T}}{}^*) \leftarrow$ PROPAGATE_INFERENCED$(p, bools, \boldsymbol{A}, \boldsymbol{P}, \boldsymbol{P}_{\mathcal{T}})$
                                                            ▷ maybe $\mathcal{T}$::LEARN
    $\boldsymbol{P}_{\mathcal{T}} \leftarrow \boldsymbol{P}_{\mathcal{T}} \parallel \boldsymbol{P}_{\mathcal{T}}{}^*$
    if $\neg ok$ then return $(\bot, \boldsymbol{P}_{\mathcal{T}})$
return $(\top, \boldsymbol{P}_{\mathcal{T}})$
```

Function $\mathcal{T}$::CONSISTENT_TRY_INFER not only checks $\mathcal{T}$-consistency, but also may assign values to theory variables with applying an inference rule. It is similar to Algorithm 6.6, but it does not apply only to numerical predicates, but also to functional predicates—concretely to functional ODEs, because invariants are ignored here. In case predicate $p$ is a functional ODE with true truth value, the corresponding functional flow is checked whether it is viable for functional inference rule (Definition 24), resulting in something similar to Algorithm 6.7. Note that here the propositional assignment is usually not full (compared to the offline approach), so it may often happen that some parts of the flow are not assigned yet.

If $\mathcal{T}$::CONSISTENT_TRY_INFER arrives at a $\mathcal{T}$-inconsistency, function $\mathcal{T}$::PROPAGATE returns false (and the current theory propagations). Before returning, it may or may not immediately learn the corresponding conflict clause, using function $\mathcal{T}$::LEARN in a similar way as in function CONFLICT_CLAUSE in Algorithm 6.4. Either way, it heads to function ANALYZE_CONFLICT in Algorithm 6.8, where the conflict clause has to be analyzed and learned anyway. If $\mathcal{T}$::LEARN was not used, then ANALYZE_CONFLICT would use $\mathcal{T}$::EXPLAIN. In this case of $\mathcal{T}$-inconsistencies, both the options are very similar, though. Functions $\mathcal{T}$::LEARN and $\mathcal{T}$::EXPLAIN will be discussed further later.

Function PROPAGATE_INFERENCED is applied if predicate $p$ participated in an inference rule in $\mathcal{T}$::CONSISTENT_TRY_INFER. It checks all neighbors of predicate $p$ in the dependency graph, in a similar way as in function $\mathcal{T}$::PROPAGATE: it checks whether they are $\mathcal{T}$-consistent and possibly propagates further inference rules using $\mathcal{T}$::CONSISTENT_TRY_INFER. This means that PROPAGATE_INFERENCED may call itself recursively. The function also has to distinguish cases when a neighbor of predicate $p$ is a part of the current assignment of the SAT solver. If not, and in the case it is also assigned to a concrete value in order to ensure $\mathcal{T}$-consistency, the $\mathcal{T}$-solver has to communicate the used value to the SAT solver.

Here, it is important how the notification is done. There are two possibilities. One is to use $\mathcal{T}$::LEARN, meaning that the $\mathcal{T}$-solver provides a full propositional explanation of the theory propagation to the SAT solver, which learns the fact and will determine the new value in SAT::PROPAGATE. The second option is just to include the new value into propagations $\boldsymbol{P}_{\mathcal{T}}{}^{*}$, without an explanation. However, this might require to use $\mathcal{T}$::EXPLAIN in the future. Both the possibilities are discussed further in a separate paragraph below.

Note that function $\mathcal{T}$::PROPAGATE traverses only via propagations $\boldsymbol{P}$. It may happen that a predicate from $\boldsymbol{P}$ cannot be checked for consistency nor theory-propagated at that moment, because some arguments are currently `undef`. It may seem that in this case the checks and propagations of the predicate will be missed in the future. However, such predicates will still be reached via PROPAGATE_INFERENCED, from the inference rules that must be used at any case to make the predicates evaluable. If the inference rules were not used, the result would be `unknown`.

**Decision Heuristics.** Now we go back to function $\mathcal{T}$::SUGGEST in function ONLINE_SAT (Algorithm 6.8). There is a bunch of possible strategies for the suggestions of decisions, depending on a specific problem and concrete encoding into a formula. In the case of formulas with an unrolling similar to BMC, each consecutive stage depends on the values from the previous one. Thus, a suitable strategy, called BMC strategy, is to first decide Booleans that correspond to the lower steps. Moreover, it is often useful to prefer deciding predicates that allow inference rules. For example, initial inference rules can always be used, or those rules that depend on values that are already evaluated. Such a predicate, after applying the inference rule, may then enable consistency checks and theory propagations. Therefore, within the same discrete stage, we prefer predicates that correspond to initial inference rules, then predicates that allow the other inference rules, then other predicates, and lastly pure Booleans.

We list also another possible strategy—to ultimately prefer the “most initial” inference rules. To achieve that, we apply a modified version of the Floyd–Warshall algorithm within the preprocessing stage, where we are interested in distances between predicates within the dependency graph. For example, with $x_0 = 0$, $x_1 = x_0 + y$ and $x_1 < 0$, the distance from $x_0 = 0$ to $x_1 < 0$ is 2. Then, floating-point variables are sorted s.t. the ones with a most distant predicate from a corresponding inference rule comes first. For example, $x_0$ comes before $x_1$. Following such an order of the variables, one of the predicates that contains the variable is decided to true. This way, it prefers predicates that allow inference rules over other predicates. Consequently, when the algorithm discovers a $\mathcal{T}$-inconsistency, the resulting conflict clause is short, because the decisions are being made not far away from initial conditions (i.e. initial inference rules). Also, the resulting order of decisions often corresponds to the actual order of stages of a BMC problem encoded into a formula, even though the information about stages is not used explicitly.

Note that the computation time of all the necessary distances in the dependency graph is not always negligible. Another drawback is that the strategy does not consider pure Boolean variables at all, which can be important decision variables of a specific problem. As a consequence, some predicates that could be propagated based on such Boolean decisions may be decided instead, because predicates have higher decision priority than pure Booleans in this strategy.

In our experiments, the performance of these heuristics differ depending on concrete problems, and sometimes the difference is negligible. These strategies are still quite general, not specific to a concrete problem. However, these strategies can serve as a base for specific strategies, where specific decisions may be preferred in the first place, and the general strategies may serve as a fallback strategy. Such a technique is used for example in the case of our railway scheduling model, which is presented in Chapter 8.

SAT solvers have their own decision heuristics too. We use it as the last fallback strategy when no suggestion is available. For example, in the case of the "initial" strategy, pure Booleans are never suggested at all. The heuristic of the SAT solver can be designed to for example prefer variables that frequently participate in Boolean conflicts. Such a strategy is beneficial, because the "problematic variables" are being resolved soon.

So far, we discussed only static strategies for decision suggestions, that is, strategies that are precomputed within the preprocessing stage and do not adapt to the current assignment of variables. It is expected that such sophisticated strategies can do even better. Moreover, strategies that embed machine learning techniques may improve the efficiency of the searching significantly (as suggested in Section 3.3.2).

**Theory Learning and Explaining.** In function $\mathcal{T}$::PROPAGATE (Algorithm 6.10), theory propagations and $\mathcal{T}$-inconsistencies may or may not use function $\mathcal{T}$::LEARN, which learns a full propositional explanation of a theory fact. If this is always used, then the SAT solver always knows all encountered theory consequences and has an entire control over the propagations and conflict reasoning. In function ANALYZE_CONFLICT in Algorithm 6.8, conflicts are analyzed and possibly a conflict clause that is as general as possible is searched. Furthermore, the SAT solver may look for a so-called backjump clause which allows to backjump to a decision level that is lower than just $d - 1$. During these operations, the SAT solver may ask for an explanation of a Boolean literal that participates in the conflict, based on a database of clauses which are either ground or have been learned. In this case, all such explanations are already available to the SAT solver.

In the case of $\mathcal{T}$-inconsistencies, we mentioned that is does not matter much whether

$\mathcal{T}$::LEARN is used or not. However, the number of theory propagations might be huge and the SAT solver might be flooded by too many propositional constraints that may not be essential. If $\mathcal{T}$::LEARN is never used, the communication of the propagated value is fast. However, later it may happen that in ANALYZE_CONFLICT, an explanation of a Boolean literal is not available, and an external explanation must be provided, resulting in a callback to $\mathcal{T}$::EXPLAIN. This way, learning of theory facts is done lazily, only in cases when they are really necessary in conflict reasoning.

On the other hand, sometimes the learned clauses can be useful within propositional propagations or indirectly in conflict reasoning. So in the case of theory propagations, there is always a trade-off between the benefit and the cost of learning such a clause. We use $\mathcal{T}$::LEARN only in cases when the resulting clause is small enough, for example, if the number of literals is at most two.

## 6.3 Input Language

We designed a core input language with a format that is derived from SMT-LIB (Section 5.3.1), extended with descriptions of constraints on systems of functions. Besides the core language, we propose optional preprocessor macros that help to parametrize formulas in place, within the core language.

### 6.3.1 Core Language

The syntax of our core language is similar to quantifier-free non-linear real arithmetic in SMT-LIB, extended of functional variables, functional operators and functional predicates. The language uses fully parenthesized prefix notation similar to Lisp. A specification of the core language is available online [38].

See an example of a possible input in Figure 6.1 to get an idea of what the language looks like. The code sample corresponds to an unrolling of Example 4.7. All constraints on particular systems of functions $\mathcal{F}_j$ must be enclosed in a special environment which we simply call a *flow*. It is the only place where functional operators and functional predicates are allowed, and it also isolates $\mathcal{F}_j$ from other systems $\mathcal{F}_k$, $k \neq j$. However, the flows do not restrict the appearance of Boolean and numerical constraints, in contrast to state-of-the-art approaches. We distinguish the definition and an instantiation of flows, corresponding respectively to commands `define_flow` and `flow`. In the example, there are two systems of functions $\mathcal{F}_1$ and $\mathcal{F}_2$, corresponding to two instantiations but just one definition. Although the systems are different, this means that their length may vary but yet they share the same pattern of the related constraints. This is typical for formulas that are based on an unrolling similar to BMC. The flows require the simple initial conditions of functional variables in the form of explicit arguments of the instantiations, including the initial value of time which here is not necessarily 0, in contrast with the original definition. Therefore, operator $\mathit{init}$ is often not necessary to be used. We also do not include deterministic final conditions which are currently redundant.

```
(declare-const up_1 Bool) (declare-const up_2 Bool) (declare-const up_3 Bool)
(declare-const t_1  Real) (declare-const t_2  Real) (declare-const t_3  Real)
(declare-const x_1  Real) (declare-const x_2  Real) (declare-const x_3  Real)
(declare-const v_1  Real) (declare-const v_2  Real) (declare-const v_3  Real)
(define-fun g () Real 9.81)
(define-fun K () Real 0.95) (define-fun D () Real 10)
;;; Initial conditions
(assert (and  (not up_1) (= t_1 0) (= x_1 5) (= v_1 0) ))
;;; Flows
(define-flow ball (x v) ((&up Bool) (&up* Bool)
                         (&t* Real) (&x* Real) (&v* Real))
(and
    ;; Functional constraints
    (= x' v) (>= x 0)
    (ite &up  (= v' (- (- g) (/ v D) ))
              (= v' (+ (- g) (/ v D) )) )
    (ite &up  (>= v 0) (<= v 0) )
    ;; Switching to the next stage
    (xor &up &up*)
    (ite &up (and (= &x* (final x)) (= &v* 0                    ) )
             (and (= &x* 0         ) (= &v* (* (- K) (final v) )) ))
    (= &t* (final _t))  ;; init(_t) != 0
))
(assert (and  (flow ball (t_1 x_1 v_1) (up_1 up_2 t_2 x_2 v_2) )
              (flow ball (t_2 x_2 v_2) (up_2 up_3 t_3 x_3 v_3) )
))
(assert (and  (<= t_3 2) (>= x_3 2) ))
;;; Execution and fetching the assignment
(check-sat) (get-model)
```

Figure 6.1: Bouncing ball encoded to the core language.

Command `define_flow` requires an identifier of the flow definition and three lists: a list of identifiers of functional variables, a list with additional parameters, and a formula. The parameters do not allow functional variables at all, which may require to use auxiliary numerical variables in order to share the final values of the functional variables with a different system of functions (i.e. the next stage). Using prefix `&` within the parameters is just a convention to better distinguish them from the functional variables, and suffix `*` denotes parameters that belong to the next stage. The convention has no semantical meaning. The formula of the flow contains functional and switching constraints. We denote the implicit functional variable that models time by `_t`.

Command `flow` requires an identifier of the corresponding flow definition and two lists: the initial values of time and the functional variables, and the parameters. Here all numerical variables that represent initial and also final values of functional variables must be explicitly declared. For this reason, although the input models 2 stages of the unrolling of the formula, there are also variables belonging to auxiliary stage 3 which is needed to store the final values of stage 2.

## 6.3.2 Preprocessor Macros

We also introduce rich preprocessing language, since it is often necessary to generate the input depending on a number of parameters in a generic way. This is useful especially in the case of formulas that are based on unrolling as in BMC. The preprocessor operates with *macros* which are used right within the input, similarly to the C preprocessor, and is usable for other SMT-LIB logics too. Our preprocessor is also partially inspired by the Lisp language, but it differs in many ways. A specification of the preprocessing language is available online [102].

The preprocessor reserves a couple of characters that have a special meaning. The characters have to be escaped if the special meaning is not desired, but they do not appear in most of the logics in SMT-LIB. The most important special character is, similarly to the C preprocessor, # which in most cases stands for macro expansion. Unlike in C, we always denote the expansions of macros explicitly.

```
#ifndef J
#define J 2
#endif

#for (j 1 $d(+ #J 1))
    (declare-const up_#j Bool) (declare-const t_#j  Real)
    (declare-const x_#j  Real) (declare-const v_#j  Real)
#endfor

#include ball/const.smto    ;; g = 9.81, K = 0.95, D = 10
#include ball/init.smto     ;; t_1 = 0, x_1 = 5, ...
#include ball/flow_def.smto

(assert (and
    #for (j 1 #J)
    #let k $d(+ #j 1)
        (flow ball (t_#j x_#j v_#j) (up_#j up_#k t_#k x_#k v_#k) )
    #endlet k
    #endfor
))

#include ball/goal.smto     ;; t_3 = 2, x_3 = 2

;;; Execution and fetching the assignment
(check-sat) (get-model)
```

Figure 6.2: A generic encoding of the bouncing ball example.

Figure 6.2 shows an example of a generic version of the input from Figure 6.1. Here the parameter of the input is `J`, which may or may not be defined before the execution (e.g. from command line). The macros of our preprocessor are quite more powerful than in the C preprocessor, because they support features like loops (`#for`), recursion, etc. Here, we introduce user macros by reserved macros `#define`, and later also `#let` which has a limited scope. Both loops contain a so-called arithmetic expansion which uses the special character `$`. It performs an immediate evaluation of the underlying expression. Suffix `d` denotes that we interpret the result as an integer.

In order to improve the readability of the input, we explicitly divide the parts of the formula that do not depend on parameters into separate files, which we include into the input using macro `#include`.

## 6.4 Implementation

We implement the algorithms presented in this chapter and the input language in C++ in our solver UN/SAT modulo ODEs Not SOT (UN/SOT) that is available online as open-source [72]. We used MiniSat2 (Section 5.2) as the underlying SAT solver, and implemented the lazy online approach to SMT with exhaustive theory propagation presented in Section 6.2.2. This required inserting several callbacks into MiniSat2, resulting in a fork of the original implementation [69]. Our current implementation of the ODE solver (Section 6.1) uses Odeint (Section 5.1) as the underlying ODE stepper. It provides the stepping algorithms in the form of templates in C++. Other solvers that offer a suitable interface that we mentioned in the section can be used too. In order to handle functional flows in an efficient way, we use the notion of so-called functional modes.

**Functional Modes.** In Section 4.1, we defined functional flows which only support conjunctions of functional constraints. However, disjunctions may appear in the formula as well, referring to different variants of ODEs and invariants, for instance in Example 4.7 (Section 4.2).

In order to model such disjunctions, one may cover all the combinations by enumerating all possible corresponding functional flows. Then, before each simulation, one could select an entire functional flow based on the selection of the variants using a supervising algorithm that handles Boolean constraints (such as a SAT solver)—similarly to the case of state-of-the-art solvers from Section 5.4.1. But this is not too convenient, because such flows can often share many functional constraints. We overcome such an issue by designing the interface of the solver such that it supports different variants of particular functional predicates.

In order to avoid confusion between the words variant and invariant, which have completely different meanings, we will prefer the word (functional) *mode* instead of variant. We use the word mode with a similar connotation as in other literature related to hybrid automata or hybrid systems, but we differ in the context. We demonstrate the difference on Examples 4.7 and 2.3. In Example 2.3, corresponding to the literature, each (discrete) mode can be modeled by a location of the hybrid automaton, resulting in two (global) modes that each describes all ODEs and invariants. In Formula 4.10 of Example 4.7, one can observe that all possible variants of particular ODEs are the following:

$$
\begin{aligned}
\dot{x}_j : \quad & (1) \quad \dot{x}_j = v_j \\
\dot{v}_j : \quad & (1) \quad \dot{v}_j = -g - \frac{v_j}{D} \\
& (2) \quad \dot{v}_j = -g + \frac{v_j}{D}.
\end{aligned}
\tag{6.4}
$$

Therefore, in the case of $\dot{x}_j$ (i.e. the derivative of functional variable $x_j$), there is just one possibility, and there are two possibilities in the case of $\dot{v}_j$. We call such particular possibilities *ODE modes* of functional variables. For example $\dot{v}_j = -g - \frac{v_j}{D}$ is an ODE mode of functional variable $v_j$. Accordingly, each such a mode corresponds to just one functional variable of the corresponding system of functions $\mathcal{F}_j$. Before each execution of the ODE solver, one must select exactly one ODE mode for each of the functional variables.

Moreover, we also distinguish whether modes are related to ODEs or to invariants. Referring again to Formula 4.10, we can enumerate all possible invariants as follows:

$$
\begin{aligned}
(1) \quad & x_j \geq 0 \\
(2) \quad & v_j \geq 0 \\
(3) \quad & v_j \leq 0
\end{aligned}
\tag{6.5}
$$

resulting in three possibilities in total. We call such possibilities *invariant modes*. Observe that these modes are *not* fixed to particular variables since in the case of invariants, according to Definition 12, the left hand side of the predicate does not have to be just a functional variable (see Example 4.3). Consequently, it is possible to select more invariants at once.

A *(functional) mode* is either an ODE mode or an invariant mode. We also allow to abbreviate an ODE mode of functional variable $f$ as a (functional) mode of $\dot{f}$. We will in addition assume that a mode is a functional mode where it is clear from context.

In the case of Example 4.7, we end up with one mode of $\dot{x}_j$, two modes of $\dot{v}_j$ and three invariant modes, resulting in $1 \cdot 2 \cdot 2^3 = 16$ combinations of all the modes. However, there are actually only two reachable combinations of all the modes wrt. the formula in the example, but the ODE solver does not have this information. This is a responsibility of the supervising algorithm (in our case, the SMT solver), that is, proper handling of the nondeterminism which is related to the choice of the combinations. The two reachable combinations correspond to the two global modes in Example 2.3.

Using functional modes is in the end similar to using functional flows (Definition 14) in the solver from Section 6.1, with the difference that the interface of the solver may require just indices of the modes instead of providing all functional constraints all the time. See the following example.

---

**Example 6.3.** We show the relationship between functional modes and functional flows. The modes just offer multiple possibilities which functional predicates to include into the flow, and which not. For instance, an illustrative visualization of a functional flow that is based on functional modes from Formula 6.4 and 6.5 and on initial conditions from Example 4.7 can look as follows:

$$\begin{gathered} init(x) = X_0 \\ init(v) = V_0 \\ \dot{x}_j = v_j \\ \left(\dot{v}_j = -g - \frac{v_j}{D}, \quad \dot{v}_j = -g + \frac{v_j}{D}\right) \\ \{x_j \geq 0, \quad v_j \geq 0, \quad v_j \leq 0\} \end{gathered} \tag{6.6}$$

where the list notation expresses one-of-many possibilities in the place of a functional ODE, resulting in ODE modes, and the set notation expresses multiple possibilities in the place of an invariant, resulting in invariant modes.

The result looks visually similar to Example 4.5. However, using something of the form as in Formula 6.6 allows to efficiently solve particular simulations in Example 4.7: before each such a simulation, the SMT solver is required to provide initial values for $x$ and $v$ and to select the appropriate functional modes.

---

A selected functional mode of a functional flow intuitively means that we already chose the mode (or more modes in the case of invariant modes) and did not include the remaining modes into the flow. This corresponds to the way how our $\mathcal{T}$-solver in Section 6.2.1 handles functional literals. Concrete representation of a structure such as Formula 6.6 depends on implementation. The selection or particular modes can be efficiently implemented e.g. using arrays and indeces of the modes.

Chapter 7

# Case Studies with ODEs

In this chapter, we study the behavior of our implementation of the solver presented in Chapter 6, UN/SOT, on selected benchmark problems and provide experimental results. All of the presented models are described using logical formulas of the theory of ODEs (Section 4.2) with an unrolling that is similar to the BMC problem (Section 2.5). All of the formulas contain differential equations. Recall that the solver checks strong satisfiability of formulas (Definition 20). Although the solver may return `unknown`, it is not the case for the benchmarks presented here.

Each experiment is parametrized by several options, but we show only some combinations. All input and output data is available online within the website of our tool[1]. Experiments were performed on a personal laptop machine with CPU Intel® i7-4702MQ, 8GB memory, running on OS Arch Linux with 5.8.14 Linux kernel.

We also briefly mention our railway scheduling problem in Section 7.1, which fits into this chapter, but the main discussion of the problem including the corresponding model and experimental results follow in a separate chapter—Chapter 8.

*Notation.* Unrolling ranges over discrete steps (i.e. stages) $j \in \{1, \ldots, J\}$, $J \in \mathbb{Z}^{>0}$. We denote by $\phi_j$ the particular parts of the formulas that belong to stage $j$ and differ from the other stages only in the index $j$.

In order to improve readability and make the presented formulas shorter, we exclude final conditions that are of the form $\mathit{init}(f_{j+1}) = \mathit{final}(f_j)$ for all $f_j \in \mathcal{F}_j$, and numerical predicates of the form $r^{[j+1]} = r^{[j]}$, for all $j \in \{1, \ldots, J-1\}$.

**Comparison with an Existing Solver.** In addition to the above, we are interested in how far our theoretical finding that using simulation semantics within SAT modulo ODE is easier than using classical mathematical semantics of ODEs holds in practice. We compare our solver with solver dReal3 (Section 5.4.1) and focus on the inherent practical difficulty of the respective problems. We did not use dReal4 since the case

[1] `https://gitlab.com/Tomaqa/unsot`, directory `doc/experiments`, subdirectory `v0.8` and `v0.7`.

studies are based on bounded model checking and the respective tool dReach is not available for that version.

We perform the comparison for two benchmarks (Sections 7.2 and 7.3) which originate from the database of benchmarks of dReal. The last experiment (Section 7.4) is excluded from the comparison (the explanation follows in the section).

The models that participate in the comparisons are usually not equivalent, mainly because dReal entirely explores all interval constraints, while we only approximate intervals using sample points, and because we check strong satisfiability. This is relevant for the benchmarks where the result is `unsat`.

Also, we do not guarantee that we used the tools dReach and dReal3 optimally.

## 7.1 Railway Scheduling

This experiment is highly inspired by [91], and relates to a planning problem where the task is to find a schedule of a number of trains, which are supposed to visit some nodes, with optional timing and ordering constraints, in a given railway network. We model the dynamics of trains based on differential equations, in contrary to [91].

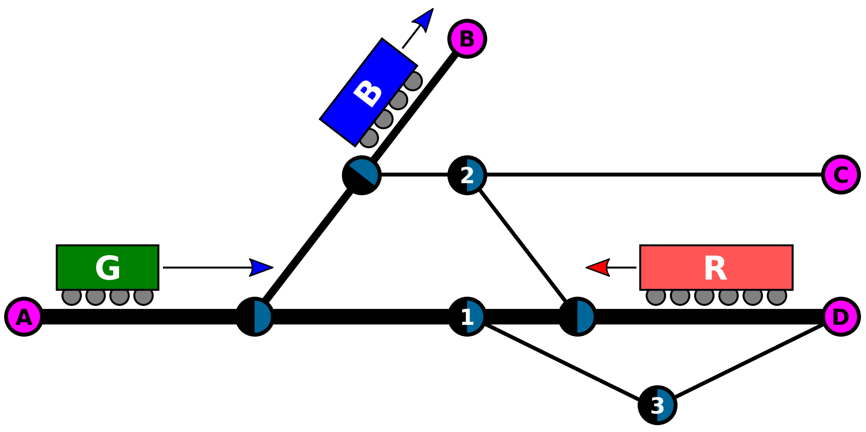


Figure 7.1: An example of a rail network graph with trains.

Figure 7.1 shows an example of a railway network that we model using a graph. Pink vertices denote boundary nodes of the network. The trains may have various lengths and properties of their dynamic behavior. Importantly, particular vertices of the graph restrict the velocity of the trains to a certain limit, and each train starts at a certain boundary and may visit given vertices.

An example of a trajectory of a single train in a trivial straight infrastructure, where it only drives through the whole network, is shown in Figure 7.2. The plot illustrates how the continuous variables that model the dynamics of the train progress in time. The important thing is that the train visits several edges that have different velocity limits. We call the edges segments. Variable $dx$ denotes a relative distance from the start of the current segment either with the back or the front of the train. Variable $v$ denotes

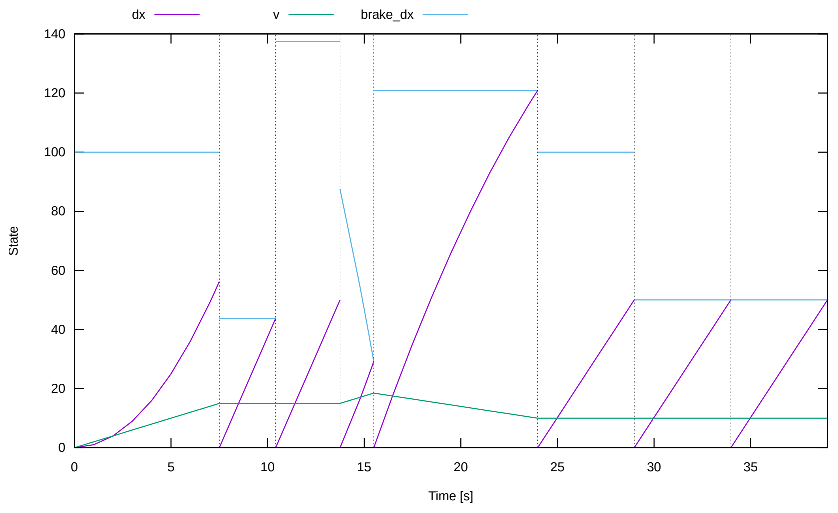


Figure 7.2: Trajectories of continuous variables of a single train on a simple track.

the current velocity of the train. The train always accelerates as long as possible and decelerates only when it has to, in order to obey the velocity limit of the next segment. Auxiliary variable $brake\_dx$ models an upper bound on the current distance that corresponds to a forced switch to a next stage, either because of reaching the end of the current segment, of because of the next velocity limit. Although the simulation of the benchmark is straightforward, the algorithm still needs to resolve some Boolean constraints that control the acceleration modes of the train, obeying the velocity limits, etc. The execution time of this trivial benchmark is approximately 0.02 s.

*Further discussion of the problem including the corresponding model and experimental results follow in Chapter 8.*

## 7.2 Hormone Therapy of Prostate Cancer

This experiment [87] studies a hormone therapy of prostate cancer in the form of androgen deprivation, or more concretely, intermittent androgen suppression (IAS). It switches between the modes on and off of the treatment while monitoring a tumor marker called prostate-specific antigen (PSA), which is described by the dynamics of mixed population of androgen-dependent (AD) and androgen-independent (AI) cancer cells. The task is to find a personalized treatment in the form of an efficient schedule of treatment modes that depends on the individual patient.

### 7.2.1 Specification

The model has only two Boolean modes: $\mathrm{on}$ and $\mathrm{off} \Leftrightarrow \neg\mathrm{on}$. The dynamics are described by three functions: $x$ and $y$ represent the population of AD and AI cells, respectively; and $z$ stands for androgen concentration. In addition, $v = x + y$ defines PSA level.

The cancer relapse is specified by the $v \geq 30$ property. The experiment should answer the question whether a relapse can be reached within $1000 = t_{\text{goal}}$ days under a chosen treatment schedule. A schedule is driven by two real parameters: $r_0 \in [0, 8)$ and $r_1 \in [8, 15]$.

## 7.2.2 Model

Particular formulas $\phi_j$ are defined s.t.

$$
\begin{aligned}
\dot{x}_j &= X_j \\
\land\ \dot{y}_j &= Y_j \\
\land\ \dot{v}_j &= V_j \\
\land\ \text{ITE}\Big(\text{on}^{[j]},\ \dot{z}_j &= -\frac{z_j}{T} + z_j \cdot c_3,\ \dot{z}_j = \frac{Z_0 - z_j}{T} + z_j \cdot c_3\Big) \\
\land\ X_j &= x_j \cdot \big(G_{x_j} - M_j + c_1\big) \\
\land\ Y_j &= x_j \cdot M_j + y_j \cdot \big(G_{y_j} + c_2\big) \\
\land\ V_j &= X_j + Y_j \\
\land\ G_{x_j} &= \alpha_x \cdot \left(k_1 + \frac{(1 - k_1) \cdot z_j}{z_j + k_2}\right) - \beta_x \cdot \left(k_3 + \frac{(1 - k_3) \cdot z_j}{z_j + k_4}\right) \\
\land\ G_{y_j} &= \alpha_y \cdot \left(1 - \frac{d_0 \cdot z_j}{Z_0}\right) - \beta_y \\
\land\ M_j &= m_1 \cdot \left(1 - \frac{z_j}{Z_0}\right) \\
\land\ x_j &\geq 0 \land y_j \geq 0 \land z_j \geq 0 \land v_j \in [0, 30] \\
\land\ t_j + t^{[j]} &\leq t_{\text{goal}} \land t^{[j+1]} = t^{[j]} + \mathit{final}(t_j) \\
\land\ \text{ITE}\Big(\text{on}^{[j]},\ v_j &> r_0 \lor V_j \geq 0,\ v_j < r_1 \lor V_j \leq 0\Big) \\
\land\ \text{ITE}\Big(\text{on}^{[j]},\ \big(\mathit{final}(v_j) &> r_0 \lor \mathit{final}(V_j) > 0\big) \Leftrightarrow \text{on}^{[j+1]}, \\
\big(\mathit{final}(v_j) &< r_1 \lor \mathit{final}(V_j) < 0\big) \Leftrightarrow \text{on}^{[j+1]}\Big)
\end{aligned}
$$

where $X_j$, $Y_j$, $V_j$, $G_{x_j}$, $G_{y_j}$ and $M_j$ are just helper functions that abbreviate the underlying expressions; $T$, $c_1$, $c_2$, $c_3$, $Z_0$, $\alpha_x$, $\beta_x$, $k_1$, $k_2$, $k_3$, $k_4$ and $d_0$ are predefined real constants and $\alpha_y$, $\beta_y$ and $m_1$ are constants that are specific for a given patient. For even more details, we refer readers to the original paper [87].

**Initial Conditions.** Fixed initial conditions are $\mathit{init}(x_1) = 15 \land \mathit{init}(y_1) = 0.1 \land \text{on}^{[1]}$, and $t^{[1]} = 0$. The patients are in addition parametrized by an initial condition on $\mathit{init}(z_1)$.

**Experimental Evaluation.** We examined two main scenarios for satisfiability:

- *safe*: $\bigwedge_{j=1}^{J} (\phi_j \ \wedge \ \mathit{final}(v_j) \leq 30)$
- *unsafe*: $\bigwedge_{j=1}^{J} \phi_j \ \wedge \bigvee_{j=1}^{J} \mathit{final}(v_j) > 30$

where also the final conditions are included. We distinguished a few different patients, represented by several constant values; and different values and ranges of $r_0$ and $r_1$ (either constant or interval).

Each scenario is parametrized by four values:

- $I \in \{\bot, \top\}$: whether $r_0$ and $r_1$ are intervals, or not (i.e. fixed to constants),
- $P \in \{2, 10, 97\}$: id. of the patient (see [87] for concrete values of the related constants),
- $S \in \{50, 1\}$: all derivatives are multiplied by this scale and $t_{\text{goal}} = 1000/S$,

Our model (not of dReal) is, in addition, parametrized by:

- $s_r \in \{0.5, 0.2\}$: an equidistance with which the intervals of $r_0$ and $r_1$ are sampled (if $I$ is true),
- $dt = \frac{5 \cdot 10^{-2}}{\sqrt{S}}$: fixed size of integration steps.

**Results.** We selected only some combinations of parameters, especially excluding the ones which were computationally too expensive.

Each patient was verified in 4 variants: *unsafe* or *safe* scenario and constant ($\neg I$) or interval ($I$) ranges of $(r_0, r_1)$. All variants are grouped together, separately for each patient.

Table 7.1: Hormone therapy verification of patient #10.

(a) *safe*, $\neg I$.

| $S$ | $N$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 50 | 6 | sat | 0.4 s | 16 s |
| 50 | 8 | sat | 0.4 s | 1 min |
| 50 | 10 | sat | 0.7 s | 4.5 min |
| 1 | 6 | sat | 2.1 s | 16 s |
| 1 | 8 | sat | 2.4 s | 1 min |
| 1 | 10 | sat | 4.1 s | 4.5 min |

(b) *unsafe*, $\neg I$.

| $S$ | $N$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 50 | 6 | unsat | 0.3 s | 8 s |
| 50 | 8 | unsat | 0.4 s | 38 s |
| 50 | 10 | unsat | 0.7 s | 2.5 min |
| 1 | 6 | unsat | 2 s | 8 s |
| 1 | 8 | unsat | 2.4 s | 33 s |
| 1 | 10 | unsat | 4.1 s | 2 min |

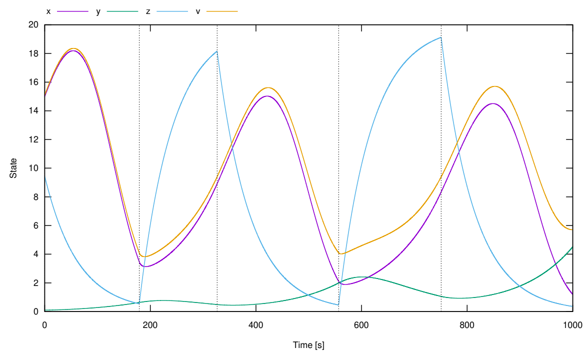
(c) *safe*, $I$, $S = 50$.

| $s_r$ | $N$ | Result | $r_0$ | $r_1$ | Time | (dReal) |
|---|---|---|---|---|---|---|
| 0.5 | 4 | sat | 4.5 | 10.5 | 1 s | 2 h |
| 0.5 | 6 | sat | 4 | 13 | 2.8 s | > 10 h |

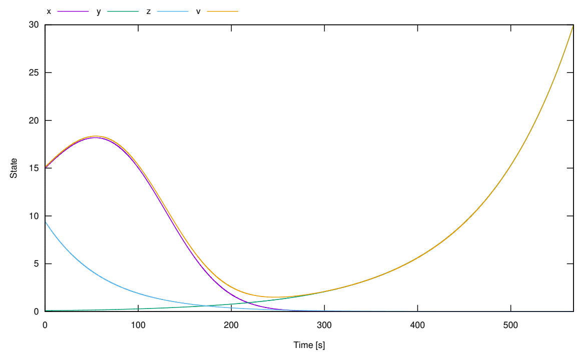
(d) *unsafe*, $I$, $S = 50$.

| $s_r$ | $N$ | Result | $r_0$ | $r_1$ | Time | (dReal) |
|---|---|---|---|---|---|---|
| 0.5 | 4 | sat | 0 | 14 | 0.5 s | > 10 h |
| 0.5 | 6 | sat | 0 | 8 | 1.1 s | × |

Results of patient #10 are shown in Table 7.1. In case of $\neg I$ ($r_0 = 4.1$, $r_1 = 9.4$), the patient is verified to be treated safely (*unsafe* case is unsatisfiable and the opposite for *safe* case). In case of $I$, *unsafe* scenario became satisfiable too, as shown in Figure 7.3. dReal is not that sensitive to $S$ parameter like our solver, but our approach is still much



(a) *safe*, $\neg I$, $S = 1$, $J = 6$.



(b) *unsafe*, $I$, $S = 1$, $J = 4$.

Figure 7.3: Trajectories of patient #10's hormone therapy.

more efficient, especially with growing $J$[2]. In the case of intervals, the difference between computation times starts to be huge. In case of *safe*, similar values of $(r_0, r_1)$ are found compared to the predefined ones. In case of *unsafe*, our solver found dangerous combinations of $(r_0, r_1)$ also quickly.

Table 7.2: Hormone therapy verification of patient #2.

(a) *safe*, $\neg I$.

| $S$ | $N$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 50 | 6 | `unsat` | 0.1 s | 5 s |
| 50 | 10 | `unsat` | 0.2 s | 1.5 min |
| 1 | 6 | `unsat` | 0.4 s | 5 s |
| 1 | 10 | `unsat` | 0.7 s | 1.5 min |

(b) *unsafe*, $\neg I$.

| $S$ | $N$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 50 | 6 | `sat` | 0.1 s | 3 s |
| 50 | 10 | `sat` | 0.2 s | 1 min |
| 1 | 6 | `sat` | 0.4 s | 3 s |
| 1 | 10 | `sat` | 0.7 s | 1 min |

(c) *safe*, $I$, $S = 50$.

| $s_r$ | $N$ | Result | $r_0$ | $r_1$ | Time | (dReal) |
|---|---|---|---|---|---|---|
| 0.5 | 6 | `unsat` | × | × | 5 s | 6 s |
| 0.5 | 10 | `unsat` | × | × | 10 s | 1 min |
| 0.2 | 6 | `unsat` | × | × | 0.5 min | 6 s |
| 0.2 | 10 | `unsat` | × | × | 1 min | 1.5 min |

(d) *unsafe*, $I$, $S = 50$.

| $s_r$ | $N$ | Result | $r_0$ | $r_1$ | Time | (dReal) |
|---|---|---|---|---|---|---|
| 0.5 | 6 | `sat` | 0 | 8 | 0.1 s | 4 s |
| 0.5 | 10 | `sat` | 2 | 10 | 0.3 s | 1 min |
| 0.2 | 6 | `sat` | 2.2 | 10 | 0.4 s | 4 s |
| 0.2 | 10 | `sat` | 2.4 | 9.6 | 4.4 s | 1 min |

Next patient, #2, follows in Table 7.2. The main difference against patient #10 is that here the patient is concluded to be untreatable: even with intervals, there was no schedule which would avoid *unsafe* state. In the case of $I$, dReal performs much better than in the case of patient #10. We also tried to increase the granularity of our interval sampling approximation ($s_r = 0.2$), where, with smaller $J$, dReal performed similarly or even better than our solver.

Last but not least patient, presented in Table 7.3, is #97. This patient, in case of $\neg I$, has the schedule predefined to $r_0 = 4$ and $r_1 = 10$, which, however, appeared to be

[2]One of reasons why dReach tool is sensitive to $J$ is that it enumerates all discrete paths (except obviously unreachable ones) and then checks each separately (until the goal is reached). Also, we modeled an absorbing discrete mode (not in our case as it is not necessary), which is visited when $t > t_{\text{goal}}$; and a higher amount of reachable modes increases the complexity.

Table 7.3: Hormone therapy verification of patient #97.

(a) *safe*, $\neg I$.

| $S$ | $N$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 50 | 6 | `unsat` | 0.2 s | 12 s |
| 50 | 10 | `unsat` | 0.6 s | 4 min |

(b) *unsafe*, $\neg I$.

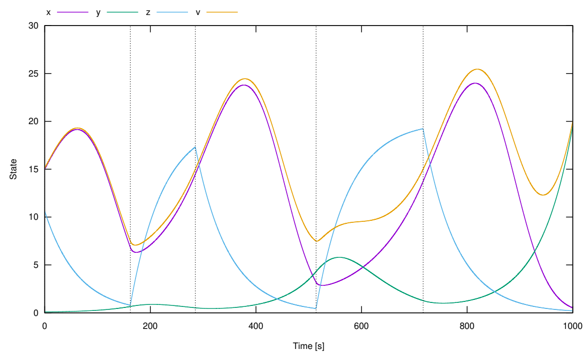
| $S$ | $N$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 50 | 6 | `sat` | 0.2 s | 7.5 s |
| 50 | 10 | `sat` | 0.6 s | 2 min |

(c) *safe*, $I$, $S = 50$.

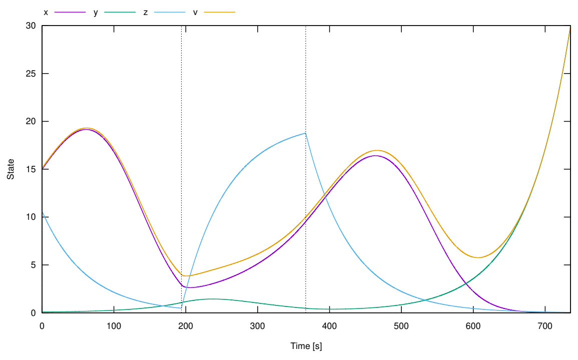
| $s_r$ | $N$ | Result | $r_0$ | $r_1$ | Time | (dReal) |
|---|---|---|---|---|---|---|
| 0.5 | 4 | `sat` | 7.5 | 15 | 5 s | 1.33 h |
| 0.5 | 6 | `sat` | 7.5 | 15 | 15 s | > 10 h |

(d) *unsafe*, $I$, $S = 50$.

| $s_r$ | $N$ | Result | $r_0$ | $r_1$ | Time | (dReal) |
|---|---|---|---|---|---|---|
| 0.5 | 4 | `sat` | 0 | 14 | 0.4 s | > 10 h |
| 0.5 | 6 | `sat` | 0 | 8 | 0.8 s | × |



(a) *safe*, $I$, $S = 1$, $J = 6$.



(b) *unsafe*, $\neg I$, $S = 1$, $J = 4$.

Figure 7.4: Trajectories of patient #97's hormone therapy.

unsafe. Still, an appropriate schedule was found with intervals, as shown in Figure 7.4.

We showed all possible cases within the model: focusing on the cases with intervals, which are more general, patient #2 was proved to be untreatable, while for patients #10 and #97 a suitable treatment schedule was found. On the other hand, an unsafe schedule is possible to find for all patients.

## 7.3 Glucose Control

Methods for inpatient glycemic control began to be important since diabetes associated complications among hospitalized patients are increasing [31]. As glucose physiology is significantly impacted by patient-specific parameters, it is critical to verify that a glycemic controller is safe, i.e. it does not drive the glucose level into dangerous low (hypoglycemia) or high (hyperglycemia) values. Formal verification of such controllers can provide a new level of safety guarantee to clinicians before performing human clinical tests, which can be invasive and costly.

Following [31], the target scenario is controlling a T1D (Type 1 Diabetes) patient in two phases: before and during a surgery process. In the first phase, the patient is being monitored (typically for 30 minutes) to ensure the patient is stable enough for surgery. If so, the surgery and the second phase follow, where the PD (proportional-derivative) controller starts operating (it drives insulin and glucose inputs), wrt. observed condition of the patient, which is periodically sampled approximately every 30 minutes. The safety property holds if, for all initial conditions, surgery starts and the glucose level stays in a certain set of safe states. The unsafety property holds if surgery starts and the set of safe states is left.

### 7.3.1 Specification

The model presented in the paper contains accurate physiological dynamics and a validated intraoperative glycemic control protocol. The dynamics can be divided into two subsystems: insulin and glucose. The insulin system is represented by 5 functions: $I_p$ and $I_l$ as insulin mass in the plasma and liver, respectively; $I_1$ and $I_d$ as a delayed insulin transportation process; and $X$ as an insulin signal that adjusts glucose concentration. The glucose system is defined by 2 functions: $G_t$ and $G_p$, which represent the glucose concentration in interstitial fluids and plasma, respectively. The insulin system is driven by the insulin input $u$ and the glucose system by the intravenous glucose input $m$. Boundary glucose levels are observed via function $y = G_p/V_g$.

The PD controller periodically updates $u^{[j]}$ and $m^{[j]}$ values based on two glucose readings: the current one ($y_j$) and the previous one ($y_{j-1}$).

The discrete space of the model is represented by 11 modes $\{\mathrm{mode1}, \dots, \mathrm{mode11}\}$, where $\mathrm{mode1}$ is an auxiliary initial mode; $\mathrm{mode2}$ and $\mathrm{mode3}$ are *absorbing* modes which are visited whenever a permitted range of the glucose level is left—$\mathrm{mode2}$ in case of the first phase (then the surgery is canceled or delayed, and the process ends), and $\mathrm{mode3}$ in case of the second phase (since the surgery is already in progress, it is considered as the *unsafe* state); and the rest of the modes depend on the PD controller, while $\{\mathrm{mode4}, \dots, \mathrm{mode7}\}$ modes model the first (monitoring) phase and $\{\mathrm{mode8}, \dots, \mathrm{mode11}\}$ the second (surgery) phase. This could definitely be modeled more practically, but I decided to follow the original description.

All periods are timed by function $f_\tau$ (even the first monitoring phase). In order to model the practical scenario that a clinician may not perform the check exactly on time, the invariants allow timing nondeterminism with a sampling jitter $\delta$.

### 7.3.2 Model

The partial formulas $\phi_j$ of the model, where some unimportant properties related to modes $\{\text{mode1}, \text{mode2}, \text{mode3}\}$ are omitted for simplicity, are:

$$
\begin{aligned}
\dot{I}_{p_j} &= -I_{p_j} \cdot (m_2 + m_4) + I_{l_j} \cdot m_1 + \frac{u^{[j]} \cdot 10^2}{BW} \\
\wedge\ \dot{X}_j &= \left( \frac{I_{p_j}}{V_I} - X_j - I_b \right) \cdot P_{2u} \\
\wedge\ \dot{I}_{1_j} &= \left( \frac{I_{p_j}}{V_I} - I_{1_j} \right) \cdot k_I \\
\wedge\ \dot{I}_{d_j} &= \left( I_{1_j} - I_{d_j} \right) \cdot k_I \\
\wedge\ \dot{I}_{l_j} &= I_{p_j} \cdot m_2 - I_{l_j} \cdot (m_1 + m_3) \\
\wedge\ \dot{G}_{t_j} &= -G_{t_j} \cdot \left( \frac{X_j \cdot V_{m_x} + V_{m_0}}{G_{t_j} + K_{m_0}} + k_2 \right) + G_{p_j} \cdot k_1 \\
\wedge\ \dot{G}_{p_j} &= G_{t_j} \cdot k_2 - G_{p_j} \cdot k_1 + \frac{m^{[j]} \cdot 10^3}{BW} - F_{snc} + h_j \\
\wedge\ \dot{f}_{\tau_j} &= 1 \\
\wedge\ \big((\text{mode4}^{[j]} \vee \text{mode8}^{[j]}) \Rightarrow h_j = 0\big) &\wedge \big((\text{mode5}^{[j]} \vee \text{mode9}^{[j]}) \Rightarrow h_j = C_{1_j}\big) \\
\wedge\ \big((\text{mode6}^{[j]} \vee \text{mode10}^{[j]}) \Rightarrow h_j = C_{1_j} + C_{2_j}\big) &\wedge \big((\text{mode7}^{[j]} \vee \text{mode11}^{[j]}) \Rightarrow h_j = C_{2_j}\big) \\
\wedge\ C_{1_j} &= -I_{d_j} \cdot k_{p3} - G_{p_j} \cdot k_{p2} + k_{p1} \\
\wedge\ C_{2_j} &= \left( G_{p_j} - k_{e2} \right) \cdot k_{e1} \\
\wedge\ \big((\text{mode4}^{[j]} \vee \text{mode5}^{[j]} \vee \text{mode6}^{[j]} \vee \text{mode7}^{[j]}) &\Rightarrow (y_j \in [70, 130] \wedge f_{\tau_j} \leq 29)\big) \\
\wedge\ \big((\text{mode8}^{[j]} \vee \text{mode9}^{[j]} \vee \text{mode10}^{[j]} \vee \text{mode11}^{[j]}) &\Rightarrow (y_j \in [60, 180] \wedge f_{\tau_j} \leq 30 + \delta^{[j]} \wedge \delta^{[j]} \in (-1, 1])\big) \\
\wedge\ \big((\text{mode4}^{[j]} \vee \text{mode8}^{[j]}) &\Rightarrow (C_{1_j} \leq 0 \wedge C_{2_j} \leq 0)\big) \\
\wedge\ \big((\text{mode5}^{[j]} \vee \text{mode9}^{[j]}) &\Rightarrow (C_{1_j} \geq 0 \wedge C_{2_j} \leq 0)\big) \\
\wedge\ \big((\text{mode6}^{[j]} \vee \text{mode10}^{[j]}) &\Rightarrow (C_{1_j} \geq 0 \wedge C_{2_j} \geq 0)\big) \\
\wedge\ \big((\text{mode7}^{[j]} \vee \text{mode11}^{[j]}) &\Rightarrow (C_{1_j} \leq 0 \wedge C_{2_j} \geq 0)\big) \\
&\dots
\end{aligned}
$$

$$
\begin{aligned}
&\wedge \Bigg(\text{mode1}^{[j]} \Rightarrow \Bigg(\text{ITE}\Big(\mathit{final}(C_{2_j}) \leq 0,\ \text{ITE}\big(\mathit{final}(C_{1_j}) \leq 0,\ \text{mode4}^{[j+1]},\ \text{mode5}^{[j+1]}\big),\\
&\qquad \text{ITE}\big(\mathit{final}(C_{1_j}) \leq 0,\ \text{mode7}^{[j+1]},\ \text{mode6}^{[j+1]}\big)\Big)\\
&\qquad \wedge\ \mathit{init}(f_{\tau_{j+1}}) = 0 \wedge u^{[j+1]} = 0 \wedge m^{[j+1]} = 0\Bigg)\Bigg)\\
&\wedge (\text{mode2}^{[j]} \Rightarrow \text{mode2}^{[j+1]}) \wedge (\text{mode3}^{[j]} \Rightarrow \text{mode3}^{[j+1]})\\
&\wedge \Bigg(\big(\text{mode4}^{[j]} \vee \text{mode5}^{[j]} \vee \text{mode6}^{[j]} \vee \text{mode7}^{[j]}\big) \Rightarrow \text{ITE}\Bigg(\mathit{final}(y_j) \notin [70,130],\ \text{mode2}^{[j+1]},\\
&\qquad \text{ITE}\Bigg(\mathit{final}(f_{\tau_j}) \leq 29,\\
&\qquad\quad \big((\mathit{final}(C_{1_j}) \leq 0 \wedge \mathit{final}(C_{2_j}) \leq 0) \Rightarrow \text{mode4}^{[j+1]}\big)\\
&\qquad\quad \wedge\big((\mathit{final}(C_{1_j}) \geq 0 \wedge \mathit{final}(C_{2_j}) \leq 0) \Rightarrow \text{mode5}^{[j+1]}\big)\\
&\qquad\quad \wedge\big((\mathit{final}(C_{1_j}) \geq 0 \wedge \mathit{final}(C_{2_j}) \geq 0) \Rightarrow \text{mode6}^{[j+1]}\big)\\
&\qquad\quad \wedge\big((\mathit{final}(C_{1_j}) \leq 0 \wedge \mathit{final}(C_{2_j}) \geq 0) \Rightarrow \text{mode7}^{[j+1]}\big),\\
&\qquad\quad \big((\mathit{final}(C_{1_j}) \leq 0 \wedge \mathit{final}(C_{2_j}) \leq 0) \Rightarrow \text{mode8}^{[j+1]}\big)\\
&\qquad\quad \wedge\big((\mathit{final}(C_{1_j}) \geq 0 \wedge \mathit{final}(C_{2_j}) \leq 0) \Rightarrow \text{mode9}^{[j+1]}\big)\\
&\qquad\quad \wedge\big((\mathit{final}(C_{1_j}) \geq 0 \wedge \mathit{final}(C_{2_j}) \geq 0) \Rightarrow \text{mode10}^{[j+1]}\big)\\
&\qquad\quad \wedge\big((\mathit{final}(C_{1_j}) \leq 0 \wedge \mathit{final}(C_{2_j}) \geq 0) \Rightarrow \text{mode11}^{[j+1]}\big)\Bigg)\Bigg)\Bigg)\\
&\wedge \Bigg(\big(\text{mode8}^{[j]} \vee \text{mode9}^{[j]} \vee \text{mode10}^{[j]} \vee \text{mode11}^{[j]}\big) \Rightarrow \text{ITE}\Bigg(\mathit{final}(y_j) \notin [60,180],\ \text{mode3}^{[j+1]},\\
&\qquad \text{ITE}\Bigg(\mathit{final}(f_{\tau_j}) \leq 30 + \delta^{[j]},\\
&\qquad\quad \big((\mathit{final}(C_{1_j}) \leq 0 \wedge \mathit{final}(C_{2_j}) \leq 0) \Rightarrow \text{mode8}^{[j+1]}\big)\\
&\qquad\quad \wedge\big((\mathit{final}(C_{1_j}) \geq 0 \wedge \mathit{final}(C_{2_j}) \leq 0) \Rightarrow \text{mode9}^{[j+1]}\big)\\
&\qquad\quad \wedge\big((\mathit{final}(C_{1_j}) \geq 0 \wedge \mathit{final}(C_{2_j}) \geq 0) \Rightarrow \text{mode10}^{[j+1]}\big)\\
&\qquad\quad \wedge\big((\mathit{final}(C_{1_j}) \leq 0 \wedge \mathit{final}(C_{2_j}) \geq 0) \Rightarrow \text{mode11}^{[j+1]}\big),\\
&\qquad\quad (\text{mode8}^{[j]} \Rightarrow \text{mode8}^{[j+1]}) \wedge (\text{mode9}^{[j]} \Rightarrow \text{mode9}^{[j+1]})\\
&\qquad\quad \wedge(\text{mode10}^{[j]} \Rightarrow \text{mode10}^{[j+1]}) \wedge (\text{mode11}^{[j]} \Rightarrow \text{mode11}^{[j+1]})\\
&\qquad\quad \wedge \mathit{init}(f_{\tau_{j+1}}) = 0\\
&\qquad\quad \wedge \text{ITE}\Big(\mathit{final}(y_j) < 100 \wedge \mathit{final}(y_j) - \mathit{final}(y_{j-1}) < -30,\\
&\qquad\quad u^{[j+1]} = 0 \wedge m^{[j+1]} = -0.1 \cdot (\mathit{final}(y_j) - \mathit{final}(y_{j-1})),\\
&u^{[j+1]} = \max\big(0, 1 + 0.05 \cdot (\mathit{final}(y_j) - 100) + 0.06 \cdot (\mathit{final}(y_j) - \mathit{final}(y_{j-1}))\big) \wedge m^{[j+1]} = 0\Big)\Bigg)\Bigg)\Bigg)
\end{aligned}
$$

where $h_j$, $C_{1_j}$ and $C_{2_j}$ are helper functions that only serve to abbreviate the formula. When operator *final* is applied to one of these functions, it means that the operator is applied to all functional variables occurring in the expression. And, again, all other variable symbols that were not mentioned yet are numerical constants—for more details here, we refer to the original paper [31].

**Initial Conditions.** Initial conditions are defined by $init(f_{\tau_1}) = 0 \wedge u^{[1]} = 0 \wedge m^{[1]} = 0 \wedge y_1 = 100 \wedge \mathrm{mode1}^{[1]}$.

**Experimental Evaluation.** We observed two main scenarios:

- *safe*: $\bigwedge_{j=1}^{J} \phi_j \ \wedge \ (\mathrm{mode8}^{[J]} \vee \mathrm{mode9}^{[J]} \vee \mathrm{mode10}^{[J]} \vee \mathrm{mode11}^{[J]})$
- *unsafe*: $\bigwedge_{j=1}^{J} \phi_j \ \wedge \ \mathrm{mode3}^{[J]}$

where $J$ is the number of discrete steps. In the *safe* scenario, we are interested only in modes of the second (surgery) phase. Also, there was a significant difference between the cases where we fixed parameters and/or initial values of the insulin and glucose system, and cases where we allowed intervals of these values. Again, in the case of intervals, we only cover the intervals by a finite number of sample points within our solver, and for dReal, we used the original intervals. The same applies also to modeling the timing jitter.

The presented model slightly *differs* from the original specification in [31], because we fixed a few mistakes:

- the model did not allow switching between all the modes 8–11 (and 4–7) at all, because the jump conditions were not compatible with invariants—there was no overlap due to using the strict inequalities that are used in jump conditions from $\mathrm{mode1}$ (where they do not cause any harm)[3]; this could produce incorrect `unsat` result,
- it did not allow transitions within the absorbing modes into themselves, which, in the case of BMC with a fixed number of unrollments, could also result in incorrect `unsat` result.

Both main scenarios, *safe* and *unsafe*, were additionally parametrized by:

- $I_F \in \{-1, 0, 1, 2\}$: the level of interval variance of initial values of the glucose and insulin system (where $I_F = 0$ stands for fixed values which lead to a safe region, and specially $I_F = -1$ stands for fixed values which lead to the fail mode),

[3]Speaking of the precise mathematical semantics, it is also necessary to avoid oscillating between neighboring modes, using some threshold only after which the transition back to the previous mode is enabled. This, however, is not necessary in the case of our solver, thanks to the discretization and to the way how invariants are treated.

- $I_P = 0$: the level of interval variance of all 18 constant parameters, which are, however, always fixed here (other values are possible: $I_P \in \{0, 1, 2\}$), because neither of the solvers was capable of solving such difficult tasks.

Our model (not of dReal) is also parametrized by:

- $s_d \in \{1, 0.5, 0.25\}$: an equidistance with which the intervals of $\delta^{[j]}$ are sampled,
- $S_F = \min(1, I_F \cdot 5)$: the number of sample points for each function variable according to $I_F$,
- $S_P = \min(1, I_P \cdot 5)$: the number of sample points for each constant according to $I_P$,
- $dt = 2.5 \cdot 10^{-2}$: fixed size of integration steps.

The tool dReach was run from a helper script `dReach.sh` due to a discovered bug in dReach[4].

**Results.** Regardless of $I_F$ or $I_P$, there is always some nondeterminism due to the timing jitter $\delta^{[j]}$. However, the cases with all other initial values being fixed were still easy for our solver. Allowing the intervals turned out to be difficult, but often not impossible, especially in satisfactory cases.

Table 7.4: Glucose control verification with fixed values ($I_F \leq 0$).

(a) $I_F = 0$, *safe*.

| $N$ | $s_d$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 6 | 1 | `sat` | 1.5 s | 14 h |
| 6 | 0.5 | `sat` | 1.2 s | 14 h |
| 6 | 0.25 | `sat` | 3.3 s | 14 h |
| 9 | 1 | `sat` | 3.6 s | × |
| 9 | 0.5 | `sat` | 3.3 s | × |
| 9 | 0.25 | `sat` | 4 s | × |
| 12 | 1 | `sat` | 6 s | × |
| 12 | 0.5 | `sat` | 7 s | × |
| 12 | 0.25 | `sat` | 14 s | × |

(b) $I_F = 0$, *unsafe*.

| $N$ | $s_d$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 4 | 1 | `unsat` | 0.5 s | 2 min |
| 4 | 0.5 | `unsat` | 1.5 s | 2 min |
| 4 | 0.25 | `unsat` | 5.5 s | 2 min |
| 5 | 1 | `unsat` | 2 s | 0.5 h |
| 5 | 0.5 | `unsat` | 7 s | 0.5 h |
| 5 | 0.25 | `unsat` | 0.75 min | 0.5 h |
| 6 | 1 | `unsat` | 5.5 s | 14 h |
| 6 | 0.5 | `unsat` | 0.75 min | 14 h |
| 6 | 0.25 | `unsat` | 7.75 min | 14 h |

(c) $I_F = -1$, *safe*.

| $N$ | $s_d$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 6 | 1 | `unsat` | 1.1 s | 30 h |
| 6 | 0.5 | `unsat` | 2.8 s | 30 h |
| 6 | 0.25 | `unsat` | 12 s | 30 h |
| 9 | 1 | `unsat` | 1.5 s | × |
| 9 | 0.5 | `unsat` | 3.7 s | × |
| 9 | 0.25 | `unsat` | 15 s | × |
| 12 | 1 | `unsat` | 1.5 s | × |
| 12 | 0.5 | `unsat` | 4 s | × |
| 12 | 0.25 | `unsat` | 16.5 s | × |

(d) $I_F = -1$, *unsafe*.

| $N$ | $s_d$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 4 | 1 | `unsat` | 0.5 s | 1.75 min |
| 4 | 0.5 | `unsat` | 1.5 s | 1.75 min |
| 4 | 0.25 | `unsat` | 4 s | 1.75 min |
| 5 | 1 | `sat` | 1 s | 6.75 min |
| 5 | 0.5 | `sat` | 1 s | 6.75 min |
| 5 | 0.25 | `sat` | 2 s | 6.75 min |
| 6 | 1 | `sat` | 1 s | 1.5 h |
| 6 | 0.5 | `sat` | 1.3 s | 1.5 h |
| 6 | 0.25 | `sat` | 2 s | 1.5 h |

[4]According to my experience and observations, when the number of modes of a model in dReach exceeds 10, the order of translated mode variables becomes inconsistent with the original ones (maybe due to alphabetical sorting instead of numerical) in the area of goal constraints, and has to be corrected.

The simplest case is with no intervals ($I_F \in \{0, -1\}$), with initial values that should lead either to a safe or an unsafe region. The results of both *safe* and *unsafe* scenarios are shown together in Table 7.4. In the case of $I_F = -1$, the unsafe state is reached after 5 steps.

Table 7.5: Glucose control verification with some interval values ($I_F = 1$).

(a) *safe*.

| $N$ | $s_d$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 3 | 1 | `sat` | 0.2 s | 6.5 h |
| 3 | 0.5 | `sat` | 0.5 min | 6.5 h |
| 3 | 0.25 | `sat` | 0.3 s | 6.5 h |
| 6 | 1 | `sat` | 0.5 min | × |
| 6 | 0.5 | `sat` | 2.7 s | × |
| 6 | 0.25 | `sat` | 3.9 s | × |
| 9 | 1 | `sat` | 4 s | × |
| 9 | 0.5 | `sat` | 4 s | × |
| 9 | 0.25 | `sat` | 8.5 s | × |

(b) *unsafe*.

| $N$ | $s_d$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 3 | 1 | `unsat` | 1.25 h | 6 s |
| 3 | 0.5 | `unsat` | 2.25 h | 6 s |
| 4 | 1 | `unsat` | 7 h | 2 min |
| 4 | 0.5 | `unsat` | $>$ 10 h | 2 min |
| 5 | 1 | `unsat` | $>$ 10 h | 0.66 h |
| 5 | 0.5 | `unsat` | × | 0.66 h |

The results of the next case, with some intervals ($I_F = 1$), are shown in Table 7.5. One can see that the satisfiable cases still work well in our solver, while the unsatisfiable ones do not, because the number of combinations that needs to be checked is already enormous. Solver dReal, on the other hand, is quite efficient in these cases.

Table 7.6: Glucose control verification with large intervals ($I_F = 2$).

(a) *safe*.

| $N$ | $s_d$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 3 | 1 | `sat` | 1 s | $>$ 23 h |
| 3 | 0.5 | `sat` | 9 s | $>$ 23 h |
| 3 | 0.25 | `sat` | 0.5 s | $>$ 23 h |
| 6 | 1 | `sat` | 1 min | × |
| 6 | 0.5 | `sat` | 2 min | × |
| 6 | 0.25 | `sat` | 1.5 min | × |
| 9 | 1 | `sat` | 1.75 min | × |
| 9 | 0.5 | `sat` | 0.75 min | × |
| 9 | 0.25 | `sat` | 0.5 min | × |

(b) *unsafe*.

| $N$ | $s_d$ | Result | Time | (dReal) |
|---|---|---|---|---|
| 3 | 1 | `unsat` | overflow | 9 s |
| 4 | 1 | ? | $>$ 10 h | $>$ 15 h |
| 5 | 1 | `sat` | 0.75 h | $>$ 24 h |

The last presented case is with large intervals ($I_F = 2$). This differs from the previous case because now the unsafe state can be reached within the possible initial values. In other cases, the results are relatively similar to the previous ones—see Table 7.6. The "overflow" result denotes that our solver crashed.

Generally, our solver performed much better in `sat` cases, including the cases with intervals. However, in `unsat` cases, and especially with intervals, the performance of our tool degrades heavily, when choosing more sample points in the intervals. In the worst case, it has to check the finite set of all floating-point numbers in the intervals, while dReal uses more sophisticated techniques. We should fix the behavior for these instances.

The original paper of the experiment also used dReal for the verification, but we achieved different results. The authors did not discuss any `sat` case, where they could

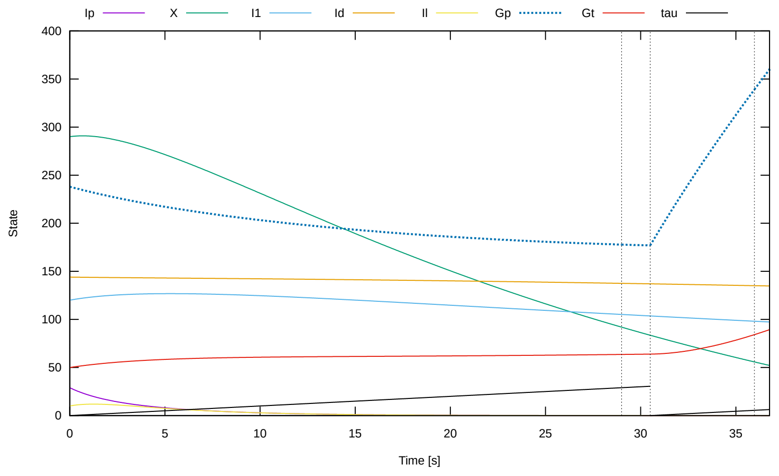


Figure 7.5: Witness of the reachability of the unsafe state in the glucose model.

probably notice unintended `unsat` results. In the end, they considered their model to be safe, as reaching an unsafe state was always `unsat`. However, the corrected model can reach an unsafe state—the witness, presented in Table 7.4, has this initial state: $\mathit{init}(I_p^{[1]}) = 29 \wedge \mathit{init}(X^{[1]}) = 290 \wedge \mathit{init}(I_1^{[1]}) = 120 \wedge \mathit{init}(I_d^{[1]}) = 144 \wedge \mathit{init}(I_l^{[1]}) = 10 \wedge \mathit{init}(G_p^{[1]}) = 238 \wedge \mathit{init}(G_t^{[1]}) = 50$. The trajectory is shown in Figure 7.5, where dangerous level of $y$, which depends on $G_p$ (the dotted line), was reached.

## 7.4 Racing Car Simulation

This experiment shows a simplified model of a car that races a square-intersection track. The car has no controller—its discrete behavior is (almost) completely nondeterministic. The tool dReach is unusable for this kind of task[5], and was not tested with this model at all.

The task is to verify whether such a model of a car can elapse the track in a limited time.

### 7.4.1 Specification

The car is represented as a small square, with two coordinates $x$ and $y$, and by its velocity $v$ and angle $\omega$. Its discrete state space is defined by variables $\mathrm{acc}$ (acceleration or deceleration), $\mathrm{brake}$ (braking), $\mathrm{rapid}$ (rapid action), $\mathrm{left}$ (steer left) and $\mathrm{right}$ (steer right). For example, the car does not alter the current velocity iff $\mathrm{acc}$ is false, and it brakes rapidly iff $\mathrm{acc} \wedge \mathrm{rapid} \wedge \mathrm{brake}$ holds. The model is partially time-triggered, that is, a blind discrete decision is being made every $T$ seconds (except that steering and brak-

[5]As was mentioned in Section 7.2, dReach enumerates all possible combinations of discrete modes, which results in exponential growth, even before the verification itself starts.

ing are forbidden while standing still), as long as no collision with the track boundaries happens. The time when the car finishes is constrained by a constant: $t_{\text{finish}} \le t_{\text{goal}}$.

The square track is defined by width $W$ and length $L$. Its outer ($B_l$ and $B_r$) and inner ($b_l$ and $b_r$) x-boundaries (left and right) are derived from $W$ and $L$: $B_l = 0$, $b_l = B_l + W$, $b_r = b_l + L$ and $B_r = b_r + W$; y-boundaries are the same. The y-coordinate of the horizontal start line is $y_{\text{start}} = B_l + W + L/2$ (see the plots of what the track looks like).

In the measured experiments, track sizes are fixed to $W = 2 \wedge L = 10$, and consecutively to these, maximum time is fixed to $t_{\max} = 20$. The number of steps of the unrolling $J$ is given by $J = \frac{t_{\max}}{T}$, and $t_{\text{goal}}$ is set to $c_{t_{\text{goal}}} \cdot t_{\max}$. Other options are parametrized:

- $T \in \{1, 0.625, 0.5\}$,
- $c_{t_{\text{goal}}} \in \{1, 0.8, 0.6\}$,
- integration step size is $dt \in \{0.05, 0.01\}$.

## 7.4.2 Model

The partial formulas $\phi_j$ are defined as follows:

$$\begin{aligned}
\dot{x}_j = v_j \cdot \cos(\omega_j) \wedge \dot{y}_j = v_j \cdot \sin(\omega_j)\\
\wedge (\neg \text{acc}^{[j]} \Rightarrow \dot{v}_j = 0)\\
\wedge (\text{acc}^{[j]} \wedge \neg \text{brake}^{[j]} \wedge \neg \text{rapid}^{[j]} \Rightarrow \dot{v}_j = \frac{g}{2})\\
\wedge (\text{acc}^{[j]} \wedge \neg \text{brake}^{[j]} \wedge \text{rapid}^{[j]} \Rightarrow \dot{v}_j = g)\\
\wedge (\text{acc}^{[j]} \wedge \text{brake}^{[j]} \wedge \neg \text{rapid}^{[j]} \Rightarrow \dot{v}_j = -g)\\
\wedge (\text{acc}^{[j]} \wedge \text{brake}^{[j]} \wedge \text{rapid}^{[j]} \Rightarrow \dot{v}_j = -2g)\\
\wedge (\neg \text{left}^{[j]} \wedge \neg \text{right}^{[j]} \Rightarrow \dot{\omega}_j = 0)\\
\wedge (\text{left}^{[j]} \Rightarrow \dot{\omega}_j = \frac{\pi}{2} \cdot c_\omega) \wedge (\text{right}^{[j]} \Rightarrow \dot{\omega}_j = -\frac{\pi}{2} \cdot c_\omega)\\
\wedge \text{ITE}\Big(v_j \le \theta,\ c_\omega = 1,\ c_\omega = \frac{2\theta}{v_j + \theta}\Big)\\
\wedge v_j \in [0, 60] \wedge t_j \le T\\
\wedge \mathit{final}(t_j) > T\\
\wedge x_j - \frac{s}{2} \ge B_l \wedge x_j + \frac{s}{2} \le B_r \wedge y_j - \frac{s}{2} \ge B_l \wedge y_j + \frac{s}{2} \le B_r\\
\wedge \Big(x_j - \frac{s}{2} \ge b_r \curlyvee x_j + \frac{s}{2} \le b_l \curlyvee y_j - \frac{s}{2} \ge b_r \curlyvee y_j + \frac{s}{2} \le b_l\Big)\\
\dots
\end{aligned}$$

$$\begin{aligned}
&\wedge \Big(init(v_j) = 0 \Rightarrow \big(\neg\text{brake}^{[j]} \wedge \neg\text{left}^{[j]} \wedge \neg\text{right}^{[j]}\big)\Big) \wedge \neg(\text{left}^{[j]} \wedge \text{right}^{[j]})\\
&\qquad \wedge t^{[j]} = T \cdot (j-1)\\
&\wedge \Big(\big(\text{middle}^{[j]} \wedge \neg\text{middle}^{[j-1]}\big) \Rightarrow t_{\text{middle}} = t^{[j]}\Big)\\
&\wedge \Big(\big(t^{[j]} > t_{\text{middle}} \wedge \text{finish}^{[j]} \wedge \neg\text{finish}^{[j-1]}\big) \Rightarrow t_{\text{finish}} = t^{[j]}\Big)\\
&\wedge \Big(\text{middle}^{[j]} \Leftrightarrow \big(\mathit{final}(x_j) > b_r \wedge \mathit{final}(y_j) < y_{\text{start}}\big)\Big)\\
&\wedge \Big(\text{finish}^{[j]} \Leftrightarrow \big(\mathit{final}(x_j) < b_l \wedge \mathit{final}(y_j) > y_{\text{start}}\big)\Big)
\end{aligned}$$

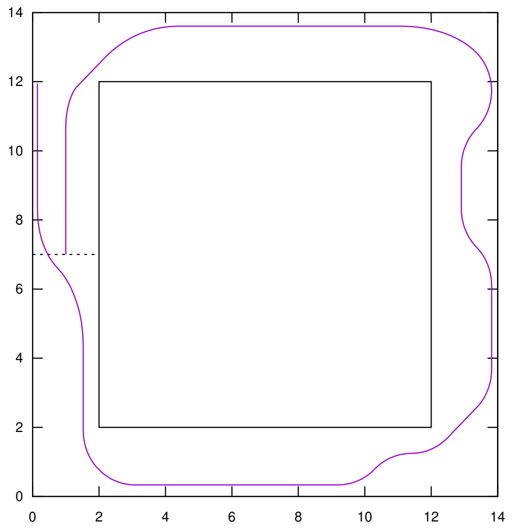
where $g$ is the gravitational constant; $\theta = g$ is steering threshold which approximates momentum effect; and $s$ is the diameter of the car square. Notably, the formula contains symbols $\curlyvee$ within invariants which denote disjunctions within the invariants. Although the original definition of invariants forbids Boolean combinations, is it not a problem in our case where invariants are based on simulations. The only thing is to clearly distinguish disjunctions that denote disjunctions within invariants, and disjunctions of separate invariants. Here, the disjunctions inside the invariant are necessary in order to model that the car must remain within the borders of the track.

Except of partial formulas $\phi_j$, there are additional (global) constraints:

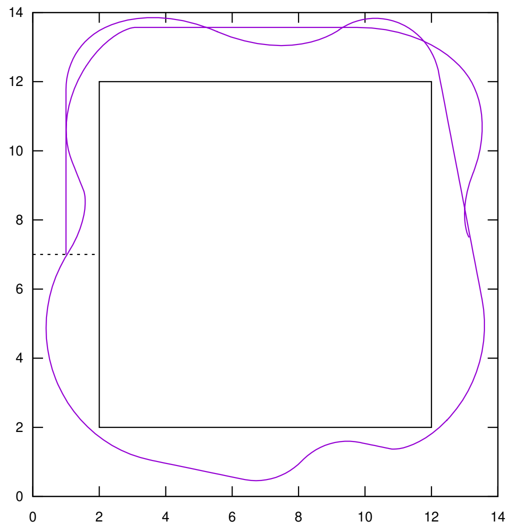
$$\begin{aligned}
&init(x_1) = B_l + W/2 \wedge init(y_1) = y_{\text{start}} \wedge init(v_1) = 0 \wedge init(\omega_1) = \frac{\pi}{2}\\
&\wedge \neg\text{middle}^{[1]} \wedge \neg\text{finish}^{[1]}\\
&\wedge \bigvee_{j=2}^{J} \text{middle}^{[j]} \wedge \bigvee_{j=2}^{J} \big(\text{finish}^{[j]} \wedge t^{[j]} > t_{\text{middle}}\big)\\
&\wedge t_{\text{finish}} \le t_{\text{goal}} \wedge \mathit{final}(v_J) < 0.
\end{aligned}$$

Table 7.7: Overview of the results of the racing car experiment.

| $T$ | $dt$ | $c_{t_{goal}}$ | Result | $t_{middle}$ | $t_{finish}$ | Time |
|---|---|---|---|---|---|---|
| 1 | 0.05 | 1 | `unsat` | × | × | 4.2 s |
| 1 | 0.01 | 1 | `unsat` | × | × | 5.2 s |
| 0.625 | 0.05 | 1 | `sat` | 14.375 | 19.375 | 2 min |
| 0.625 | 0.05 | 0.8 | `sat` | 8.75 | 15.625 | 9.5 min |
| 0.625 | 0.05 | 0.6 | × | × | × | > 1 h |
| 0.625 | 0.01 | 1 | `sat` | 9.375 | 16.875 | 1.5 min |
| 0.625 | 0.01 | 0.8 | `sat` | 11.25 | 15.625 | 14.5 min |
| 0.5 | 0.05 | 1 | `sat` | 8.5 | 15.5 | 2 min |
| 0.5 | 0.05 | 0.8 | `sat` | 9 | 16 | 0.75 min |
| 0.5 | 0.05 | 0.6 | `sat` | 8.5 | 12 | 6.25 min |
| 0.5 | 0.01 | 1 | `sat` | 14.5 | 20 | 19.5 min |
| 0.5 | 0.01 | 0.8 | × | × | × | > 1 h |



(a) $T = 0.5$, $dt = 0.05$, $c_{t_{\text{goal}}} = 0.8$.



(b) $T = 0.625$, $dt = 0.05$, $c_{t_{\text{goal}}} = 0.6$.

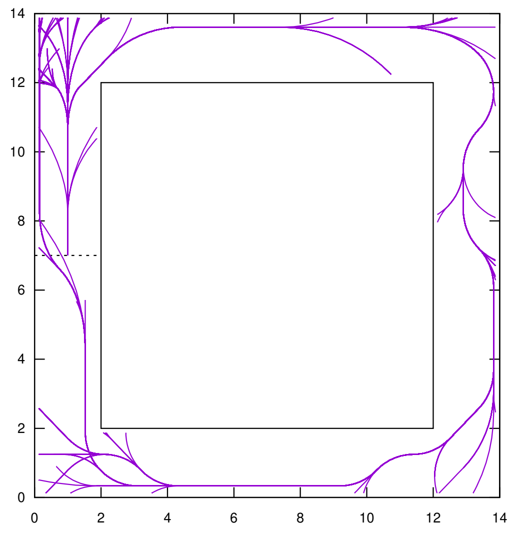
Figure 7.6: Selected trajectories of the racing car.

**Results.** Table 7.7 gathers results of all mentioned parameters. In case of $T = 1$, regardless of $c_{t_{\text{goal}}}$, all results are `unsat`, and even in suprisingly short time[6], but this is just caused by the low frequency of decisions (i.e. $\frac{1}{T}$), s.t. the car cannot make it through the very first curve (as shown in the plot Figure 7.7). For some parameters, the computation exceeded one hour. In all other presented cases, trajectories that satisfy the time constraint were found. Some cases with a tighter constraint were even faster than the loose ones, just because they were "more fortunate"—for satisfiable cases, there can be many nondeterministic ways how to reach a solution. In the end, the searched state space is quite huge, with the size somewhere below $15^{40}$ in case of $T = 0.5$. Still, the performance is not ideal here with the lazy offline approach.

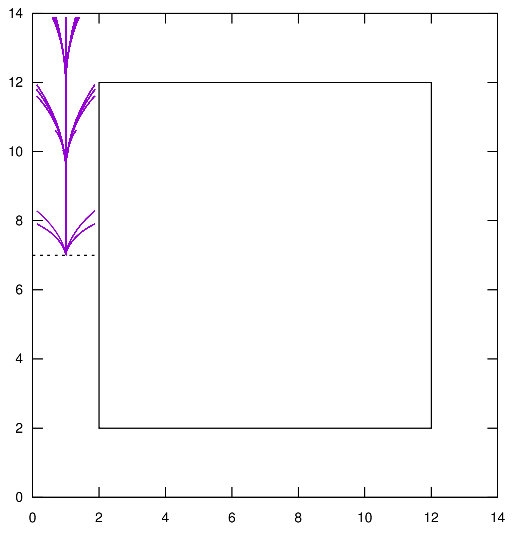
Figure 7.6 illustrates some resulting trajectories, which lack progress in time, but at least show that the car elapsed the race without collision. An animation would be much more descriptive. We also attach Figure 7.7 with all tried paths for some parameters.

We expect many opportunities in future work here, for example in expanding the model with a sophisticated controller s.t. the car drives more or less autonomously, and does not decide blindly. Then, it will be quite challenging to put such car into a much more complex environment, than this simple track.

[6]Cases when the car can approach the finish line, but not in time, are omitted here, as their computation times are expected to be very high.



(a) $T = 0.5$, $dt = 0.05$, $c_{t_{\mathrm{goal}}} = 0.8$ (`sat`).



(b) $T = 1$, $dt = 0.05$, $c_{t_{\mathrm{goal}}} = 1$ (`unsat`).

Figure 7.7: All tried paths wrt. the selected parameters of the racing car experiment.

## 7.5 Discussion of the Results

To summarize the results of presented experiments, and to offer an interpretation, the major observation is that the complexity of our method is much lower than in the case of SAT modulo ODE solver with classical mathematical semantics, especially in the satisfiable cases. In the unsatisfiable cases with intervals, however, the performance of our tool is still quite poor. The number of checked possibilities grows significantly with the number of sample points, and each case is computed separately from the others. Also, we currently cannot prove the unsatisfiability wrt. interval constraints.

We also state that the execution time of our tool is usually much more predictable than in the case of dReal, where the computation time of apparently simple instances, especially the satisfiable ones, is sometimes huge, while in our case, it scales in a predictable way wrt. the size of the formula and the number of sample points in intervals. A possible explanation is that methods based on interval computation have to fight with the so-called dependency problem that tends to blow up intervals over long time horizons.

Despite the different semantics, both solvers always came up with the same result of the satisfiability (with the exception when a solver did not finish the computation wrt. given time limit).

Finally, we would like to mention that we found dReal, including its benchmark database, immensely useful for developing, tuning, and debugging our own tool.

CHAPTER **8**

# Railway Scheduling

This chapter integrates an extended version of our paper [73] into the dissertation thesis.

We model the problem of railway scheduling that exhibits both non-trivial discrete and continuous behavior, using the definitions in Chapter 4, and we solve the problem using the algorithm from Chapter 6. The resulting solver is competitive with methods based on dedicated railway simulators while being more general and extensible.

## 8.1 Introduction

Existing benchmark problems for SAT modulo ODE [47, 58] do not exhibit complex discrete state space. We develop a benchmark problem that combines a non-trivial propositional part with differential equations. Moreover, we apply a corresponding algorithm (Chapter 6) that tightly integrates SAT and numeric simulations of differential equations. The resulting tool is available online [72].

Scheduling is a native satisfiability problem, where a plan that meets all criteria is sought. It starts to be challenging when plans should also cover complex dynamic phenomena, usually in the form of differential equations. Such dynamic models, typically of cyber-physical systems, can be simulated using numerical solvers, such as Xcos, Simulink, or SpaceEx [53]. However, automatic verification tasks, or planning tasks, are difficult with such complex models, especially if discrete state space is huge and complicated, which is typical for SAT. In the area of railway transport, precise scheduling can become to play an important role within upcoming autonomous traffic control systems.

The benchmark problem comes from the domain of railway scheduling, and is inspired by an approach to railway design capacity analysis [91], that combines a SAT solver with a railway simulator. The authors of that approach, referring to SAT modulo non-linear real arithmetic, "found these solvers insufficiently scalable for real-world problem sizes". Our experiments show that it indeed *is* possible to realistically handle continuous dynamics in the railway domain directly by SAT modulo theory solvers.

A major difficulty lies in modeling the fact that trains sometimes have to switch to a deceleration phase to obey velocity limits. Here, it is non-trivial to predict when such a switch must happen when modeling dynamics based on differential equations.

We show that it is possible to solve specific tasks efficiently even with a general purpose algorithm. The only parts that are dedicated to the particular railway scheduling problem are an appropriate decision heuristic, and the way how a formula is formed.

**Related Work.** We are not aware of any other approach to railway scheduling based on SAT modulo theories with realistic modeling of continuous dynamics. The mentioned approach [91] solves the problem of design capacity analysis, a different, but related problem. The main differences are:

- Instead of an ad-hoc combination of SAT and a simulator, we model the problem in a precisely defined Satisfiability Modulo Theories language (Section 4.2). As a result, numeric (e.g. timing) constraints can appear throughout a formula. They are analyzed in tight integration of the simulator and the SAT solver.
- Our model allows rich timing constraints. Boolean combinations of both relative and absolute timing constraints, with upper or lower bounds, are possible. Consequently, trains are allowed to keep waiting in stations, or before entering the network, even in cases when their routes do not collide with the other trains. Hence, our model may exhibit more nondeterminism which makes the scheduling problem more difficult.
- The dynamics of trains is an integral, but modifiable part of the model, instead of being hidden in a simulator.

Both approaches have different strong and weak aspects of the run-time performance.

As in the case of any formal model of real-world problems, also here, we abstract from certain aspects of the problem domain. Our model does not take into account railway policies or such, and our approach may be more generic than it is actually necessary in practice. For example, we do not model objects like signaling principles, train detectors, switches, and the like, as Luteberget et. al. [91] do. Especially we do not claim ETCS (European Train Control System) compatibility of our model, meaning that it may be less suitable for railway systems based on signal interlocking. However, not all railways use such a mechanism, for example urban railways may leave the responsibility to the driver. More specific comparison of particular differences follows in related sections.

Railway route planning can also be viewed as a multi-agent path finding problem [107], where trains are viewed as agents. However, in this area, usually much simpler models of continuous behavior are used [4]. On the other hand, the resulting plans are often minimized wrt. a given parameter, for example, sum of lengths of the agents' paths, while we do not optimize at all.

Of course, many other approaches dedicated to railway scheduling exist. Some support only limited precision, or work only under certain assumptions, for example, fixed routes, or not taking into account limited track capacity. Some use networks that were transformed from a microscopic level[1] to an aggregated, macroscopic level [108]. Some use an approximation where both microscopic and macroscopic models are included [2]. Also, probabilistic methods exist [109, 62].

There are approaches [121, 61] that are quite accurate, but still ignore some constraints that we take into account, for example:

- Not all combinations of possible train paths are considered [121].
- They are based on an already existing time table, which can even be assumed to be fixed [61]. No such prerequisites are necessary in our case.
- They still over-approximate the available capacity [61], by replacing bi-directional tracks by pairs of one-directional tracks. In our case, the topology of an infrastructure can correspond to the reality accurately.
- Simpler train dynamics is used [61].

On the other hand, most of the approaches that were mentioned so far are optimization algorithms, while we present a decision procedure.

The chapter is structured as follows. We start with an explanation of the problem area in Section 8.2. In Section 8.3 we present an encoding of the problem as a formula of the theory from Section 4.2. Finally, in Section 8.4 we analyze the behavior of our approach and of [91] on selected case studies.

## 8.2 Problem Overview

The problem is related to finding a low-level schedule within a railway network. In other words, it is *searching* for a simulation of trains that meets given properties. The real-world objects, like trains and railroads, are being abstracted in the form of mathematical models.

This section describes the overall problem and introduces related keywords. We start with an illustrative example.

### 8.2.1 Example

In Figure 8.1, one can see a model of a rail network with three trains. We distinguish the model itself and the required constraints on trains.

[1]Microscopic level corresponds to railway simulation, like in our case.

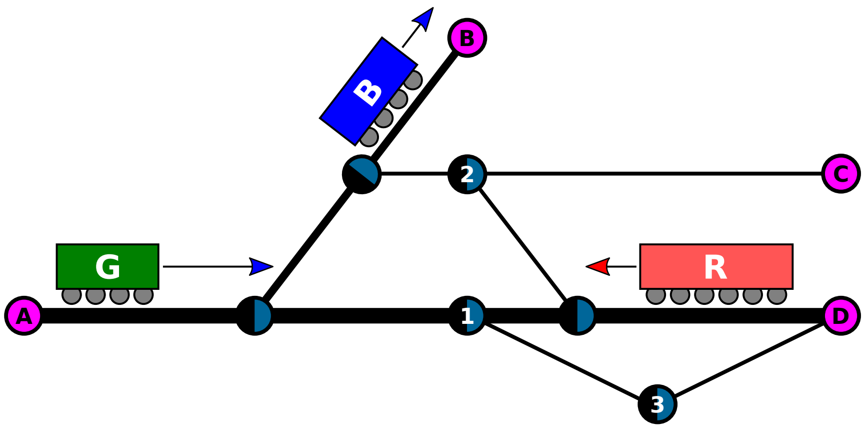


Figure 8.1: An example of a rail network graph with trains.

**Model.** The model consists of a graph of the network, and of abstracted trains. Each train is described by its physical properties, for example length, velocity limit, etc. The red train is a freight train, longer and slower than the other, passenger trains. For illustrative reasons, the boundary nodes of the graph are distinguished from the others. The thicker an edge is, the faster railroad it represents. Nodes that model stations are labeled with a number. To support modeling of railway junctions, nodes of the graph have two sides, illustrated by black and blue colors in the figure. In order to avoid physically impossible (e.g. too sharp) turns, a train has to visit both sides when transferring via such a double-sided node.

**Constraints.** Examples of constraints that the trains in the figure might be required to satisfy are:

- The blue train must start from the boundary A, and has no further requirements on visiting nodes.
- The green train must start from A, and is in addition required to visit node 3, where it will stop. Eventually, the train must continue to node D afterwards.
- The red train must start at D and exit at A, with no other required visits.
- Possible orderings of the trains: the blue train must start before the green train; the red train starts before the green train approaches node 1.
- Possible timings of the trains: the red train must arrive at A within 10 min after entering; the green train must wait at node 3 for at least 2 min.

**Result.** The result of the search is a plan that demonstrates how the trains can move through the network while satisfying the given constraints and with no collisions of trains.

An example of such a plan as a whole, satisfying the constraints listed above, follows. We refer to Figure 8.1 as a snapshot of the plan. The blue train entered first and is currently approaching boundary B, not interfering with the other two trains. The green train is free to enter next, but the red train must enter before the green approaches node 1. Although the red train is actually allowed to enter even before the green train, we assume the case where the green train enters first.

So, the green train aims to node 3, and plans to exit the network at boundary D afterward. However, as soon as the red train enters, the collision of the trains must be avoided. Either the green train must leave node 1 before it is approached by the ren train, or the red has to bypass node 1 via node 2. Then, the red train aims to boundary A.

This way, all the mentioned constraints were met, and the resulting schedule is also efficient wrt. the railway capacity and to real-time.

### 8.2.2 General Problem Statement

The task is to find a plan for a given set of trains and a railway network (viewed as a graph) such that all specified places are visited, meeting all timing and ordering constraints, and with no collisions of trains. It is a satisfiability problem—it decides whether there exists an assignment of variables that satisfies a formula. If it exists, the assignment is supplied, optionally including the trajectories of trains.

We assume that each train can only enter the network at a boundary, and that at the beginning of the whole search, there are no trains present in the network. Also, trains are not allowed to reverse their direction. All these aspects can be included in the model, though.

The decisions to be made for each train are branching edges, when to enter the network, and if it stopped at a station, when to exit it; everything else is, ideally, deterministic.

### 8.2.3 Railway Model

A closer specification of what we support in the model, for example, which properties of trains, or which kind of graphs, follows.

The environment of the presented experiments is a low-level model of railway transport. It consists of a steady infrastructure, that can actually relate to real world railroads, and of a given number of various train models with realistic dynamic behavior.

**Infrastructure.** An infrastructure (or a network) is modeled using a graph of vertices called *nodes* and edges called *segments*. Each segment has a length and a velocity limit. Properties like the segment's slope, angle or cant, or whether it is a tunnel, are missing. Only a single train is allowed inside a segment and the chosen next segment where the train currently aims to. A node that is not boundary either may or may not allow stopping, where nodes that allow stopping model stations. The fact that a train shall stop at a node is not modeled explicitly, but by temporarily setting the velocity limit of the train's chosen next segment (which the other trains are not allowed to enter) to zero. After stopping, trains may wait in stations for a limited time, or may not.

As explained in the example (Section 8.2.1), the graph is a *double-vertex* graph [94], which is commonly used for modeling railways with junctions [108].

We assume that each segment is at least as long as the longest train (Figure 8.1 violates this property). As a result, each train is always present within at most two segments on its way. It should not be difficult to improve the encoding s.t. this restriction is not necessary, though. The model directly supports infrastructures with cycles and looping of trains, in contrast with [91] where this needs an extra effort.

**Train.** A train $T$ has an acceleration and a deceleration rate, a velocity limit, and a length. The dynamics of trains is deterministic—each train drives at the maximum possible speed, which, however, depends on discrete decisions—the choice of segments on the train's way, and where to stop. Such a model already allows meaningful experiments, but can be easily extended by features like weight, number of wagons, etc.

### 8.2.4 Constraints

**Connection Constraints.** A *connection* is a mapping of a train to a non-empty list of nodes that must be visited in the given order. For instance, $T_{green} \mapsto (A, 3)$ is the connection of the green train from the example. The user must specify exactly one connection for each train. The list can contain boundary nodes too, but only as the first or the last element. The first element of the list indeed must be a boundary node. Trains always stop at the listed nodes that model stations, and never stop at any other stations. Other attitudes can be considered too (e.g., trains do stop there too, or they *may* stop there). For example, regarding the connection $green \mapsto (A, 3)$, the green train will stop at node 3, but will not stop at node 1, because it is not listed, and even if the list contained the node between nodes A and 1, the train will not stop there neither, because it does not model a station.

The *starting* node is the first node in the list. A connection may have several *ending* nodes—any boundary node terminating a path following the given connection. For example, in Figure 8.1, given a connection list $(A, 2)$, $A$ is the starting node and $C, D$ are two possible ending nodes, but for connection list $(A, 2, D)$, $D$ is the only ending node. We call segments incident with the starting node *starting segments*, and segments incident with an ending node *ending segments*.

**Schedule Constraints.** Schedule constraints are optional constraints that compare the time when a train either arrives at or departs from a node[2]. A departure is when a train

[2]For a required visit, it can be useful to also support specifying sets of nodes, instead of single nodes (meaning "any of the nodes"), which is possible to encode into the formula, but we do not support such a rule at the moment.

starts accelerating to leave a node after it stopped there earlier, or when entering the network.

In the following, we will denote by $\mathit{arrival}(T, N)$ (or $\mathit{departure}(T, N)$) the time when train $T$ arrives at (or departs from) node $N$. To allow both variants in a formula, we will write $\mathit{visit}(T, N)$, possibly distinguishing several occurrences by indices ($\mathit{visit}_1(T_1, N_1)$, $\mathit{visit}_2(T_2, N_2)$, etc.). Schedule constraints assume that all mentioned visits are the consequence of some connection constraint.

We allow two types of schedule constraints, ordering, and timing constraints. An *ordering* enforces two visits to happen in a given order. It has the form

$$\mathit{visit}_1(T_1, N_1) \bowtie \mathit{visit}_2(T_2, N_2), \tag{8.1}$$

where $\bowtie$ is one of $\{<, \leq, >, \geq, =\}$. In the case of [91], the only supported orderings are with $\bowtie$ being $<$ or $>$.

A *timing* is either absolute, concerning one visit:

$$\mathit{visit}(T, N) \bowtie \xi, \tag{8.2}$$

or relative, requiring a time constraint on a transfer, that is, on the time from one visit to another. It has the form

$$\mathit{transfer}(\mathit{visit}_1(T_1, N_1), \mathit{visit}_2(T_2, N_2)) \bowtie \xi, \tag{8.3}$$

where $\mathit{transfer}(v_1, v_2) := v_2 - v_1$, $\bowtie \in \{<, \leq, >, \geq\}$, and $\xi \in \mathbb{Q}^{\geq 0}$. Also, $\mathit{wait}(T, N) := \mathit{transfer}(\mathit{arrival}(T, N), \mathit{departure}(T, N))$. In the case of [91], the only supported timing constraints are $\mathit{transfer}(\mathit{arrival}(T_1, N_1), \mathit{arrival}(T_2, N_2)) < \xi$, which is quite sufficient for railway capacity verification, though.

## 8.3 Encoding and Formalization

In this section, we present an encoding of the planning problem from Section 8.2 as a formula in the theory described in Section 4.2. All of the presented formulas are generated automatically, from user input in the form of a preprocessing language (Section 6.3). The user input consists of a specification of an infrastructure and of trains, and of connections and schedule constraints.

We unroll the planning problem in a similar way as in BMC (Section 2.5). Unrolling ranges over discrete steps $0, 1, \dots, J$. A variable $x$ specific to a discrete step $j$ has the form $x^{[j]}$. All trains are modeled *synchronously*, meaning that every discrete step $j$ corresponds to the same global moment in time. Functional variables specific to one and the same discrete step will have the same length $\tau^{[j]}$ of integration (which corresponds to $\tau_j$ in Section 4.1), from which we get global time by defining real variables $t^{[j]}$ s.t. $t^{[0]} = 0$ and for all $j > 0$, $t^{[j]} = t^{[j-1]} + \tau^{[j-1]}$. This simplifies the encoding and analysis, because it is simple to compare current values of trains' variables. This implies that all switches from a discrete step $j$ to $j + 1$ are globally shared. However, synchronicity also implies that to cover overall dynamics of all trains, the total number of discrete steps must be quite high, which has a bad influence on performance. In the case of [91], the planner considers longer units for unrolling where a step may consist of movements over several segments, and within such a step all deterministic discrete constraints are handled by the simulator.

We use one-hot encoding for some Boolean variables for increased readability.

### 8.3.1 Railway Model

The model itself is represented by a formula entirely. Almost all variables are related to a particular train $T$, including the graph representation. The only variables that are global are those related to the shared time.

**Infrastructure.** The graph that models the network is not itself represented by dedicated variables, but instead, by Boolean variables and constraints related to each particular train $T$, which define possible transitions of the train. All segments $S \in \mathcal{S}$ and nodes $N \in \mathcal{N}$ of the graph are just identifiers that serve as parts of the variable names related to the train. Incidency of all the nodes and segments is covered within the preprocessing stage, which is when the corresponding Boolean variables and constraints are generated into the formula—see below.

**Train.** A train $T \in \mathcal{T}$ is defined by fixed constants $T.A$, $T.B$, $T.V_{max}$, and $T.L$ that represent the properties of the train (acceleration and deceleration rate, velocity limit and length), where $T$ is just a prefix of the constant names, representing an identifier of the train. In a similar way, the state of each train is described by a set of variables, distinguished by a discrete step $j$. The most important variables are:

- *Booleans*: $T.mode^{[j]}$, $mode \in \mathcal{M} = \{idle, steady, acc, brake\}$ ($steady$ means the train does not accelerate, but in mode $idle$, in addition, it has zero velocity); $T.away^{[j]}$, $T.enter^{[j]}$ and $T.finished^{[j]}$ (whether the train is currently outside the graph, whether it is entering, and whether it already finished); and $T.pos\_S^{[j]}$, $pos \in \mathcal{P} = \{back, front, next\}$, for a segment $S \in \mathcal{S}$ (the train's back and front being in $S$; whether $S$ is selected as the next segment).

- *Reals*: $T.a^{[j]}$ (acceleration/deceleration rate); $T.d_{max}{}^{[j]}$ (remaining distance to the end of the current segments, either with the back or the front of the train, i.e., for segments $S$ where $T.back\_S^{[j]}$ or $T.front\_S^{[j]}$ holds); $T.v_{max}{}^{[j]}$ (velocity limit of the current segments and the train itself); and $T.next\_v_{max}{}^{[j]}$ (velocity limit of the selected next segment $S$, for which $T.next\_S^{[j]}$ holds).

- *Functional variables*: $T.d^{[j]}$, $init(T.d^{[j]}) = 0$ (relative distance traveled from the start of unrolling step $j$), and $T.v^{[j]}$ (current velocity). The functional variables range over $\left[0, \tau^{[j]}\right]$, where a timeout $\tau^{[j]} < \rho$, with the constant $\rho$ user-defined, must hold.

This allows decisions on when to enter the network or when to leave the current station to happen in certain intervals—if the timeout is too short, the number of necessary discrete steps may be too high; if it is too long, a plan where trains stay idle for too long may be returned.

**Result.** The resulting plan is represented by the global variables $t^{[j]}$ and the variables $T.idle^{[j]}$, $T.front\_S^{[j]}$ and $T.finished^{[j]}$, for all trains $T$, segments $S$ and discrete steps $j$. All other variables are either auxiliary or are completely determined by the plan and the model described in this subsection.

**Dynamic Phenomena.**

**Mode Conditions.** Unlike in capacity analysis [91], where behavior is deterministic, as soon as routes have been chosen, here continuous dynamics depends on each train's mode, where a train can choose to stay idle in stations, or before entering the network. Each train $T$ is always in exactly one dynamic mode:

$$\bigvee_{mode \in \mathcal{M}} T.mode^{[j]} \wedge \bigwedge_{mode_1, mode_2 \in \mathcal{M}, mode_1 \neq mode_2} \neg\big(T.mode_1^{[j]} \wedge T.mode_2^{[j]}\big) \tag{8.4}$$

and according to this mode, an appropriate (constant) acceleration rate is set:

$$\begin{aligned} &\big((T.idle^{[j]} \vee T.steady^{[j]}) \Leftrightarrow T.a^{[j]} = 0\big) \\ &\quad \wedge \big(T.acc^{[j]} \Leftrightarrow T.a^{[j]} = T.A\big) \wedge \big(T.brake^{[j]} \Leftrightarrow T.a^{[j]} = -T.B\big). \end{aligned} \tag{8.5}$$

To reduce nondeterminism of switching the modes, we add:

$$\begin{aligned} &\Big(T.idle^{[j]} \Rightarrow \big(T.idle^{[j+1]} \vee T.acc^{[j+1]}\big)\Big) \\ \wedge &\Big(\big(T.steady^{[j]} \vee T.acc^{[j]}\big) \Rightarrow \big(\neg T.idle^{[j+1]} \vee T.away^{[j+1]}\big)\Big). \end{aligned} \tag{8.6}$$

Furthermore, sometimes it is clear that braking must follow (i.e. constraints on $T.brake^{[j+1]}$), but it is discussed later, within the paragraph with braking prediction.

There are also other restrictions, like that braking is not possible if the velocity is already zero, or that *steady* mode is not allowed if acceleration is possible.

**Dynamics.** We model the dynamics of trains using the basic laws of motion, but it is possible to extend the model such that it exhibits more complex phenomena, like engine power curves, tunnel air resistance, curve rolling resistance, train weight distribution, etc. Figure 8.2 illustrates how the resulting trajectories of functional variables can look like ($T$ is omitted from the variable names). Both functions $v$ and $d$ are limited by a corresponding dashed line, a constant $v_{max}{}^{[\star]}$ in the case of the function $v$, and a distance limit in the case of $d$, either in the form of a straight line, representing $d_{max}{}^{[\star]}$, or

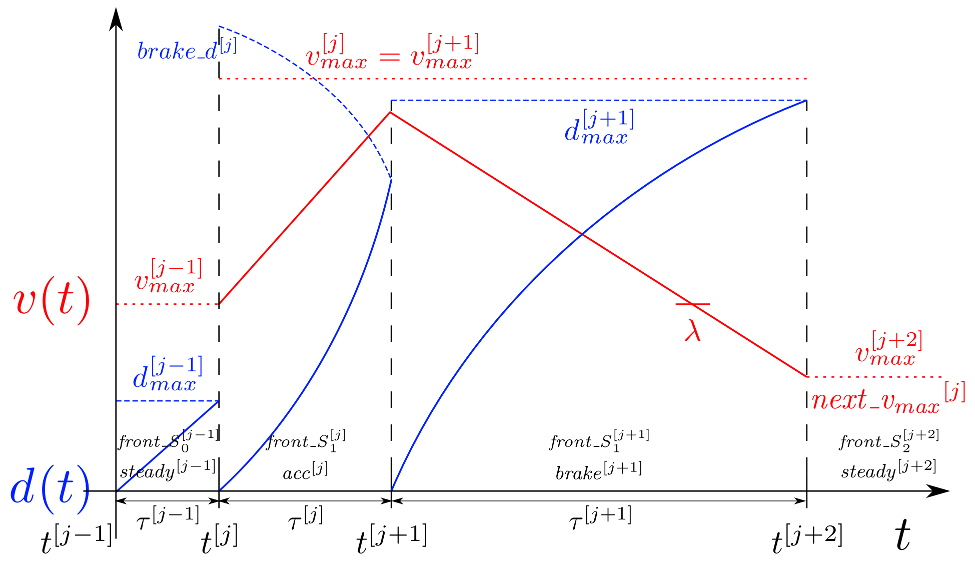


Figure 8.2: Possible train trajectories and their limits.

a curve, that stands for the function $brake\_d^{[\star]}$ that is about to be discussed further. The limit $v_{max}{}^{[j+2]}$ is equivalent to $next\_v_{max}{}^{[j]}$ (and to $next\_v_{max}{}^{[j+1]}$ as well). In the figure, it is always a distance limit that ends each stage, because no velocity limit is exceeded there.

Since trains are modeled synchronously, the dynamics of the trains is represented mainly by one system of ODEs—for each train $T$, and discrete step $j$:

$$\begin{aligned} & T.\dot{d}^{[j]} = T.v^{[j]} \quad \wedge \quad T.\dot{v}^{[j]} = T.a^{[j]} \\ \wedge \quad & T.d^{[j]} \leq T.d_{max}{}^{[j]} \quad \wedge \quad T.v^{[j]} \in \left[0, T.v_{max}{}^{[j]}\right]. \end{aligned} \tag{8.7}$$

The first row of the formula shows particular ODEs, and the second the invariants. Thus, each integration ends when a distance limit or a velocity limit is exceeded, or when the timeout is reached, which was explained in the description of functional variables.

For the definition of the variables $T.\alpha_{max}{}^{[j]}$, $\alpha \in \{d, v\}$, we use auxiliary variables $T.pos\_\alpha_{max}{}^{[j]}$, $pos \in \mathcal{P}$ which correspond to the limits of the current and the next segments, as mentioned in the description of the real variables. Moreover, $T.min\_\alpha_{max}{}^{[j]} := \min\{T.back\_\alpha_{max}{}^{[j]}, T.front\_\alpha_{max}{}^{[j]}\}$. Then, the distance limit is defined by $T.d_{max}{}^{[j]} = T.min\_d_{max}{}^{[j]}$ and the velocity limit as

$$\begin{aligned} \text{ITE}\big(& init(T.v^{[j]}) \geq T.next\_v_{max}{}^{[j]}, \\ & T.v_{max}{}^{[j]} = \min\{T.V_{max}, T.min\_v_{max}{}^{[j]}\}, \\ & T.v_{max}{}^{[j]} = \min\{T.V_{max}, T.min\_v_{max}{}^{[j]}, T.next\_v_{max}{}^{[j]}\}\big), \end{aligned} \tag{8.8}$$

where $T.next\_v_{max}{}^{[j]}$ is used to ensure the correctness of braking prediction.

**Braking Prediction.** In Figure 8.2, within stage $j$, one can see that the function $d$ is limited by a yet unexplained function $brake\_d^{[j]}$. Such a function is necessary for prediction of the moment when a train has to start braking to obey the velocity limit of the next segment—in cases when $T.v^{[j]} > T.next\_v_{max}{}^{[j]}$ (if the train is not already braking). The main idea is to compute the braking trajectory backward from the point where the train enters the next segment, synchronously with the actual forward dynamics. Details follow.

The prediction depends on the relation $init(T.v^{[j]}) \bowtie T.next\_v_{max}{}^{[j]}$, where $\bowtie \in \{=, >\}$. First, let us assume that $init(T.v^{[j]}) = T.next\_v_{max}{}^{[j]}$. To make $T.v^{[j]} > T.next\_v_{max}{}^{[j]}$ happen eventually, $T.acc^{[j]}$ must hold. Such a case would correspond to Figure 8.2, if $next\_v_{max}{}^{[j]}$ was in the place of the separator $\lambda$. Since $T.a^{[j]}$ from Formula 8.7 is a constant (due to Formula 8.5), the ratio between the length (in time) of the acceleration phase and the braking phase is fixed. Since the temporal relationship between the two phases is not yet clear, we use independent time axes, writing

$$\frac{dv_A}{dt_A} = T.A, \frac{dv_B}{dt_B} = -T.B, \tag{8.9}$$

where $v_A$ and $v_B$ corresponds to $T.v^{[j]}$ and $T.v^{[j+1]}$, resp., and $t_A$ and $t_B$ corresponds to $\tau^{[j]}$ and $\tau^{[j+1]}$, resp. To determine the time to switch from acceleration to braking, it would be possible to compute the braking trajectory backward in time starting at the position corresponding to $T.front\_d_{max}{}^{[j]}$, and with the velocity corresponding to $T.next\_v_{max}{}^{[j]}$. However, it is not clear how far backward such a backward braking trajectory has to be computed, and moreover, even after its computation, it is non-trivial to ensure that at the switching time, *both* position and velocity of the train are identical to a corresponding point on the backward braking trajectory. To get around these complications, we not only reverse, but also scale the time axis of the braking process using the relationship

$$t_B = -\frac{T.A}{T.B} \cdot t_A. \tag{8.10}$$

As a result, we have a common time axis $t_A$, along which the derivative of the velocity of the braking train is identical to the derivative of the velocity of the accelerating train:

$$\frac{dv_B}{dt_A} = \frac{dv_B}{dt_B}\frac{dt_B}{dt_A} = -\frac{dv_B}{dt_B}\frac{T.A}{T.B} = T.B \cdot \frac{T.A}{T.B} = \frac{dv_A}{dt_A}. \tag{8.11}$$

As a consequence, both velocities will be identical at all time if starting from the same initial value. Under this assumption, we can compute both the acceleration phase and the backward braking trajectory synchronously along the same time axis, ensuring identical speed at all times. Such an approach can be generalized for more complicated systems of ODEs (e.g. with $T.v^{[j]}$ other than a linear function), if such a relationship between the time axes is available.

Based on Formula 8.11, it suffices to switch from acceleration to braking at the point when the corresponding positions are identical. This results in a synchronous braking prediction with ODEs and an invariant of the form

$$\begin{aligned} &\text{ITE}\big(T.acc^{[j]},\ T.\dot{brake\_d}^{[j]} = -\frac{T.A}{T.B} \cdot T.v^{[j]},\ T.\dot{brake\_d}^{[j]} = 0\big) \\ &\quad \wedge \quad \big(\neg T.brake^{[j]} \Rightarrow T.d^{[j]} \leq T.brake\_d^{[j]}\big) \end{aligned} \tag{8.12}$$

where the coefficient $-\frac{T.A}{T.B}$ implements the mentioned scaling also for the prediction of the position of the train. Note that the same $T.v^{[j]}$ as in Formula 8.7 is used. For example, in the figure, deceleration must proceed more quickly within the braking prediction.

If $init(T.v^{[j]}) > T.next\_v_{max}{}^{[j]}$, the part of the braking phase with $T.v^{[j+1]} \in \big[T.next\_v_{max}{}^{[j]}, init(T.v^{[j]})\big]$ must be precomputed asynchronously. In the figure, this corresponds to the part from the end of stage $j+1$ to the separator $\lambda$ (backwards). Such an asynchronous prediction uses the functional variables $back\_d$ and $back\_v$, starting from

$$init(T.back\_d^{[j]}) = T.front\_d_{max}{}^{[j]} \wedge init(T.back\_v^{[j]}) = T.next\_v_{max}{}^{[j]}, \tag{8.13}$$

with a flow defined by the following ODEs and invariants:

$$\begin{aligned} &T.\dot{back\_d}^{[j]} = -T.back\_v^{[j]} \quad \wedge \quad T.\dot{back\_v}^{[j]} = T.B \\ &\wedge \quad T.back\_d^{[j]} \geq 0 \quad \wedge \quad T.back\_v^{[j]} \leq init(T.v^{[j]}). \end{aligned} \tag{8.14}$$

These functional variables are the only ones that may have a different length $\tau$ of integration than the other variables (which are synchronous). The reached position serves for the consecutive synchronous part:

$$init(T.brake\_d^{[j]}) = final(T.back\_d^{[j]}), \tag{8.15}$$

and Formula 8.12 becomes computable then. This works even in cases when $T.steady^{[j]}$ holds, where $T.brake\_d^{[j]}$ just serves as a constant upper bound on $T.d^{[j]}$, based on the value from Formula 8.15.

Consequently to Formula 8.12, switching to the braking mode is defined by

$$\begin{aligned} &\big(T.acc^{[j]} \Rightarrow (T.brake^{[j+1]} \Leftrightarrow D_j)\big) \\ \wedge\ &\big(T.steady^{[j]} \Rightarrow (T.brake^{[j+1]} \Leftrightarrow (D_j \wedge init(T.v^{[j]}) > T.next\_v_{max}{}^{[j]}))\big) \end{aligned} \tag{8.16}$$

with $D_j \Leftrightarrow final(T.d^{[j]}) \geq final(T.brake\_d^{[j]})$. If already braking, staying in the mode is defined depending on not reaching the end of the current segment yet (see Formula 8.7), and possibly also based on the eventual necessity of a further consecutive braking (due to Formula 8.14). That is

$$\begin{aligned} T.brake^{[j]} \Rightarrow \text{ITE}\big(&final(T.d^{[j]}) < T.d_{max}{}^{[j]},\ T.brake^{[j+1]}, \\ &T.brake^{[j+1]} \Leftrightarrow final(T.back\_d^{[j+1]}) \leq 0\big). \end{aligned} \tag{8.17}$$

If $T.v_{max}{}^{[j]}$ had been reached before the start of the braking phase (i.e. the invariant in Formula 8.7 is violated before the invariant in Formula 8.12), there just would be an additional phase in the steady mode between the phases $j$ and $j+1$ in the figure.

As a result, $T.d^{[j]}$ is limited by both $T.d_{max}{}^{[j]}$ (Formula 8.7) and the braking prediction, which consists of two parts: the asynchronous part (Formula 8.14), and the synchronous—$T.brake\_d^{[j]}$ (Formula 8.12).

**Positional Constraints.** In the following, we use train $T \in \mathcal{T}$ and the relation $S_1 \to_T S_2$ for segments $S_1, S_2 \in \mathcal{S}$ to denote that segment $S_2$ is adjacent to segment $S_1$ on a path that obeys the connection constraints of train $T$. In fact, this relation enforces the connection constraints completely if $T.\mathit{finished}^{[J]}$ (at the final step $J$) holds. The relation is used only within the preprocessing stage when generating the formula.

For each segment $S_1$, the possible next segments are defined s.t.

$$\neg T.idle^{[j]} \Rightarrow \Big(T.front\_S_1^{[j]} \Rightarrow \bigvee_{S_2 \in \mathcal{S}, S_1 \to_T S_2} T.next\_S_2^{[j]}\Big). \tag{8.18}$$

In cases when the front of the train is idle, we do not want to choose any next segment:

$$\Big(T.idle^{[j]} \vee \neg \bigvee_{S \in \mathcal{S}} T.front\_S^{[j]}\Big) \Rightarrow \neg \bigvee_{S \in \mathcal{S}} T.next\_S^{[j]}. \tag{8.19}$$

Next, a train cannot be at more segments at once with any of its part. That is, for all segments $S_1$,

$$\bigwedge_{pos \in \mathcal{P}} \ \bigwedge_{S_2 \in \mathcal{S}, S_2 \neq S_1} \neg\Big(T.pos\_S_1^{[j]} \wedge T.pos\_S_2^{[j]}\Big). \tag{8.20}$$

The chosen next segment must be different than the current one:

$$\bigwedge_{pos \in \{back, front\}} \neg\Big(T.pos\_S_1^{[j]} \wedge T.next\_S_1^{[j]}\Big). \tag{8.21}$$

And, situations like where the train's back is farther than its front are forbidden:

$$\bigwedge_{\substack{pos_1 \in \{back, front\} \\ pos_2 \in \{front, next\}}} \ \bigwedge_{S_2 \in \mathcal{S}, S_1 \to_T S_2} \neg\Big(T.pos_2\_S_1^{[j]} \wedge T.pos_1\_S_2^{[j]}\Big). \tag{8.22}$$

Finally, we control the progress of trains in a way that a train must either stay the same, or its back or front has moved forwards (from a previous segment), applied both to the past and to the future. That is, for all segments $S$:

$$\begin{aligned} \neg T.idle^{[j]} \Rightarrow \bigwedge_{\substack{pos_1 \in \{back, front\} \\ pos_2 = \Delta(pos_1)}} &\Big( (T.pos_1\_S^{[j+1]} \Rightarrow (T.pos_1\_S^{[j]} \vee T.pos_2\_S^{[j]})) \\ &\wedge (T.pos_2\_S^{[j]} \Rightarrow (T.pos_2\_S^{[j+1]} \vee T.pos_1\_S^{[j+1]}))\Big), \end{aligned} \tag{8.23}$$

with $\Delta = \{back \mapsto front, front \mapsto next\}$.

**Away Conditions.** Away conditions distinguish the cases when a train already entered the network, or is outside of it. The decision variable *enter* triggers a starting segment:

$$T.enter^{[j]} \Rightarrow \bigvee_{S \in T.Start} \left(\neg T.back\_S^{[j]} \wedge T.front\_S^{[j]}\right), \tag{8.24}$$

where $T.Start$ is the set of starting segments of the train $T$. A next segment is already constrained by Formula 8.18. After entering, the front of the train is at the beginning of the chosen segment, while the back is still outside the network, with the whole train's length. To denote that a train is entirely outside the network, we use

$$T.away^{[j]} \Leftrightarrow \neg\Big(\bigvee_{S \in \mathcal{S}} T.back\_S^{[j]} \vee \bigvee_{S \in \mathcal{S}} T.front\_S^{[j]}\Big). \tag{8.25}$$

The variable *finished* is triggered within the transfer constraints when reaching a boundary in Formula 8.34 below. Once the variable is activated, it implies that at least the front of the train is already outside of the network:

$$T.finished^{[j]} \Rightarrow \neg \bigvee_{S \in \mathcal{S}} T.front\_S^{[j]}. \tag{8.26}$$

Trains that are leaving the network remain in the steady mode, until they get away entirely.

Next, there are constraints related to the *idle* mode:

$$(T.enter^{[j]} \Rightarrow \neg T.idle^{[j]}) \;\wedge\; (T.away^{[j]} \Rightarrow T.idle^{[j]}). \tag{8.27}$$

Then, there are some restrictions on variables in the next discrete step:

$$(T.enter^{[j]} \Rightarrow \neg T.enter^{[j+1]}) \;\wedge\; (T.finished^{[j]} \Rightarrow T.finished^{[j+1]}). \tag{8.28}$$

Finally, restrictions of the related variables must hold—mutual exclusion of entering and finishing; being away before entering the graph, and after finishing:

$$\begin{aligned} &\neg(T.enter^{[j]} \wedge T.finished^{[j]}) \;\wedge\; (\neg T.away^{[j]} \Rightarrow \neg T.enter^{[j+1]}) \\ &\wedge\; T.away^{[j]} \Rightarrow \mathrm{ITE}(T.finished^{[j]},\; T.away^{[j+1]}, \\ &\qquad \neg T.finished^{[j+1]} \wedge (T.away^{[j+1]} \vee T.enter^{[j+1]})). \end{aligned} \tag{8.29}$$

**Transfer Constraints.** These constraints control the transferring of a train to a next segment when the end of one of the current segments is reached (even when stopping). We denote the fact that the back or front of train $T$ reaches the end of segment $S_1 \in \mathcal{S}$ by $T.pos\_exceed\_S_1^{[j]}$, $pos \in \{back, front\}$, which allows the train to move into segment $S_2$:

$$\begin{aligned} \neg T.idle^{[j]} \Rightarrow \bigwedge_{S_2 \in \mathcal{S}, S_1 \rightarrow_T S_2} &\big((T.pos_1\_S_1^{[j]} \wedge T.pos_2\_S_2^{[j]}) \Rightarrow \\ &\mathrm{ITE}(T.pos_1\_exceed\_S_1^{[j]},\; T.pos_1\_S_2^{[j+1]},\; T.pos_1\_S_1^{[j+1]})\big) \end{aligned} \tag{8.30}$$

where $pos_1 \in \{back, front\}$, $pos_2 = \Delta(pos_1)$, $\Delta = \{back \mapsto front, front \mapsto next\}$. For starting segments $S \in T.Start$, it is also necessary to eventually move the back of the train inside the network (which is not initially there, as stated in Formula 8.24):

$$\left(\neg T.idle^{[j]} \land \neg \bigvee_{S \in \mathcal{S}} T.back\_S^{[j]}\right) \Rightarrow \left(T.back\_inside^{[j]} \Leftrightarrow \bigvee_{S \in T.Start} T.back\_S^{[j+1]}\right), \tag{8.31}$$

where $T.back\_inside^{[j]}$ means that the train reaches the beginning of a starting segment with its back.

In *idle* mode, no transfers happen from any segment $S$:

$$\left(T.idle^{[j]} \land \neg T.enter^{[j+1]}\right) \Rightarrow \bigwedge_{pos \in \{back, front\}} \left(T.pos\_S^{[j]} \Leftrightarrow T.pos\_S^{[j+1]}\right). \tag{8.32}$$

Also, when a train is inside a single segment, the back stays within:

$$\left(T.back\_S^{[j]} \land T.front\_S^{[j]}\right) \Rightarrow T.back\_S^{[j+1]}. \tag{8.33}$$

Finally, when a train exceeds a segment $S \in \mathcal{S}$ that is boundary, the train is claimed as finished based on the front of the train:

$$T.front\_S^{[j]} \Rightarrow \text{ITE}(T.front\_exceed\_S^{[j]}, T.finished^{[j+1]}, T.front\_S^{[j+1]}), \tag{8.34}$$

and it is claimed as away based on its back:

$$T.back\_S^{[j]} \Rightarrow \text{ITE}(T.back\_exceed\_S^{[j]}, T.away^{[j+1]}, T.back\_S^{[j+1]}). \tag{8.35}$$

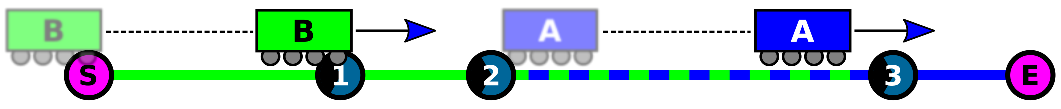


Figure 8.3: A conflicting plan of two consecutive trains with no stops.

**Mutual Exclusion Conditions.** Here we prevent trains from collisions. For each train $T_1$ and for all segments $S$, all the mutual exclusion conditions are jointly defined as

$$\bigwedge_{pos_1, pos_2 \in \mathcal{P}} \bigwedge_{T_2 \in \mathcal{T}, T_2 \neq T_1} \neg\left(T_1.pos_1\_S^{[j]} \land T_2.pos_2\_S^{[j]}\right). \tag{8.36}$$

Thus, we require the segments adjacent to the current front segment to be free (because $next \in \mathcal{P}$)—while it is whole sections[3] in the case of [91], as a consequence of signal interlocking. As a result, tighter plans are possible in our case, but the algorithm may also be forced to resolve more violations of mutual exclusion conditions. Figure 8.3 illustrates a situation where train $A$ is followed by train $B$ that enters as soon as train $A$ leaves node 2. Since the segment 2–3 is long, train $B$ will reach node 1 sooner than train $A$ leaves node 3, resulting in a conflict at segment 2–3 that is claimed by train $B$ as the next segment.

[3]In [91], such sections are called "elementary routes".

**Initial Conditions.** At the beginning, each train stands still, either is away or starts its journey, and is not finished. And some train has to enter:

$$\begin{aligned}&\bigwedge_{T\in\mathcal{T}}\left(init(T.v^{[0]})=0\ \wedge\ (T.enter^{[0]}\vee T.away^{[0]})\ \wedge\ \neg T.finished^{[0]}\right)\\ \wedge\ &\bigvee_{T\in\mathcal{T}} T.enter^{[0]}.\end{aligned} \tag{8.37}$$

**Final Conditions.** In order to satisfy the connection constraints of trains completely, we require the trains to have finished moving through the network at the final unrolling step $J$:

$$\bigwedge_{T\in\mathcal{T}}(T.finished^{[J]}\wedge T.away^{[J]}). \tag{8.38}$$

## 8.3.2 Schedule Constraints

Constraints of the model itself must be satisfiable, otherwise it likely means that it is corrupt (e.g. the infrastructure is disconnected). Unsatisfiability of connection constraints is possible, but rarely desirable (i.e. cases with connection lists where it is not possible to visit all the nodes in the given order wrt. the graph). It is the schedule constraints that make the satisfiability problem interesting.

Schedule formulas enforce schedule constraints and their Boolean combinations. Orderings and timings described in Section 8.2.4 are translated into particular constraints related to visiting nodes at discrete steps. To encode such a visit related to train $T\in\mathcal{T}$, node $N\in\mathcal{N}$, where $\mathcal{N}$ represents the set of nodes of the network, and discrete step $j$, we use auxiliary Boolean variables $T.visit\_N^{[j]}$, $visit\in\{arrive, depart\}$, defined s.t.

$$\begin{aligned}T.arrive\_N^{[j]}\Leftrightarrow&\begin{cases}\bot, & \text{if } j=0, \text{else}\\ \bigvee_{S,S\to_T N} T.front\_S^{[j-1]} & \\ \qquad\wedge\, T.finished^{[j]}, & \text{if } N\in T.End,\ \text{otherwise}\\ \bigvee_{S,N\to_T S}\left(\neg T.front\_S^{[j-1]}\ \wedge T.front\_S^{[j]}\right) & \\ \qquad\wedge\,\neg T.enter^{[j]}; & \end{cases}\\ T.depart\_N^{[j]}\Leftrightarrow&\begin{cases}T.enter^{[j]}, & \text{if } N=T.Start, \text{else}\\ \bot, & \text{if } j=0\vee N\in T.End,\ \text{otherwise}\\ \bigvee_{S,N\to_T S} T.front\_S^{[j]}\ \wedge T.acc^{[j]}\wedge init(T.v^{[j]})=0, & \end{cases}\end{aligned} \tag{8.39}$$

where $N\to_T S$ and $S\to_T N$ means incidence of the node $N$ and segment $S\in\mathcal{S}$ within the train $T$'s connection, in the corresponding direction; $T.Start$ is the starting node of train $T$, and $T.End$ is the set of the train's ending nodes.

**Ordering.** Formula 8.1 enforces an order of $visit_1$ and $visit_2$. Cases with $\bowtie\in\{<,\leq\}$ requires to forbid $visit_2$ to take place before $visit_1$, *and* to make sure that $visit_2$ implies that $visit_1$ already happened:

$$\begin{aligned}&\bigwedge_{k=0}^{J}\Big(T_1.visit_1\_N_1^{[k]} \Rightarrow \bigwedge_{l=0}^{K(k)} \neg T_2.visit_2\_N_2^{[l]}\Big)\\ \wedge &\bigwedge_{l=0}^{J}\Big(T_2.visit_2\_N_2^{[l]} \Rightarrow \bigvee_{k=0}^{L(l)} T_1.visit_1\_N_1^{[k]}\Big),\end{aligned} \tag{8.40}$$

where $K(k) = k$, $L(l) = l - 1$ if $\bowtie$ is $<$, and $K(k) = k - 1$, $L(l) = l$ if $\bowtie$ is $\leq$. Orderings with $\bowtie$ being $>$ or $\geq$ are simply handled as cases with $<$ or $\leq$, respectively, with swapped arguments.

For cases where $\bowtie$ is $=$, the produced formula is

$$\bigwedge_{k=0}^{J}\Big(T_1.visit_1\_N_1^{[k]} \Leftrightarrow T_2.visit_2\_N_2^{[k]}\Big). \tag{8.41}$$

**Timing.** In the first place, it is necessary to guarantee that the corresponding time condition holds in cases when all the corresponding visits are active. In cases where $\bowtie \in \{<, \leq\}$, similarly to orderings, we also make sure that violation of the timing implies that the corresponding visits did already happen.

Following Formula 8.2, an *absolute timing* is translated into

$$\bigwedge_{k=0}^{J}\big(T.visit\_N^{[k]} \Rightarrow t^{[k]} \bowtie \xi\big) \wedge \psi. \tag{8.42}$$

If $\bowtie \in \{>, \geq\}$, then $\psi \Leftrightarrow \top$, otherwise

$$\psi \Leftrightarrow \bigwedge_{k=0}^{J}\Big(\neg(t^{[k]} \bowtie \xi) \Rightarrow \bigvee_{l=0}^{k-1} T.visit\_N^{[l]}\Big). \tag{8.43}$$

*Relative timing* constrains a pair of visits. So Formula 8.3 translates to

$$\bigwedge_{j=0}^{J}\Big(T_1.visit_1\_N_1^{[j]} \Rightarrow \Big(\psi_j \ \wedge \ \bigwedge_{k=j}^{J}\big(T_2.visit_2\_N_2^{[k]} \Rightarrow (t^{[k]} - t^{[j]}) \bowtie \xi\big)\Big)\Big), \tag{8.44}$$

where

$$\psi_j \Leftrightarrow \begin{cases} \top, & \text{if } \bowtie \in \{>, \geq\};\\ \bigwedge_{k=j}^{J}\Big(\neg\big((t^{[k]} - t^{[j]}) \bowtie \xi\big) \Rightarrow \bigvee_{l=j}^{k-1} T_2.visit_2\_N_2^{[l]}\Big), & \text{if } \bowtie \in \{<, \leq\}.\end{cases} \tag{8.45}$$

Since timings support both lower and upper bounds, and since Boolean combinations are allowed, it is possible to define interval boundaries and more.

Recall that the variables $t^{[j]}$ only depend on the lengths $\tau^{[j]}$ of integrations, so the timing constraints are checked at the end of each integration.

### 8.3.3 Formula Size

Let $N_N$ be the size of the set of nodes $\mathcal{N}$ and $N_S$ the size of the set of segments $\mathcal{S}$. Although $N_S = \mathcal{O}({N_N}^2)$, the graphs that model railway networks are usually sparse, thus $N_S = \Theta(N_N)$. The number of trains (i.e. size of $\mathcal{T}$) should be negligible compared to $N_N$ and $N_S$. For each train, the number of all the variables scales to $\mathcal{O}(J)$ (recall that $J$ is the last discrete step), with the exception of auxiliary variables $T.visit\_N^{[j]}$, which scale to $\mathcal{O}(J \cdot N_N)$, and the variables $T.pos\_S^{[j]}$, which scale to $\mathcal{O}(J \cdot N_S)$.

The number of particular differential equations and invariants, for each train, scales to $\mathcal{O}(J)$ as well. The number of other atomic predicates, that are part of the railway model, related to a separate train or not, scales to $\mathcal{O}(J \cdot {N_S}^2)$.

In the case of schedule constraints, especially Formula 8.40 and Formula 8.44, the number of produced atomic predicates corresponds to a polynomial of $J$ (assuming that the graph is sparse): $\mathcal{O}(J^3)$ in the case of relative timings with the operator $<$ or $\leq$ (Formula 8.44 and Formula 8.45), and $\mathcal{O}(J^2)$ in the rest cases.

## 8.4 Experimental Part

In Sections 8.1, 8.2 and 8.3, we mentioned differences between our model and algorithm compared to an approach that is based on dedicated railway simulations [91]. Although we support a richer set of schedule constraints, here we stick to case studies that can be handled by both approaches. Our model is not based on signal interlocking and exhibits more nondeterminism (e.g., we allow the trains to wait in stations and before entering the network). There are more differences between both the approaches, though. We firstly focus on a qualitative analysis of the behavior of the tools, where the differences are not that significant. Then, we also show numerical comparisons (e.g. the absolute run-times) of the tools, with a proper discussion.

We use our model from Section 8.3 and our implementation [72] of the algorithm from Chapter 6, and the *railperfcheck* tool [91]. Since we check strong satisfiability (Definition 20) of the formula, all nondeterminism in the model described above stems from discrete decisions[4]. We focus on case studies where it is not trivial to decide whether a plan that meets both ordering and timing constraints exists, that is, if the formula is strongly satisfiable (`sat`), or not (`unsat`). For this, we generalized the experiments named *Gen* in [91], where all the other experiments, in contrast, exhibit easily satisfiable schedule constraints, which should not be challenging for approaches that are based on SAT solving.

Both presented tools use Minisat [46] as the underlying SAT solver. We use Odeint [1] as the ODE solver. We exclude execution time of our preprocessing (i.e., generating the formula), which we did not optimize[5].

[4]For example, although one requires interval timing constraints, this does not mean that all times from the interval will be tried, only all discrete consequences that lead into this interval.

[5]In the worst case ($N_T = N_S = 4$, scenario *all*), the preprocessing takes 12 min.

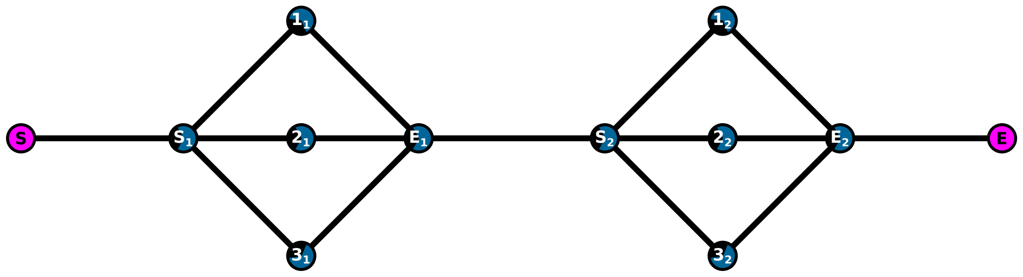


Figure 8.4: An example of a serial-parallel infrastructure, with $N_S = 2$ and $N_P = 3$.

**Specification.** We use a serial-parallel network for our experiments—a track with $N_S$ serially connected groups of $N_P$ identical parallel tracks with a station. See an example in Figure 8.4. For our experiments, we will assume $N_S = N_P$. We use equivalent trains $\mathcal{T} = \{T_1, \ldots, T_{N_T}\}$ with acceleration rate $A = 2$, deceleration rate $B = 1$, velocity limit $V_{max} = 40$ and length $L = 50$. Each train is assigned to connection list $(start, end)$, which only contains the boundary nodes. As a result, multiple paths are possible ($N_P^{N_S}$, at most) for each train. Also, the trains are not allowed to stop at any station, but just drive through, once they enter the network. In the case of [91], it is not possible to force the trains not to stop at stations *and* to make them drive consecutively after each other, due to signal interlocking. However, this fact affects only the numeric comparisons, not the qualitative analysis.

First, we define a scenario $nop$ with no schedule constraints at all, to show that the trains can finish within the given $J$ discrete steps—if $J$ was too low, the result could generally be `unsat`, even with no violations of schedule constraints. In our case, we selected the number of unrollings $J$ manually for each particular experiment—high enough to allow all the trains to finish (i.e., to satisfy Formula 8.38). Such a parameter is not needed in the case of *railperfcheck*.

Next, there are two regular scenarios, $last$ and $all$, that are defined as follows:

- $last$: the last train $T_{N_T}$ must satisfy a relative timing, and the other trains $T_i$ just enter in a given order:

$$timing(T_{N_T}, bnd) \wedge \bigwedge_{i<N_T} \big(enter_{before}(T_i, T_{i+1}) \wedge early_{after}(T_{i+1}, T_i)\big), \tag{8.46}$$

- $all$: each particular train $T_i$ must satisfy a relative timing:

$$\bigwedge_i \Big(timing(T_i, bnd) \wedge \Big(enter_{first}(T_i) \vee \bigvee_{j\neq i} \big(enter_{before}(T_j, T_i) \wedge early_{after}(T_i, T_j)\big)\Big)\Big), \tag{8.47}$$

where $T_i, T_j \in \mathcal{T}$, and with

- $\mathit{timing}(T, \mathit{bnd}) \Leftrightarrow \mathit{transfer}(\mathit{departure}(T, \mathit{start}), \mathit{arrival}(T, \mathit{end})) \bowtie \mathit{bnd}$,
- $\mathit{enter}_{\mathit{before}}(T_1, T_2) \Leftrightarrow \mathit{departure}(T_1, \mathit{start}) < \mathit{departure}(T_2, \mathit{start})$,
- $\mathit{early}_{\mathit{after}}(T_1, T_2) \Leftrightarrow \mathit{departure}(T_1, \mathit{start}) \leq \mathit{arrival}(T_2, \mathit{end}_1)$, and
- $\mathit{enter}_{\mathit{first}}(T) \Leftrightarrow \mathit{departure}(T, \mathit{start}) = 0$,

where $\mathit{end}_1$ is the joint node $E_1$ in the figure. The purpose of $\mathit{early}_{\mathit{after}}$ along with $\mathit{enter}_{\mathit{before}}$ is to avoid long gaps between two consecutive trains, to reduce the amount of nondeterminism of waiting of the trains. Parallel segments which come into the node $\mathit{end}_1$ are the first ones which do not block a preceding train from entering (according to Formula 8.36). As a result, a next train is allowed to enter only as long as the successor drives within these segments. In the case of *railperfcheck*, this is not necessary since waiting is deterministic. Note that in scenario $\mathit{last}$, trains are fully ordered, while in scenario $\mathit{all}$, they are not ordered at all.

Each case study is parametrized by a scenario, variables $N_T, N_S \in \{1, 2, 3, 4\}$, and a timing upper bound $\mathit{bnd} \in \{10^1, 10^2, 10^3\}$. In our case, additionally, $\rho = 30$ (timeout for functional variables in Section 8.3.1), and $J = \Gamma(N_T)$, with $\Gamma = \{1 \mapsto 45, 2 \mapsto 80, 3 \mapsto 115, 4 \mapsto 150\}$. In the case of [91], $J$ is incrementally increased up to $2 \cdot N_T$. Since timings with lower bounds and absolute timings are not supported by *railperfcheck*, we omit them.

Both scenarios *last* and *all* are equivalent in cases with only one train ($N_T = 1$). These are the cases named *Gen* in [91].

Table 8.1: Running time comparison of $\mathit{nop}$ scenario.

| $N_T$ | $N_S$ | Result | Our | conflicts | Their |
|---|---|---|---|---|---|
| 1 | 2 | sat | 0.1 s | 0 | 0 s |
| | 3 | sat | 0.1 s | 0 | 0 s |
| | 4 | sat | 0.1 s | 0 | 0 s |
| 2 | 2 | sat | 0.4 s | 4 | 0 s |
| | 3 | sat | 0.6 s | 26 | 0.1 s |
| | 4 | sat | 1 s | 51 | 0.7 s |
| 3 | 2 | sat | 4.3 s | 29 | 0 s |
| | 3 | sat | 8.3 s | 284 | 0.3 s |
| | 4 | sat | 17 s | 684 | 2.1 s |
| 4 | 2 | sat | 39 s | 106 | 0.1 s |
| | 3 | sat | 1.5 m | 1559 | 0.6 s |
| | 4 | sat | 4 m | 5100 | 4.3 s |

**Results.** We present the qualitative results (`sat` or `unsat`) along with the run times of the tools. Discussion and interpretation of the results follow in the next paragraph.

Firstly, let's see cases with scenario $\mathit{nop}$, that is, with no schedule constraints. The results are shown in Table 8.1. Since all the results are `sat`, it is proved that all trains can finish within the corresponding $J$ steps (i.e. Formula 8.38 holds). We additionally show

Table 8.2: Running time comparison of *last* and *all* scenarios.

| | | | | *last* | | *all* | |
|---|---|---|---|---|---|---|---|
| $N_T$ | $N_S$ | *bnd* | Result | Our | Their | Our | Their |
| 1 | 2 | $10^1$ | `unsat` | | | 0.1 s | 0 s |
| | | $10^2$ | `unsat` | $\rightarrow$ | | 0.1 s | 0 s |
| | | $10^3$ | `sat` | | | 0.1 s | 0 s |
| | 3 | $10^1$ | `unsat` | | | 0.1 s | 0.3 s |
| | | $10^2$ | `unsat` | $\rightarrow$ | | 0.1 s | 0.3 s |
| | | $10^3$ | `sat` | | | 0.1 s | 0 s |
| | 4 | $10^1$ | `unsat` | | | 0.1 s | 3.5 s |
| | | $10^2$ | `unsat` | $\rightarrow$ | | 0.2 s | 3.5 s |
| | | $10^3$ | `sat` | | | 0.1 s | 0 s |
| 2 | 2 | $10^1$ | `unsat` | 0.4 s | 0.6 s | 0.2 s | 0.9 s |
| | | $10^2$ | `unsat` | 2.2 s | 0.6 s | 2.5 s | 0.9 s |
| | | $10^3$ | `sat` | 0.4 s | 0 s | 0.4 s | 0 s |
| | 3 | $10^1$ | `unsat` | 0.8 s | 1.8 m | 0.3 s | 3.5 m |
| | | $10^2$ | `unsat` | 8.9 s | 1.8 m | 9.2 s | 3.5 m |
| | | $10^3$ | `sat` | 0.7 s | 0.1 s | 0.7 s | 0.1 s |
| | 4 | $10^1$ | `unsat` | 1.6 s | $> 2$ h | 0.5 s | $> 2$ h |
| | | $10^2$ | `unsat` | 26 s | $> 2$ h | 29 s | $> 2$ h |
| | | $10^3$ | `sat` | 1.1 s | 0.7 s | 1.1 s | 0.7 s |
| 3 | 2 | $10^1$ | `unsat` | 11 s | 4.6 s | 0.6 s | 24 s |
| | | $10^2$ | `unsat` | 1.1 m | 4.6 s | 2.6 m | 24 s |
| | | $10^3$ | `sat` | 3.7 s | 0 s | 4.2 s | 0 s |
| | 3 | $10^1$ | `unsat` | 38 s | $> 2$ h | 0.9 s | $> 2$ h |
| | | $10^2$ | `unsat` | 13 m | $> 2$ h | 16 m | $> 2$ h |
| | | $10^3$ | `sat` | 6.8 s | 0.3 s | 8.3 s | 0.3 s |
| | 4 | $10^1$ | `unsat` | 2.2 m | $> 2$ h | 1.1 s | $> 2$ h |
| | | $10^2$ | `unsat` | 1.1 h | $> 2$ h | 1 h | $> 2$ h |
| | | $10^3$ | `sat` | 13 s | 2.1 s | 16 s | 2.1 s |
| 4 | 2 | $10^1$ | `unsat` | 3.6 m | 33 s | 1.2 s | 9.3 m |
| | | $10^2$ | `unsat` | 13 m | 33 s | 21 m | 9.3 m |
| | | $10^3$ | `sat` | 25 s | 0.1 s | 39 s | 0.1 s |
| | 3 | $10^1$ | `unsat` | 31 m | $> 2$ h | 1.4 s | $> 2$ h |
| | | $10^2$ | `unsat` | $> 2$ h | $> 2$ h | $> 2$ h | $> 2$ h |
| | | $10^3$ | `sat` | 51 s | 0.6 s | 1.4 m | 0.6 s |
| | 4 | $10^1$ | `unsat` | $> 2$ h | $> 2$ h | 2.1 s | $> 2$ h |
| | | $10^2$ | `unsat` | $> 2$ h | $> 2$ h | $> 2$ h | $> 2$ h |
| | | $10^3$ | `sat` | 2 m | 4.3 s | 3.9 m | 4.3 s |

According to $N_S$, the total number of nodes is $2 \mapsto 10, 3 \mapsto 17, 4 \mapsto 26$, and more importantly, the number of possible paths for each train is $2 \mapsto 4, 3 \mapsto 27, 4 \mapsto 256$.

the number of conflicts which the SAT solver of our tool had to resolve with a backjump. Note that all these conflicts are unrelated to schedule constraints, but are consequences of violations of mutual exclusion conditions instead, as discussed earlier at Figure 8.3.

Concrete results of the scenarios *last* and *all* are shown in Table 8.2. Most importantly, the results of all the specified case studies of these scenarios are as follows:

- `unsat` when $bnd \leq 10^2$: cases with lower timing upper bound are unsatisfiable, that is, it is impossible for the trains to finish within this time bound,
- `sat` when $bnd = 10^3$: plans with high timing upper bound do exist.

**Discussion.** First, we compare the scenarios *last* and *all*. In the satisfiable cases, the run times of both scenarios are similar. In the unsatisfiable cases, the run time of scenario *all* is generally longer than that of *last*, because the trains are unordered and all their permutations are tried, which is a significant effort with multiple trains. On the other hand, to detect that the relative timing of the last train in scenario *last* is unfeasible, all the preceding trains have to be simulated first, regardless the timing. Depending on the value of the timing upper bound *bnd*, it is not certain which part will dominate the run time—simulations of the preceding trains, or of the last train. More on this comparison follows later.

Next, we investigate the behavior of our tool and the tool *railperfcheck* [91]. We start with *railperfcheck*. Recall that it handles mutual exclusion conditions using signal interlocking, which is efficient for networks that do use signals. Moreover, they do simulations in lazy offline fashion, that is, only after a full propositional assignment was found. This is especially efficient in the presented satisfiable cases. However, within the unsatisfiable cases of the scenarios $last$ and $all$, it is entirely insensitive to the value of $bnd$, because only the overall simulation is checked, independently from whether it satisfies the timing or not. In this way, early detection of unsatisfiability is not possible, and all $N_P^{N_S}$ choices of paths are always examined.

In our case, the value of $bnd$ has significant impact on pruning the searched state space in our case—Formula 8.45 ensures termination of all search attempts where it is already obvious that the timing cannot be satisfied. As a result, with growing size of the network and the set of trains, unsatisfiability is detected more efficiently by our more sophisticated algorithm. Consequently, if the timing upper bound is low ($bnd = 10^1$), scenario $all$ is always faster than scenario $last$ in our case, because the unfeasible timing of the train that enters first in the case of scenario $all$ can be detected sooner than that in the case of scenario $last$, where the train that must satisfy the timing is the last one (as discussed above). For example, the run time of our tool in the cases with $N_T = N_S = 3$, $bnd = 10^1$ was 1 s and 38 s in the case of scenario $all$ and $last$, respectively, while with $bnd = 10^2$, it was approximately 15 min in both cases. Also, in the case of just one train ($N_T = 1$), our algorithm detects the symmetry of parallel railroads, and prunes alternative routes. This is especially significant in the cases with large infrastructure ($N_S = 4$) in Table 8.2.

When multiple trains drive consecutively, our method suffers from a number of mutual exclusion conflicts (Formula 8.36). For example, in the case of the conflict captured in Figure 8.3, we resolve it by backtracking the whole situation and seeking another plan where train $B$ enters later. The tool *railperfcheck* prevents such a conflict implicitly within the simulator—by stopping train $B$ at node 1 (if there is a signal) until the conflicting section becomes free. If the signal was not there, such a plan of two consecutive trains would not even be considered.

Thus, it is not surprising that *railperfcheck* is much faster in the satisfiable cases, especially in the scenario $nop$. Our model and/or algorithm should be improved so that such a scenario, with no schedule constraints, is trivial to be solved, like in the case of [91].

When comparing the absolute run times of our tool and *railperfcheck*, we would like to emphasize that the tools solve different problems, and even the underlying scenarios $last$ and $all$ of the corresponding case studies are not equivalent in cases with multiple trains ($N_T > 1$). The issues stem from the way how mutual exclusivity of trains is handled. In the case of *railperfcheck*, it is based on signal interlocking, where whole sections surrounded by signals are being allocated for each train. Since the presented case studies do not require (nor allow) the trains to stop at stations, it would be necessary not to put signals in the place of stations. However, this would also mean that each train always allocates the whole network, so the next one would have to wait until the train that is ahead exits at the boundary, resulting in way too long gaps between trains, and thus in satisfying the timing constraints under different conditions. To avoid this, we placed signals at stations. However, this allows situations like when a train stops in a station and waits until another one overtakes it. While this does not harm the performance in the case of the satisfiable cases, it has quite a significant effect on the run-time performance of unsatisfiable cases, because all combinations of particular overtakings of trains are tried, in addition to all possible routes. We are not aware of a way how to avoid these overtakings, though, even using additional ordering constraints, which do not seem to be possible to be encoded in *railperfcheck* in the case of scenario *all*, because they do not allow Boolean combinations of particular ordering constraints. Also, we did not allow stopping at stations in our case, because otherwise, it would yield an even more difficult problem (with more nondeterminism) than in the case of *railperfcheck*, since we cannot set deterministic waiting times at stations, nor make the trains stop at stations only when it is necessary to avoid a conflict of trains.

In Table 8.2, one can see that at least in the case of a single train ($N_T = 1$), where both approaches actually yield comparable problems, our approach performs significantly faster within the unsatisfiable cases. In the rest unsatisfiable cases, our approach is usually faster, but one has to take the differences of both approaches into consideration. Nevertheless, in the case of low timing upper bound ($bnd = 10^1$), it is expectable that even if *railperfcheck* did not consider plans with overtakings of the trains, our approach would still be faster, because we prune the searched state significantly there.

**Possible Improvements.** In cases of a missed deadline by a train, a possibility is to learn that all other trains, that are *not faster*[6] than the one that participates in the conflict, would miss the deadline too, if choosing the same route (if it is a part of their connection), and if they are constrained by such a deadline as well. Such more sophisticated conflict reasoning would enhance, for example, solving the scenario *all* a lot, because it would avoid trying all the permutations of the trains, in the unsatisfiable cases.

Finally, the synchronous model is not efficient, because the granularity of the discretization is too high. Furthermore, when resolving a conflict, the solver backtracks to a previous state, causing to also throw away simulations that had nothing to do with the conflict. This would not happen in an asynchronous model. It should be replaced by an asynchronous model, with embedding some timing constraints also into the systems of ODEs, to achieve proper synchronization[7]. Note that this does not concern the algorithm, only the encoding of a formula.

## 8.5 Conclusion

We presented a formalization of a low-level railway scheduling problem, where the dynamics of trains is described by differential equations, and where rich timing and ordering constraints are supported. We analyzed the behavior of our approach compared to an existing method on selected case studies, and identified strong and weak aspects of the run-time performance. We demonstrated that despite the complexity of our model,

[6]For example, train $T_2$ is surely not faster than train $T_1$ if $T_2.A \leq T_1.A \wedge T_2.B \leq T_1.B \wedge T_2.V_{max} \leq T_1.V_{max} \wedge T_2.L \geq T_1.L$.

[7]This would also allow efficient handling of timings with the relational operator $=$.

the resulting problems can be solved successfully within a SAT modulo theory framework. This opens the possibility of applying such techniques to further application domains with similar complexity.

CHAPTER 9

# Multi-Agent Path-Finding

This chapter integrates a full version of our paper [76] into the dissertation thesis.

We introduce a new approach to solving a continuous-time version of the multi-agent path-finding (MAPF) problem. The algorithm translates the problem into Satisfiability Modulo Theories (SMT) that can be solved by off-the-shelf solvers. This enables the exploitation of conflict generalization techniques that such solvers can handle. Computational experiments show that the new approach scales better with respect to the available computation time than state-of-the-art approaches and is usually able to avoid their exponential behavior on a class of benchmark problems modeling a typical bottleneck situation.

## 9.1 Introduction

Multi-agent path finding (MAPF) [115, 106, 89, 122] is the problem of navigating agents from their start positions to given individual goal positions in a shared environment so that agents do not collide with each other. The standard discrete variant of the MAPF problem is modeled using an undirected graph in which $k$ agents move instantaneously between its vertices. The space occupancy by agents is modeled by the requirement that at most one agent reside per vertex and via movement rules that forbid conflicting moves that traverse the same edge in opposite directions.

Standard discrete MAPF however lacks expressiveness for various real life problems where continuous time and space play an important role such as robotics applications and/or traffic optimization [50, 92].

This drawback of standard MAPF has been mitigated by introducing various generalizations such as multi-agent path-finding with continuous time (MAPF$_R$) [4]. This allows more accurate modeling of the target application problem without introducing denser and larger discretizations. Especially in applications, where agents correspond to robots, it is important to consider graph edges that interconnect vertices corresponding to more distant positions. It is unrealistic to consider unit time for such edges as done in the standard MAPF, hence general duration of actions must be adopted. The action duration often corresponds to the length of edges which implies fully continuous reasoning over the time domain.

In this chapter, we show how to solve the $\text{MAPF}_R$ problem by directly translating it to an SMT problem (Section 2.4). In this chapter, we will use the theory of quantifier free linear real arithmetic (Section 2.4.1), denoted as QF_LRA in SMT-LIB (Section 5.3.1). This will allow us to reason about time in MAPF modeled in a continuous manner.

An example of a state-of-the-art approach for $\text{MAPF}_R$ is Continuous-time Conflict-based Search (CCBS) [4], a generalization of Conflict-based Search (CBS) [114] that represents one of the most popular algorithms for MAPF. State-of-the-art approaches search for optimal plans. However, in real-world applications, where the formalized MAPF problem results from an approximation of the original application problem, an overly strong emphasis on optimality is often pointless. Moreover, it may result in non-robust plans that are difficult to realize in practice [7]. Hence we aim for a sub-optimal method whose level of optimality can be adapted to the needs in the given application domain.

Unlike methods based on CCBS that approaches the optimum from below by iterating through plans that still contain collisions, our method approaches the optimum from above, iterating through collision-free plans. This has the advantage that—after finding its first plan—our method can be interrupted at any time, still producing a collision-free, and hence feasible plan. This anytime behavior is highly desirable in practice [82].

Another advantage over existing methods is the fact that the objective function is a simple expression handed over to the underlying SMT solver. This allows any objective function that the SMT solver is able to handle without the need for any algorithmic changes.

We did experiments comparing our method with the state-of-the-art approaches CCBS and SMT-CCBS [4] on three classes of benchmark problems and various numbers of agents. The results show that our method is more sensitive to time-outs than the existing approaches, typically being able to solve more instances than existing approaches for high time-outs and less for lower time-outs. Future improvement of computer efficiency will consequently make the method even more competitive.

Moreover, for one class of benchmark problems—modeling a bottleneck situation where all agents have to queue for passing a single node, the new method usually avoids the exponential behavior of CCBS and SMT-CCBS whose run-times explode from a certain number of agents on. Such bottleneck situations frequently occur for certain types of application problems (e.g., in traffic problems or navigation of characters in computer games through tunnels and the like).

**Further Related Work:** Existing methods for generalized variants of MAPF with continuous time include variants of Increasing Cost Tree Search (ICTS) [120] where durations of individual actions can be non-unit. The difference from our generalization is that agents do not have an opportunity to wait an arbitrary amount of time but wait times are predefined via discretization. Similar discretization has been introduced in the Conflict-based Search algorithm [34]. Since discretization in case of ICTS as well as CBS brings inaccuracies of representation of the time, it is hard to define optimality. Moreover, a more accurate discretization often increases the number of actions, which can lead to an excessively large search space.

Our method for MAPF$_R$ comes from the stream of compilation-based methods for MAPF, where the MAPF instance is compiled to an instance in a different formalism for which an off-the-shelf efficient solver exists. Solvers based on formalisms such as Boolean satisfiability (SAT) [118, 117], Answer Set Programming [19], Constraint Programming (CP) [105, 55], or Mixed-integer Linear Programming (MILP) [79] exist. The advantage of these solvers is that any progress in the solver for the target formalism can be immediately reflected in the MAPF solver that it is based on.

The earlier MAPF method related to SMT (the SMT-CBS algorithm) [117] separates the rules of MAPF into two logic theories, one theory for conflicts between agents and one theory for the rest of the MAPF rules. The two theories are used to resolve conflicts between agents lazily similarly as it is done by the CBS algorithm.

The application of SAT and SMT solvers to planning problem different from MAPF is not new [104, 81, 30], usually in the context of temporal and numerical planning—extensions of the classical planning problem with numerical variables. We have used an SMT solver for a specific planning problem with multiple agents (Chapter 8), employing however a synchronous model that identifies each step of the unrolled planning problem with a fixed time period.

## 9.2 Problem Definition

We follow the definition of multi-agent path finding with continuous time (MAPF$_R$) from [4].

We define a MAPF$_R$ problem by the tuple $(G, M, A, s, g, \mathit{coord})$, where $G = (V, E)$ is a directed graph with $V$ modeling important positions in the environment and $E$ modeling possible transitions between the positions, $M$ is a metric space that models the continuous environment, $A = \{a_1, a_2, ..., a_k\}$ is a set of agents, functions $s : A \to V$ and $g : A \to V$ define start and goal vertices for the agents, and $\mathit{coord} : V \to M$ assigns each vertex a coordinate in metric space $M$.

The edges $E$ define a set of possible move actions, where each $e = (u, v) \in E$ is assigned a duration $e_D \in \mathbb{R}_{>0}$ and a motion function $e_M : [0, e_D] \to M$ where $e_M(0) = \mathit{coord}(u)$ and $e_M(e_D) = \mathit{coord}(v)$. In addition to this, there is infinite set of wait actions associated with each vertex $v \in V$ such that an agent can wait in $v$ any amount of time. The motion function for a wait action is constant and equals to $\mathit{coord}(v)$ throughout the duration of the action.

We define collisions between a pair of agents based on a collision-detection predicate $\textsc{IsCollision} \subseteq A \times A \times M \times M$ such that $\textsc{IsCollision}(a_i, a_j, m_i, m_j)$ if and only if the bodies of agents $a_i$ and $a_j$ overlap at coordinates $m_i$ and $m_j$. For this purpose, we assume that the bodies are open sets and overlapping is understood to be strict. Hence agents are permitted to touch if they are assumed to have a closed boundary which is not defined as a collision.

The algorithm described in this chapter is abstract in the sense that it does not explicitly restrict the class of motion actions. Instead it assumes that it is possible to do collision detection and avoidance, as described in Section 9.5. This is possible, for example, if the agents and motion functions are described by polynomials, due to the fact that the theory of real closed fields allows quantifier elimination. Note that this allows the modeling of non-constant agent speed and of movements along non-linear curves. Still, in our implementation, for reasons of efficiency and ease of implementation, the motion functions are required to be linear.

Given a sequence of actions $\pi = (e_1, e_2, ..., e_n)$, we generalize the duration and motion functions from individual actions to overall $\pi$ which we denote by $\pi_D$ and $\pi_M$, respectively. Let $\pi[:n'] = (e_1, e_2, ..., e_{n'})$ denote the prefix of the sequence of actions, then $\pi_D = \sum_{i=1}^{n} e_{iD}$ and analogously $\pi[:n']_D = \sum_{i=1}^{n'} e_{iD}$. The motion function $\pi_M$ needs to take into account the relative time of individual motion functions $e_{iM}$, that is: $\pi_M(t) = e_{1M}(t)$ for $t \leq e_{1D}$, ..., $\pi_M(t) = e_{n'M}(t - \pi[:n'-1]_D)$ for $\pi[:n'-1]_D \leq t \leq \pi[:n']_D$, ..., $\pi_M(t) = e_{nM}(e_{nD})$ for $\pi_D < t$. The last case means that the agent stops after executing the sequence of actions and stays at the coordinate of the goal vertex.

**Definition 25.** *There is a collision between sequences of actions $\pi_i$ and $\pi_j$ if and only if $\exists t \in [0, \max\{\pi_{iD}, \pi_{jD}\}]$ such that* $\textsc{IsCollision}(a_i, a_j, \pi_{iM}(t), \pi_{jM}(t))$.

**Definition 26.** *A* pre-plan *of a given MAPF$_R$ problem $(G, M, A, s, g, coord)$ is a collection of sequences of actions $\pi_1$, $\pi_2$, ..., $\pi_k$ s.t. for every $i \in \{1, \ldots, k\}, \pi_i(0) = s(a_i)$ and $\pi_i(\pi_{iD}) = g(a_i)$. A* plan *for given MAPF$_R$ problem $P$ is a pre-plan of $P$ whose sequences are pair-wise collision free.*

We define several types of cost functions that we denote by $cost(\Pi)$, for a given plan $\Pi$. For example, we will work with sum-of-costs (in this case, $cost(\Pi) = \sum_{i=1}^{k} cost(\pi_{iD})$), or makespan.

For a MAPF$_R$ problem $P$, we denote by $opt(P)$ its optimal plan and by $opt_{pre}(P)$ its optimal pre-plan. Clearly $cost(opt_{pre}(P)) \leq cost(opt(P))$, but $opt_{pre}(P)$ is much easier to compute than $opt(P)$ since it directly follows from the plans of the individual agents.

For our approach, the following two observations are essential:

- Multiple subsequent wait actions can always be merged into a single one without changing the overall motion.
- It is always possible to insert a wait action of zero length between two subsequent move actions—again without changing the overall motion.

Due to this, we can restrict the search space to plans for which each wait action is immediately followed by a move action and vice versa. Our SMT encoding will then be able to encode wait and move actions in pairs, which motivates counting the number of steps of plans by just counting move actions. Hence, for a sequence of actions $\pi$ we denote by $|\pi|$ the number of move actions in the sequence, and for a plan $\Pi$, we call $|\Pi| := max_{i=1}^{k}|\pi_i|$ the number of steps of plan $\Pi$.

## 9.3 Algorithm

Our goal is to encode the planning problem in an SMT theory that is rich enough to model time and to represent conflict generalization constraints. Since SMT solvers only encode a fixed number of steps, we have to use a notion of optimality that takes this into account. Hence the first optimization criterion is the number of steps, and the second criterion cost, which we optimize up to a given $\delta > 0$:

**Definition 27.** *A plan $\Pi$ satisfying a MAPF$_R$ problem $P$ is* minstep $\delta$-optimal *iff*

- $|\Pi| = \min\{|\Pi| \mid \Pi$ *is a plan of* $P\}$*, and*
- $cost(\Pi) \leq (1+\delta)\inf\{cost(\Pi') \mid |\Pi'| = |\Pi|\}$.

The result is Algorithm 9.1. It searches from below for a plan of minimal number of steps, and then minimizes cost for the given number of steps using iterative bisection. For this, it uses a function *findplan* that searches for a plan with a fixed number of states whose cost is between some minimal and maximal cost and that we will present in more details below in Algorithm 9.2.

When using a SAT solver to implement the function *findplan*, it would be an overkill to encode the *whole* planning problem at once, since we would have to encode the avoidance of a huge number of potential collisions. Instead, we will encode this information on demand, initially looking for a pre-plan, and adding information on collision avoidance only based on collisions that have already occurred.

However, whenever a collision occurs, we do not only avoid the given collision, but also collisions that are in some sense similar. We will call this a *generalization* of a collision which we will also formalize in Section 9.5.

So denote by $\varphi_{P,h,\underline{t},\overline{t}}$ an SMT formula encoding the existence of a pre-plan $\Pi$ of MAPF$_R$-problem $P$ with number of steps $h$ and cost in $[\underline{t},\overline{t}]$, that is, $|\Pi| = h$ and $cost(\Pi) \in [\underline{t},\overline{t}]$ (see Section 9.4 for details). We will use an SMT solver to solve those formulas and assume that for any formula $\varphi$ encoding a planning problem, $sat(\varphi)$ either returns the pre-plan satisfying $\varphi$ or $\perp$ if $\varphi$ is not satisfiable.

The result is Algorithm 9.2 below.

Note that if $p \neq \perp$, the pre-plan $p$ may have several collisions. The algorithm leaves it open for which of those collisions to add collision avoidance information into the formula $\varphi_{c_{new}}$. The algorithm leaves it open, as well, how much to generalize a found collision occurring at a certain point in time. In our approach, we use a specific choice here that we will describe in Section 9.5.

**Theorem 1.** *The main algorithm is correct, and if $\hat{t}$ is chosen as $(1-c)t_{\min} + ct_{\max}$, for some fixed $c \in (0,1)$, then it also terminates.*

**Algorithm 9.1:** Main algorithm MAPF-LRA.

```
MAPF-LRA$(P, \delta) \rightarrow p_{opt}$
**Input:**
  - a MAPF$_R$ problem $P = (G, M, A, s, g, coord)$
  - $\delta \in \mathbb{R}_{>0}$
**Output:**
  - $p_{opt}$: a minstep $\delta$-optimal plan for $P$

$h \leftarrow |opt_{pre}(P)|$
$t_{min} \leftarrow cost(opt_{pre}(P))$
$C \leftarrow \emptyset$
$(p, C) \leftarrow findplan(P, h, t_{min}, \infty, C)$
while $p = \bot$ do
    $h \leftarrow h + 1$
    $(p, C) \leftarrow findplan(P, h, t_{min}, \infty, C)$
$p_{opt} \leftarrow p$
while $cost(p_{opt}) > (1 + \delta)t_{min}$ do
    let $\hat{t} \in (t_{min}, cost(p_{opt}))$
    $(p, C) \leftarrow findplan(P, h, t_{min}, \hat{t}, C)$
    if $p = \bot$ then $t_{min} \leftarrow \hat{t}$ else $p_{opt} \leftarrow p$
return $p_{opt}$
```

*Proof.* Since $|opt_{pre}(P)|$ is a lower bound on the number of steps of any plan of $P$, $h \leq \min\{|\Pi| \mid \Pi \text{ is a plan of } P\}$ at the beginning of the first while loop. After termination of the first while loop, $h = \min\{|\Pi| \mid \Pi \text{ is a plan of } P\}$. Moreover, the second while loop does not change $h$, and hence the result of the algorithm certainly satisfies the first condition of Definition 27. Throughout the first loop, $t_{\min}$ is a lower bound on all collision free plans, and throughout the second loop, it is a lower bound on all collision free plans that take $h$ steps, and $p_{opt}$ contains a $h$-step plan. Hence, after termination of the second loop also the second condition of Definition 27 holds.

Finally, if $c \in (0, 1)$, $cost(p_{opt}) - t_{\min}$ goes to zero as the second-while loop iterates. Hence the termination condition of this loop must eventually be satisfied. □

## 9.4 SMT Encoding

In this section, we present an encoding of the planning problem from Section 9.2 as an SMT formula in the quantifier-free theory of linear real arithmetic QF_LRA. Here we concentrate on the formula $\varphi_{P,h,\underline{t},\bar{t}}$ that models time constraints of the agents and their paths in graph $G$ but does *not* model collisions of the agents and metric space $M$. We leave the SMT encoding of collision avoidance to the next section.

**Algorithm 9.2:** Function *findplan* that searches for a plan with a bounded cost.

```
$findplan(h, t_{\min}, t_{\max}, C) \rightarrow (p, C')$
**Input:**
  - $h \in \mathbb{N}_0$
  - $t_{\min} \in \mathbb{R}_{\geq 0}$
  - $t_{\max} \in \mathbb{R}_{\geq 0} \cup \{\infty\}$
  - $C$: a set of formulas that every plan must satisfy
**Output:**
  - $p$: either a plan $\Pi$ with $|\Pi| = h$ and $cost(\pi) \in [t_{\min}, t_{\max}]$, or
      $\perp$, if such a plan does not exist
  - $C'$: a set of formulas that every plan must satisfy
$p \leftarrow sat(\varphi_{P,h,t_{\min},t_{\max}} \wedge \bigwedge_{\varphi_c \in C} \varphi_c)$
while $\neg[p = \perp \vee p$ is collision-free$]$ do
    let $\varphi_{c_{new}}$ represent the generalization of collisions in $p$
    $C \leftarrow C \cup \{\neg\varphi_{c_{new}}\}$
    $p \leftarrow sat(\varphi_{P,h,t_{\min},t_{\max}} \wedge \bigwedge_{\varphi_c \in C} \varphi_c)$
return $(p, C)$
```

**Variables.** As usual in planning applications of SAT solvers [104], we unroll the planning problem in a similar way as in Bounded Model Checking (Section 2.5), where each step $0, 1, \ldots, h$ corresponds to one wait and one move action. As a consequence, unrolling over $h$ steps corresponds to search for a pre-plan $\Pi$ with $|\Pi| = h$.

Note that $h$ corresponds to the maximum of the steps of all agents, so an agent that already reached the goal in step $j < h$ may remain in the same state in steps $j, \ldots, h$, without any further actions.

Each agent is modeled using a separate set of Boolean and real-valued variables. For each agent $a \in A$ and discrete step $j$, we define variables $V_a^{[j]}$, $T_a^{[j]}$, $w_a^{[j]}$ and $m_a^{[j]}$: We model the vertex position of the agent by $V_a^{[j]}$ which is a Boolean encoding of a vertex $v \in V$ using $\mathcal{O}(|V|)$ or $\mathcal{O}(\log(|V|))$ Boolean variables. We will use the notation $V_a^{[j]} = v$ to denote a constraint that expresses that an agent occupies vertex $v \in V$ at the beginning of discrete step $j$. We will also use $V_a^{[j]} \neq v$ as an abbreviation for $\neg\left(V_a^{[j]} = v\right)$.

Next, we model time constraints of the agent using real variables $T_a^{[j]}$, $w_a^{[j]}$, and $m_a^{[j]}$. The variables $T_a^{[j]}$ model the absolute time when the agent occupies a vertex that corresponds to $V_a^{[j]}$, before it takes further actions within discrete step $j$ (or later). The variables $w_a^{[j]}$ model the duration of wait actions and the variables $m_a^{[j]}$ the duration of move actions.

Finally, we use an auxiliary real variable $\lambda$ that, for a pre-plan $\Pi$, corresponds to $cost(\Pi)$. The objective is to minimize this variable. There may be arbitrary linear constraints on the variable, allowing specification of rich cost functions.

**Constraints.** We define (1) initial and goal conditions on the agents, (2) constraints that ensure that the agents follow paths through the graph $G$, and (3) time constraints that correspond to occurrences of the agents at vertices of the graph. For that, we only use the variables defined above.

The initial and goal conditions ensure that each agent $a$ visits the start and goal vertex at the beginning and end of the plan, respectively: $V_a^{[0]} = s(a) \wedge V_a^{[h]} = g(a)$.

To ensure that the agents follow paths through the graph $G$, we use for each agent $a$ and $j < h$ a constraint ensuring that the pair of vertex positions $V_a^{[j]}$ and $V_a^{[j+1]}$ corresponds to an edge of the graph $G$. However, this is not necessarily the case for an already finished agent, that is, if $V_a^{[j]} = g(a)$, then also $V_a^{[j+1]} = g(a)$ is allowed.

The time constraints ensure that the initial value of time of all agents is zero: $T_a^{[0]} = 0$. For $j > 0$, they assume that during each discrete step, an agent may first wait and then it moves, resulting in the constraint $T_a^{[j]} = T_a^{[j-1]} + w_a^{[j-1]} + m_a^{[j-1]}$. For the waiting times, we require $w_a^{[j]} \geq 0$. In addition, we ensure that at least one agent starts to move at the beginning of a plan without waiting, asserting $\bigvee_{a \in A} w_a^{[0]} = 0$. For the moving times, if $j < h$ we ensure that $m_a^{[j]}$ corresponds to the duration of the edge between $V_a^{[j]}$ and $V_a^{[j+1]}$. In addition, $m_a^{[j]} = 0$ if $j = h$ or $V_a^{[j]} = g(a) \wedge V_a^{[j+1]} = g(a)$.

Note that agents are modeled asynchronously, meaning that for a pair of agents $a, b \in A$, $T_a^{[j]}$ and $T_b{}^{[j]}$ corresponds *not* necessarily to the same moment in time. This implies that comparing times and the corresponding positions of agents, in order to check whether there are collisions, cannot be done in a straightforward way, and in the worst case, variables corresponding to all discrete steps must be examined.

We present two variants of cost functions: sum of costs, defined as $\lambda = \sum_{a \in A} T_a^{[h]}$, and makespan, defined as $\lambda = \max_{a \in A} T_a^{[h]}$. To ensure that the formula $\varphi_{P,h,\underline{t},\bar{t}}$ satisfies the bounds of the cost function, we simply require $\lambda \geq \underline{t} \wedge \lambda \leq \bar{t}$. An example of an alternative cost function is $\lambda = \sum_{a \in A} \sum_{j=0}^{h-1} \left(2m_a^{[j]} + w_a^{[j]}\right)$ which prefers minimizing moving times over waiting times and can therefore result in more power-optimal plans.

Building the formula $\varphi_{P,h,\underline{t},\bar{t}}$ from scratch after each increase of the number of steps $h$ would be inefficient. Hence we build the formula incrementally. However, some parts of the formula (e.g., the cost functions or constraints such as $V_a^{[h]} = g(a)$), explicitly depend on $h$, and hence need to be updated when $h$ is increased. Here we use the feature of modern SMT solvers, that allow the user to cancel constraints asserted after a previously specified milestone, and to reuse the rest.

## 9.5 Collision Detection and Avoidance

Whenever the algorithm *findplan* from Section 9.3 computes a pre-plan that still contains a collision, it represents the generalization of collisions by a formula $\varphi_{c_{new}}$ whose negation it then adds to the formula passed to the SMT solver. We will now discuss how to first detect collisions and how to then construct the formula $\varphi_{c_{new}}$ generalizing detected collisions. Here, we will assume precise arithmetic, deferring the discussion of implementation in finite computer arithmetic to Section 9.6.

**Collision Detection.** Assume that two agents $a$ and $b$ follow their motion functions $\alpha_{Ma} : [0, \alpha_{Da}] \to M$ and $\alpha_{Mb} : [0, \alpha_{Db}] \to M$ with durations $\alpha_{Da}$ and $\alpha_{Db}$, respectively, corresponding to either a move or a wait action. Assume that the agents start the motions at points in time $\hat{\tau}_a$ and $\hat{\tau}_b$, respectively. To determine whether there is a collision, we will use the abstract predicate IsCollision introduced in Section 9.2. Based on this, we can check for a collision of two agents that follow motion functions starting at certain times:

**Definition 28.** *For two motion functions $\alpha_{Ma}$ and $\alpha_{Mb}$ with respective starting times $\hat{\tau}_a$ and $\hat{\tau}_b$, $\textsc{InConflict}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b)$ iff*

$$\exists t \in [\hat{\tau}_a, \hat{\tau}_a + \alpha_{Da}] \cap [\hat{\tau}_b, \hat{\tau}_b + \alpha_{Db}] \;.\; \textsc{IsCollision}(a, b, \alpha_{Ma}(t - \hat{\tau}_a), \alpha_{Mb}(t - \hat{\tau}_b)).$$

We will now discuss the construction of the formula $\varphi_{c_{new}}$ that generalizes collisions of pre-plans found in Algorithm 9.2. A found pre-plan may result in several such collisions. We start with generalizing one of them and consider two cases: The case of a collision between two moving agents, and the case of a collision between a waiting and a moving agent. We can ignore the case when both agents are waiting: Such a conflict either should have been avoided already in the previous discrete steps, or the agents must in the case of a conflict overlap right at the beginning of a pre-plan, resulting in a trivially infeasible plan.

**Collisions While Moving.** In this case, one of the two agents has to wait until the conflict vanishes. We are interested in waiting the minimal time and hence define $\textsc{Safe}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b) :=$

$$\inf\{\tau_a \mid \hat{\tau}_a < \tau_a, \neg\textsc{InConflict}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \tau_a, \hat{\tau}_b)\}.$$

Note that $\hat{\tau}_a < \textsc{Safe}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b) \leq \hat{\tau}_b + \alpha_{Db}$. Here, the lower bound is a consequence of the assumption that agents are open sets which makes collisions happen in the interior of those sets. The upper bound follows from the fact that $\textsc{InConflict}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_b + \alpha_{Db}, \hat{\tau}_b)$ is always false.

Assume that we detected a conflict between two move actions starting at times $\hat{\tau}_a$ and $\hat{\tau}_b$ and hence $\textsc{InConflict}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b)$. We know that any value of $\tau_a$ with $\hat{\tau}_a \leq \tau_a < \textsc{Safe}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b)$ also leads to a conflict. In addition—letting the second agent wait—any value of $\tau_b$ with $\hat{\tau}_b \leq \tau_b < \textsc{Safe}_{b,a}(\alpha_{Mb}, \alpha_{Ma}, \hat{\tau}_b, \hat{\tau}_a)$ also leads to a conflict.

However, we know even more. For seeing this, observe that $\textsc{InConflict}_{a,b}$ is invariant wrt. translation along the time-axes, that is, for every $\Delta \in \mathbb{R}$, $\textsc{InConflict}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b)$ iff $\textsc{InConflict}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a + \Delta, \hat{\tau}_b + \Delta)$ which can be seen by simply translating the witness $t$ from Definition 28 by the same value $\Delta$. Due to this, the same conflict happens for all pairs $(\tau_a, \tau_b)$ with the same relative distance as the relative distance of $(\hat{\tau}_a, \hat{\tau}_b)$. Hence we know that both

$$\hat{\tau}_a - \hat{\tau}_b \leq \tau_a - \tau_b < \textsc{Safe}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b) - \hat{\tau}_b$$

and

$$\hat{\tau}_b - \hat{\tau}_a \leq \tau_b - \tau_a < \textsc{Safe}_{b,a}(\alpha_{Mb}, \alpha_{Ma}, \hat{\tau}_b, \hat{\tau}_a) - \hat{\tau}_a$$

lead to a conflict.

Multiplying the second inequality by $-1$, we get

$$\hat{\tau}_a - \textsc{Safe}_{b,a}(\alpha_{Mb}, \alpha_{Ma}, \hat{\tau}_b, \hat{\tau}_a) < \tau_a - \tau_b \leq \hat{\tau}_a - \hat{\tau}_b$$

and combining the result with the first inequality, we get

$$\begin{aligned} &\hat{\tau}_a - \textsc{Safe}_{b,a}(\alpha_{Mb}, \alpha_{Ma}, \hat{\tau}_b, \hat{\tau}_a) < \tau_a - \tau_b \\ \wedge\ &\tau_a - \tau_b < \textsc{Safe}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b) - \hat{\tau}_b. \end{aligned}$$

For applying this to the variables of the SMT encoding described in Section 9.4, we denote this formula by $\varphi^{mm}(a, b, \alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b, \tau_a, \tau_b)$, replacing the arguments by the corresponding terms from the SMT encoding. More concretely, we observe that the start of a move action of agent $a$ at step $j_a$ is modeled by the term ${T_a}^{[j_a]} + {w_a}^{[j_a]}$ and the start of a move action of agent $b$ at step $j_b$ by the term ${T_b}^{[j_b]} + {w_b}^{[j_b]}$.

Now assume that we detected a conflict of two agents $a$ and $b$ moving along respective edges $(u_a, v_a)$ and $(u_b, v_b)$, starting in discrete steps $j_a$ and $j_b$ and times ${\hat{T}_a}^{[j_a]} + {\hat{w}_a}^{[j_a]}$ and ${\hat{T}_b}^{[j_b]} + {\hat{w}_b}^{[j_b]}$ (the hats denoting the values assigned to the respective variables). In this case, $\textsc{InConflict}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, {\hat{T}_a}^{[j_a]} + {\hat{w}_a}^{[j_a]}, {\hat{T}_b}^{[j_b]} + {\hat{w}_b}^{[j_b]})$, and the formula $\varphi_{c_{new}}$ has the form

$$\begin{aligned} &{V_a}^{[j_a]} = u_a \wedge {V_a}^{[j_a+1]} = v_a \\ \wedge\ &{V_b}^{[j_b]} = u_b \wedge {V_b}^{[j_b+1]} = v_b \\ \wedge\ &\varphi^{mm}(a, b, (u_a, v_a)_M, (u_b, v_b)_M, \\ &\qquad {\hat{T}_a}^{[j_a]} + {\hat{w}_a}^{[j_a]}, {\hat{T}_b}^{[j_b]} + {\hat{w}_b}^{[j_b]}, \\ &\qquad {T_a}^{[j_a]} + {w_a}^{[j_a]}, {T_b}^{[j_b]} + {w_b}^{[j_b]}). \end{aligned}$$

This means that there are 6 possibilities how to resolve such a conflict (changing one of the four vertices of edges along which the two move actions took place or changing one of the two starting times of the move actions).

Now we also discuss conflicts where a waiting agent participates.

**Collisions While Waiting.** We also have to ensure that no collisions happen while an agent $a$ is waiting. In principle, the motion function $\alpha_{Ma}$ can also be constant, and hence one might be tempted to just specialize the formula for two moving agents to this case. However, unlike move actions, wait actions do not have fixed durations, but their duration is a consequence of the timing of the previous and following move action. We take this into account, generalizing the given conflict over arbitrarily long wait actions.

So assume an agent $a$ waiting at a point $x_a \in M$ and an agent $b$ following motion function $\alpha_{Mb}$ starting from time $\hat{\tau}_b$. Assume that a collision happens at a certain point in time $\hat{t}$. So we have $\hat{\tau}_b \leq \hat{t} \leq \hat{\tau}_b + \alpha_{Db} \wedge \textsc{IsCollision}(a, b, x_a, \alpha_{Mb}(\hat{t} - \hat{\tau}_b))$.

Let $\hat{\tau}_a$ be the end of the move action of the waiting agent $a$ before this waiting period and let $\hat{\tau}'_a$ be the starting time of the move action of the waiting agent after this waiting period. We illustrate the relationship of the variables in Figure 9.1.

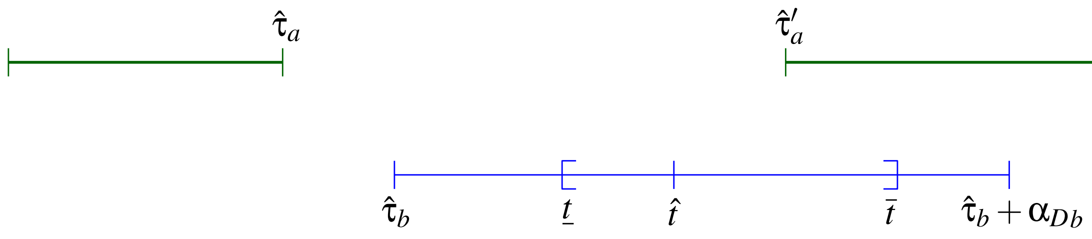


Figure 9.1: Illustration of action times and durations of an agent $a$ waiting on an agent $b$.

So we compute the beginning of the collision $\underline{t}_{a,b,x_a,\alpha_{Mb},\hat{\tau}_b} :=$

$$\inf\{\hat{\tau}_b \leq t \leq \hat{t} \mid \textsc{IsCollision}(a, b, x_a, \alpha_{Mb}(t - \hat{\tau}_b))\}$$

and its end $\overline{t}_{a,b,x_a,\alpha_{Mb},\hat{\tau}_b} :=$

$$\sup\{\hat{t} \leq t \leq \hat{\tau}_b + \alpha_{Db} \mid \textsc{IsCollision}(a, b, x_a, \alpha_{Mb}(t - \hat{\tau}_b))\}.$$

So for any wait action of agent $a$ starting at $\tau_a$ and ending at $\tau'_a$ and any move action of agent $b$ starting at $\tau_b$, the collision happens if the upper bound $\overline{t}_{a,b,x_a,\alpha_{Mb},\hat{\tau}_b}$ is after the end of the previous action and the lower bound $\underline{t}_{a,b,x_a,\alpha_{Mb},\hat{\tau}_b}$ is before the beginning of the next action. The result is

$$\begin{aligned}\tau_a - \tau_b &< \overline{t}_{a,b,x_a,\alpha_{Mb},\hat{\tau}_b} - \hat{\tau}_b \\ &\wedge \underline{t}_{a,b,x_a,\alpha_{Mb},\hat{\tau}_b} - \hat{\tau}_b < \tau'_a - \tau_b,\end{aligned}$$

which we will denote by $\varphi^{wm}(a, b, x_a, \alpha_{Mb}, \hat{\tau}_b, \tau_a, \tau'_a, \tau_b)$.

Now we again apply this to the variables of the SMT encoding described in Section 9.4, replacing the arguments of the formula $\varphi^{wm}(a, b, x_a, \alpha_{Mb}, \hat{\tau}_b, \tau_a, \tau'_a, \tau_b)$ by their corresponding terms from the SMT encoding. So when we detect a conflict between an agent $a$ that waits at vertex $u_a$ at time step $j_a$ and an agent $b$ moving along an edge

$(u_b, v_b)$ at time step $j_b$, starting at $\hat{T_b}^{[j_b]} + \hat{w_b}^{[j_b]}$, the formula $\varphi_{c_{new}}$ has the form

$$\begin{aligned}
&{V_a}^{[j_a]} = u_a \\
&\wedge {V_b}^{[j_b]} = u_b \wedge {V_b}^{[j_b+1]} = v_b \\
&\wedge \varphi^{wm}(a, b, \mathit{coord}(u_a), (u_b, v_b)_M, \\
&\qquad \hat{T_b}^{[j_b]} + \hat{w_b}^{[j_b]}, \\
&\qquad {T_a}^{[j_a]}, {T_a}^{[j_a]} + {w_a}^{[j_a]}, {T_b}^{[j_b]} + {w_b}^{[j_b]}).
\end{aligned}$$

We ended up with a conflict clause that actually does not depend on a previous move action. In the case where the wait action does not have a next move action, the conflict clause can be modified in a straightforward way.

**Further Generalization.** To fully exploit the computational effort that is necessary to find pre-plans, we generate generalizations for all conflicts a found pre-plan contains. Hence we check for conflicts between all pairs of agents, discrete steps and corresponding move and wait actions.

We further generalize the conflicts such that we also check for other conflicts of the given pair of agents when taking a move action from the same source to a different target vertex.

We did not find it useful to furthermore generalize the conflicts to further pairs of agents and/or discrete steps.

## 9.6 Implementation

**Collision Detection and Avoidance.** We implemented the predicates and functions introduced in Section 9.5 as follows:

- We assume that each agent $a$ is abstracted into an open disk with a fixed radius $r_a \in \mathbb{R}^{>0}$. Hence for agents $a$ and $b$, $\textsc{IsCollision}(a, b, c_a, c_b)$ iff $||c_a - c_b|| < r_a + r_b$, where $c_a, c_b \in M$ are respective centers of the disks of the agents.

- We assume that the agents move with constant velocity following straight lines of the edges. As a result, $\textsc{InConflict}_{a,b}$ corresponds to checking whether a quadratic inequality has a solution in the intersection of the time intervals from Definition 28.

- Exploiting the observation that $\textsc{Safe}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \hat{\tau}_a, \hat{\tau}_b) \in (\hat{\tau}_a, \hat{\tau}_b + \alpha_{Db}]$ we compute the resulting value by binary search of the switching point $\tau_a$ in the interval for which $\neg\textsc{InConflict}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \tau_a, \hat{\tau}_b)$ and $\textsc{InConflict}_{a,b}(\alpha_{Ma}, \alpha_{Mb}, \tau_a - \epsilon, \hat{\tau}_b)$, for a small enough $\epsilon > 0$.

**Floating-point Numbers.** Our simulations of collisions of agents are based on floating-point computation whereas SMT solvers treat linear real arithmetic precisely, using rational numbers for all computation. There are two main issues here:

- Conversion of a floating point number to a rational number may result in huge integer values for the numerator and denominator, although the intended value is very close to a simple rational or even integer number.

- Conversion of rational numbers to floating point numbers, and the following computation in floating point representation may incur approximation errors (e.g., due to round-off or discretization). For example, this may lead to the situation where—in the case of a collision between two moving agents—the added conflict clause does not require the waiting agent to wait long enough to completely avoid the same collision. Hence a very close collision may re-appear, and the same situation may repeat itself several times.

We overcome these deficiencies with an overapproximation of the conflict intervals along with simplification of the resulting rational numbers using simple continued fractions and best rational approximation: in the case that a floating-point value $x$ is respectively a lower or a higher bound of a conflicting interval, the result corresponds to the best rational approximation from $(x - \epsilon, x]$ or $[x, x + \epsilon)$, respectively, for an $\epsilon$ that is large enough. This not only avoids the re-appearance of the same conflict but also maps floating-point values that are close to each other to the same rational numbers, avoiding the appearance of tiny differences between rational numbers that tend to clog the SMT solver.

**Heuristics.** The formula passed to the SMT solver often represents a highly underconstrained problem, spanning a huge solution space. Due to this, it is essential that the SMT solver chooses a solution in a goal oriented way in order to maximize the chances of hitting upon a $\delta$-optimal plan, or at least to concentrate search on the most promising part of the solution space which also concentrates the addition of conflict avoidance clauses to this part. For this, we prefer transitions to vertices that lie on shorter paths to the goal over transitions to vertices that lie on longer paths. This can be easily precomputed using Dijkstra's algorithm for all vertices with a fixed start and goal.

Nonetheless, using such heuristics does not evade the problem of encoding all the transitions into the formula, which floods the SMT solver with a lot of constraints that are not essential for arriving at the desired plan. Also, conversion of the resulting formula to CNF might be expensive.

**Tools.** We implemented the resulting algorithm on top of MathSAT5 [33] SMT solver. We incrementally build the formula described in Section 9.4 using API. However, since we do not even require the SMT solver itself to handle optimization, it is easy to replace the API calls to another SMT solver that handles QF_LRA. We also implemented a visualization tool of MAPF$_R$ problems. Our tools are available online and are open-source [70, 71].

## 9.7 Computational Experiments

We compare run-times of our implementation from Section 9.6 denoted as SMT-LRA and state-of-the-art tools CCBS and SMT-CCBS, both presented in [4], which define the MAPF$_R$ problem in a similar way. These tools search for optimal plans, which is more difficult than searching for sub-optimal plans, such as minstep $\delta$-optimal plans in our case. However, as discussed in the introduction, not only that the price of getting optimal plans may be too high, but the resulting plans may also be not an ideal fit in practice, due to possible requirements on flexible dispatchability of the plans, and due to the fact that the dynamics of the agents may not be modeled accurately. Based on these observations, we consider the comparisons to be practically reasonable.

There are also differences concerning the function that is being optimized which is sum of costs in the case of CCBS and makespan in the case of SMT-CCBS. We support both of these cost functions in the form of a parameter. While there are certainly instances where the choice of the cost function qualitatively matters, [4] showed that both the tools yield similar respective costs of the resulting plans within their benchmarks. Hence, comparing such tools with different objectives also makes sense.

In the following experiments, we will use a similar setup to [4], that is, similar to both of the presented state-of-the-art tools.

We will start with the description of the benchmarks. Finally, we will present and discuss the computational results of the performed experiments.

### 9.7.1 Description of Benchmarks

A benchmark is specified by a graph and a set of agents, each defined by a radius and a starting and goal vertex. The following benchmarks use the same radius for all agents, and hence we will not discuss radii anymore.

We did experiments with three classes of problems: `empty`, `roadmap` and `bottleneck`. Benchmarks `empty` and `roadmap` are adopted from [4] and correspond to MAPF maps from the Moving AI repository. Our `bottleneck` benchmark is an additional simple experiment that identifies a weakness of the state-of-the-art approaches. A more detailed description of the benchmarks follows below.

We did not include benchmarks with large graphs (i.e. with a high number of vertices or edges), because our current algorithm encodes the whole graph into the formula, as discussed within heuristics in Section 9.6.

**Empty Room.** This benchmark is based on a graph that represents an empty square room with $16 \times 16$ vertices—the result of grid approximations of MAPF maps from the Moving AI repository [4].

Interconnection of the vertices depend on a neighborhood parameter $n$, which defines that each vertex has exactly $2^n$ neighbors (with the exception of boundary vertices). For example, $n = 2$ corresponds to a square grid, $n = 3$ extends the square grid of diagonal edges, etc. Using such a graph, it may be necessary to include a high amount of useless edges in order to cover a suitable number of realistic movements of the agents. On the other hand, it might be possible to exploit the fact that such graphs are highly symmetric.

The resulting benchmark `empty` represents a model with no physical obstacles. Still, with an increasing number of agents $k$, the number of possible collisions of the agents grows significantly, because most of the shortest paths lead via central regions of the graph.

**Roadmap.** Unlike the previous benchmark, here the maps from the Moving AI repository are not approximated based on grids but based on the "roadmap-generation tool from the Open Motion Planning Library (OMPL), which is a widely used tool in the robotics community". Such an approximation results in asymmetric graphs with possibly very different lengths of edges. On one hand, such edges can model realistic route choices of the agents. On the other hand, the number of possible places where agents are allowed to wait in order to avoid collisions may be too low, if the edges are too long—since we only allow waiting at vertices.

We follow the original benchmarks where the roadmap generation was applied on a large map `den520d` which comes from the field of video games. It is possible to set various levels of discretization (i.e. density) of the original map. Here, we only experimented with benchmarks with the lowest density, denoted as `sparse`.

**Bottleneck.** Benchmark `bottleneck` models the problem of steering $k$ agents from $k$ initial vertices through a single transfer vertex to $k$ goal vertices. Hence the transfer vertex represents a bottleneck every agent has to pass through. We place the initial and goal vertices (i.e., altogether $2k$ vertices) on a circle whose center is formed by the transfer vertex. An example of the benchmark with $k = 4$ is illustrated in the Figure 9.2, where the bigger colored disks denote the agents at their starting positions and the smaller disks denote the vertices of the graph, where the colored ones in addition indicate goal positions of the corresponding agents.

The task here boils down to just choosing a certain order of the agents. Resolving such a benchmark problem can still result in an exponential complexity in the number of agents $k$—if just various permutations of the agents are tried, without a thorough exclusion of the conflicting time intervals of particular agents.

### 9.7.2 Experimental Setup

In the case of benchmarks `empty` and `roadmap`, we observe whether particular experiments finish within a given timeout. The set of experiments contains instances where

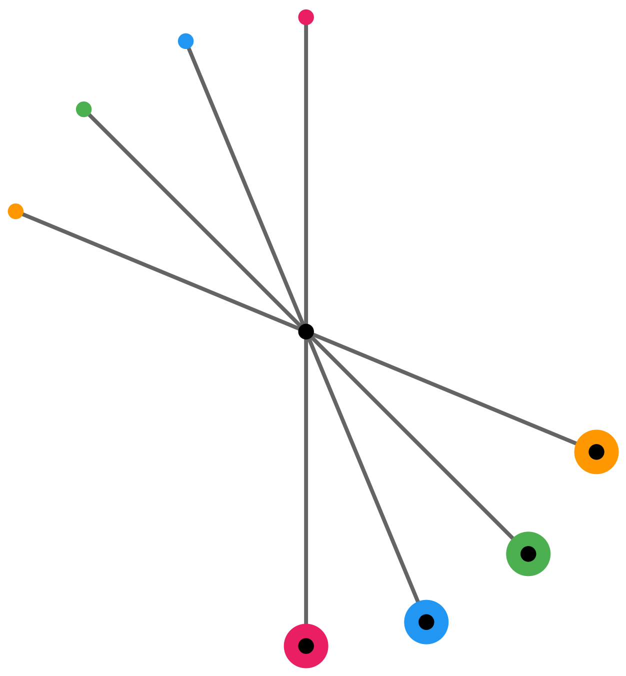


Figure 9.2: Bottleneck benchmark with $k = 4$ agents.

the number of agents ranges from 1 to 64 (none of the tools managed to finish with more agents within the selected timeouts). For each number of agents $k \in \{1, \ldots, 64\}$, each start and goal vertex of each agent is pre-generated in 25 random variants. Here, when generating the variants for agent $k+1$, all the previous $k$ agents are reused and only the positions of the new one are generated randomly. The result is $64 \times 25 = 1600$ instances for each of `empty` (for each neighborhood $n$) and `roadmap`.

The subject of our interest is how the evaluated algorithms scale with time, so we ran all the experiments with different timeouts ranging from 30 seconds up to 16 minutes (with exponential growth) and observed how many instances finished in time. We will show the results in the form of box plots.

To also directly illustrate the relationship of the number of solved instances and $k$, we will also show success rate plots, that is, plots with the ratios of the number of solved instances out of the total number of instances (i.e. out of 25) wrt. a given number of agents $k$, and with a fixed timeout.

Our tool SMT-LRA is in addition parametrized by a cost function—either makespan or sum of costs—and by a sub-optimal coefficient $\delta \in \{1, \frac{1}{2}, \frac{1}{4}\}$. In the plots, the parameters are denoted in the form $(C, \delta)$, where $C$ is either `M` (makespan) or `S` (sum of costs). In all experiments, the higher $\delta$ was, the more instances were solved. Hence, to make the box plots more compact, we merged all the variants of $\delta$ related to the same cost function such that the boxes of the variants with lower $\delta$ overlay the boxes of the variants with higher $\delta$. Also, higher values of $\delta$ correspond to lighter colors. For example, boxes $\delta = \frac{1}{4}$ overlay boxes $\delta = \frac{1}{2}$, but the magnitudes of both boxes are still visible since the number of solved instances is always lower in the case of $\delta = \frac{1}{4}$ than in the case of $\delta = \frac{1}{2}$. As a result, for each timeout in the box plot, our tool always takes two columns, each consisting of three (overlaid) boxes. In the case of success rate plots, we use dashed curves in the case of our tool in order to increase readability, and include all curves that correspond to the variants of $\delta$, where again higher values of $\delta$ correspond to lighter colors.

We will provide tables to further illustrate the effect of parameter $\delta$. For this, observe that Algorithm 9.1 terminates as soon as $cost(p_{opt}) \leq (1+\delta)t_{min}$, which ensures $\delta$-optimality. However, the ratio $cost(p_{opt})/t_{min}$, that we call *guaranteed ratio*, may actually be well below the required value $1+\delta$, meaning that the algorithm produced a better plan than required. The tables contain the average of the guaranteed ratio of plans that finished within 16 min (with lower timeouts, the ratios are even lower).

In the case of benchmark `bottleneck`, we only focus on the runtime of the evaluated tools for some numbers of agents ranging from 2 to 30. In the case of our implementation, we again include all the variants of parameters mentioned above. We used timeout 30 min to set some upper boundary on the runtimes of the tools.

We executed all the benchmarks on a machine with Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz, with 180 cores and 1TB memory. To unify the runtime environment, we reused and adapted the scripts from the previous experiments [4] which are a part of the SMT-CCBS tool. These scripts do not exploit all the available resources of the machine, though. Still, none of the evaluated tools use parallel computation—the cores are only used to run multiple benchmarks concurrently.

### 9.7.3 Results

**Empty Room.** Recall that benchmark `empty` is parametrized by its neighborhood $n$ which means that vertices have approximately $2^n$ neighbors. We did experiments with $n \in \{2,3,4,5\}$, all of which are shown in box plots in Figure 9.3. We also provide Table 9.1 with guaranteed ratios of the resulting plans, depending on $n$.

We first focus on the comparison of the selected cost functions in the case of our tool. We see that usually the cases that optimize the sum of costs perform better than the cases optimizing makespan. Observe, though, that the results are similar in the case of $\delta = \frac{1}{4}$ which correspond to the boxes at the base. We explain these as that in the case of the sum of costs there are more possibilities for how to reduce the cost of the plan than in the case of makespan where the cost usually depends on just one agent, regarding the symmetry of the graph. We assume that at the same time, this is the reason why, in the case of makespan, there are lower increases in the number of solved instances with growing $\delta$ compared to the variant which optimizes the sum of costs. Also, notice that in the cases of neighborhood $n = 2$ and especially $n = 5$, there are quite low performance growths with increasing $\delta$, which may be caused by the fact that

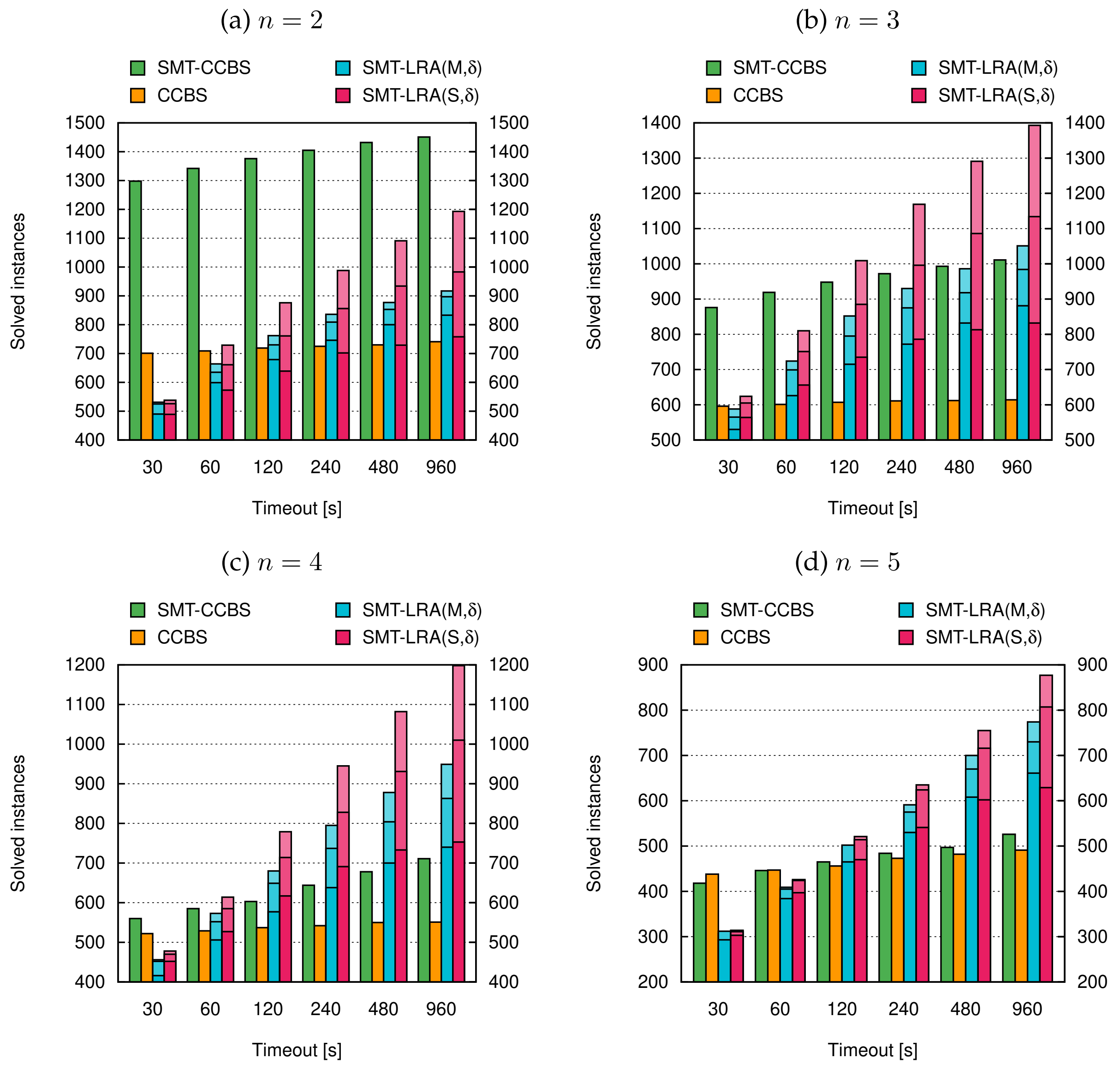


Figure 9.3: Comparison of the number of solved instances of benchmark `empty` with given $n$ wrt. a given timeout.

the actual guaranteed ratio is lower than in other cases of $n$, namely with $\delta = 1$ (see Table 9.1). Furthermore, [4] showed that these corner cases of $n$ are actually the least useful benchmarks: benchmarks with $n = 3$ offer much faster plans than in the case of $n = 2$, and $n = 5$ on the other hand provide only very low improvement over the case of $n = 4$. All in all, our approach scales well with the growing timeouts, in every case of neighborhood $n$, cost function and parameter $\delta$.

Now we also focus on the state-of-the-art tools, where we will actually confirm the observations made in [4]. These tools are consistent in the sense that the lower parame-

Table 9.1: Average guaranteed ratios of SMT-LRA in benchmark `empty` wrt. the parameters of the tool and a given $n$.

| $(C, \delta) \setminus n$ | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| $(\mathtt{M}, 1)$ | 1.45 | 1.55 | 1.54 | 1.43 |
| $(\mathtt{S}, 1)$ | 1.56 | 1.57 | 1.56 | 1.40 |
| $(\mathtt{M}, 0.5)$ | 1.29 | 1.33 | 1.32 | 1.29 |
| $(\mathtt{S}, 0.5)$ | 1.30 | 1.30 | 1.31 | 1.26 |
| $(\mathtt{M}, 0.25)$ | 1.16 | 1.18 | 1.18 | 1.17 |
| $(\mathtt{S}, 0.25)$ | 1.16 | 1.15 | 1.15 | 1.14 |

ter $n$, the faster their algorithm—because there are fewer possible paths to the goals. In our case, the observation holds as well, but with one exception in the case of $n = 2$ vs. $n = 3$, where the runtime of the experiments with the lowest neighborhood is higher. The reason is that the graphs with higher $n$ allow that the shortest paths to the goals take fewer edges—which in our case becomes more important than the number of possible choices, because our current algorithm is sensitive to the number of edges in the graph which we all encode into the formula.

The state-of-the-art tools usually perform better than our tool when the timeout is less or equal to 1 min. SMT-CCBS performs especially well in the case of $n = 2$ because it maps a lot of time points to the same values since many of them are integer values. We however consider this case to be the least useful benchmark referring to the earlier discussion and in addition since the square grids are not too realistic models and can also be handled by standard MAPF approaches (which are currently much faster than MAPF$_R$ approaches). Although SMT-CCBS scales better with time than CCBS, the highest growth of the number of solved instances still occurs in the case of our approach, even in the cases of $\delta = \frac{1}{4}$ which correspond to the boxes of our algorithm at the base. In the cases of $n \geq 3$ and $\delta = 1$, which correspond to the upmost boxes of our algorithm, we outperform the state-of-the-art tools if the timeout is high.

We also provide Figure 9.4 with success rate plots. Here we selected timeout 8 min (480 s). Recall that for our tool, we distinguish the curves that correspond to higher $\delta$ by lighter colors. Within a single cost function, especially in the case of the sum of costs, the distances between the curves of particular cases of $\delta$ seem to be quite uniform, which confirms the observations based on Figure 9.3 that in many cases our algorithm scales well with the parameter $\delta$. At the same time, the plots also confirm that sometimes in the case of makespan the performance does not increase much with growing $\delta$.

In the case of all the tools, especially in the case of our tool, there happen to be glitches in the success rates—sometimes the performance increases a bit with a higher number of agents. In some cases, it is probably just caused by inaccurate measurements, however approaches that are based on a SAT solver (SMT-CCBS) or even an SMT solver (SMT-LRA) may naturally exhibit such behavior since the algorithms are more complex

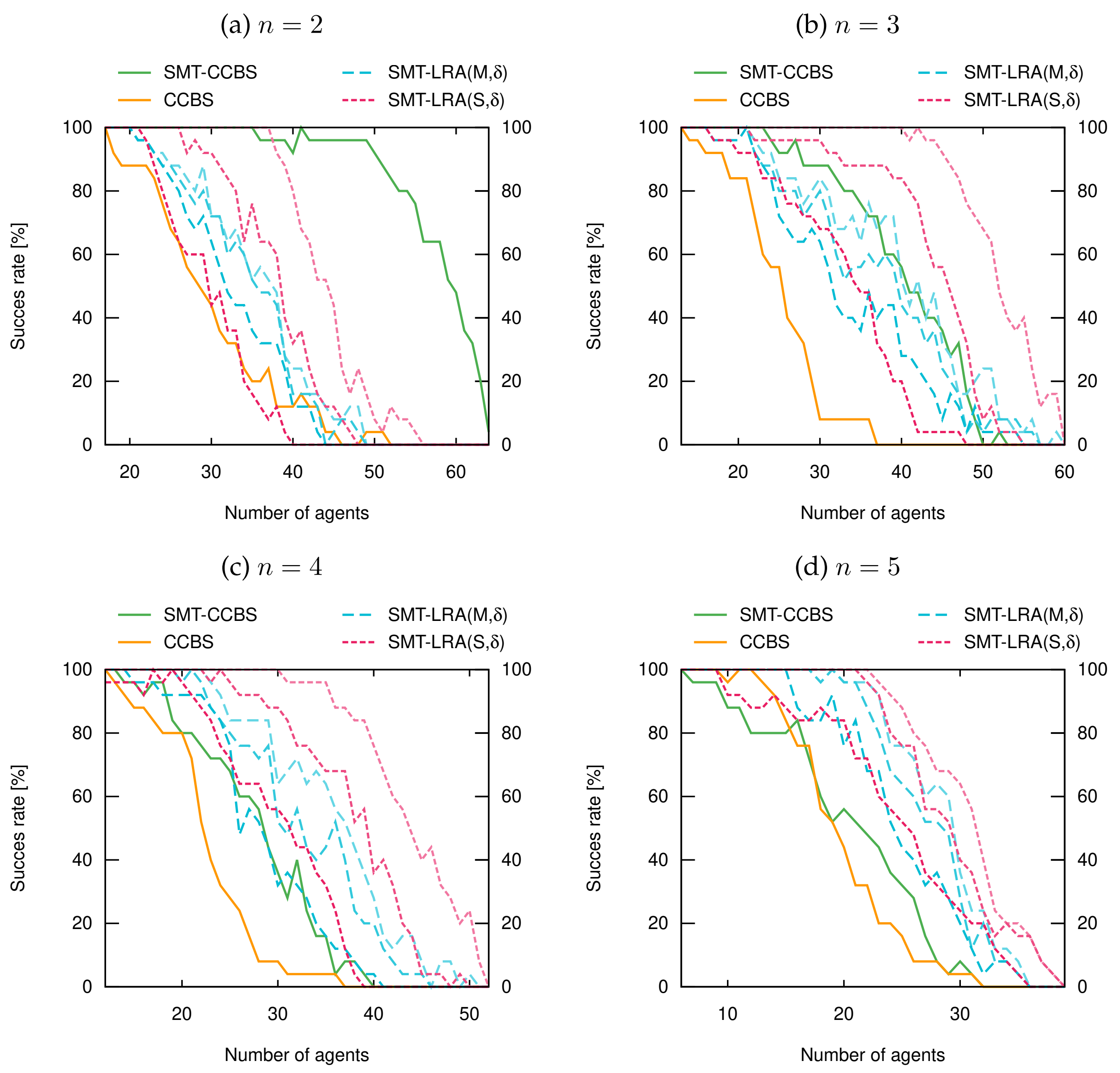


Figure 9.4: Comparison of the success rates of benchmark `empty` with given $n$ within timeout 8 min wrt. a number of agents.

and not that straightforward like CBS-based algorithms.

**Roadmap.** Figure 9.5 shows how particular tools scale with time in the case of benchmark `roadmap`. Clearly, SMT-CCBS does not handle this benchmark well, independent of the chosen timeout. By contrast, CCBS performs very well, especially with smaller timeouts. However, at the timeout of 4 min (240 s) it reaches a point after which it almost stops scaling with time at all. In our case, the variants that optimize the sum of costs or makespan perform almost the same. However, the performance depends a lot

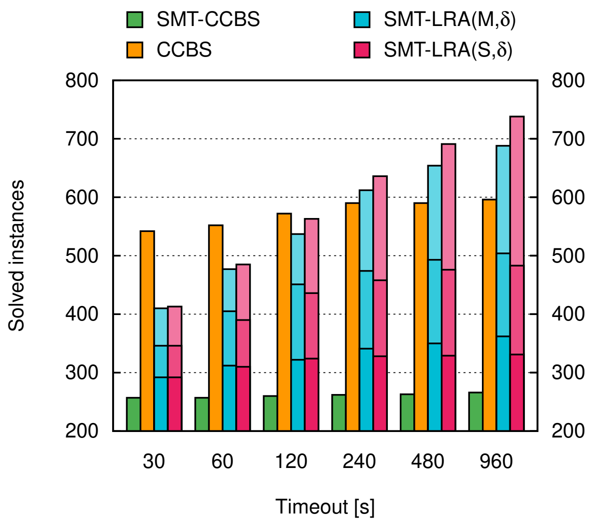


Figure 9.5: Comparison of the number of solved instances of benchmark `roadmap` wrt. a given timeout.

on parameter $\delta$. For example, with $\delta = 1$, which corresponds to the upmost boxes, our approach scales very well and outperforms CCBS for timeouts greater or equal to 4 min. However, in the cases of $\delta = \frac{1}{4}$ (i.e., the boxes at the base), the algorithm is not competitive with CCBS and scales poorly. We explain this as follows: the `roadmap` graph is highly asymmetric and contains a lot of long edges, compared to the graphs in benchmark `empty`. Therefore, the shortest paths to the goals often consist of a low number of edges. At the same time, paths to the goals with similar distances can actually consist of a different number of edges. Thus, once we find a (collision-free) plan and fix the number of steps $h$ for all agents, it may happen that when optimizing the plan, we miss alternative paths that consist of more steps which could be essential to arriving at easier possibilities of finding faster plans.

In addition, we provide Table 9.2 with the guaranteed ratios of our tool. The ratios are quite high in the cases of $\delta = 1$, which can explain why the difference in the number of solved instances is so high compared to $\delta = \frac{1}{2}$, and also compared to benchmark `empty`, where the guaranteed ratios are lower.

Similarly to benchmark `empty`, we also provide a plot with success rates of the tools, in Figure 9.6, again with the timeout of 8 min. Here the success rates are well distributed with no anomalies.

**Bottleneck.** We summarize the runtimes of benchmark `bottleneck` of particular tools in Table 9.3. In the case of SMT-CCBS, we excluded the built-in verification of the solutions which here seemed to be very time-consuming. In our case, we merge all the corresponding cases of parameter $\delta$ into a single column since the runtimes were almost the same regardless of the parameter. We further merge both cost functions `M`

Table 9.2: Average guaranteed ratios of SMT-LRA in benchmark `roadmap` wrt. the parameters of the tool.

| $(C, \delta)$ | |
|---|---|
| $(\mathtt{M}, 1)$ | 1.64 |
| $(\mathtt{S}, 1)$ | 1.65 |
| $(\mathtt{M}, 0.5)$ | 1.31 |
| $(\mathtt{S}, 0.5)$ | 1.31 |
| $(\mathtt{M}, 0.25)$ | 1.14 |
| $(\mathtt{S}, 0.25)$ | 1.14 |

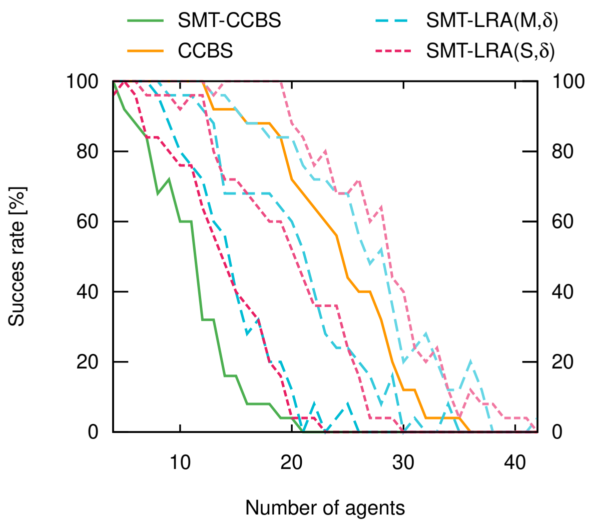


Figure 9.6: Comparison of the success rates of benchmark `roadmap` within timeout 8 min wrt. a number of agents.

and S into one column s.t. the respective runtimes are separated by a pipe character. The average guaranteed ratio of our presented plans is 1.08 in the case of makespan (M) and 1.04 in the case of the sum of costs (S).

It is clear that the runtimes of both state-of-the-art solvers exhibit an exponential relationship with the number of agents $k$, while our algorithm is much less sensitive. For example, CCBS is fastest until $k = 6$ but after that point our SMT-LRA dominates the runtime. The reason is that we resolve the conflicts of agents using the learning mechanism of generalized conflict clauses where the timing constraints efficiently exclude inappropriate orderings of the agents, making the benchmark fairly easy for our approach—which is consistent with the observation that such a problem is indeed trivial, as discussed in the description of benchmarks. For example, the problem is easily

Table 9.3: Comparison of the runtimes in seconds of benchmark `bottleneck` with a given number of agents.

| $k$ | SMT-CCBS | CCBS | SMT-LRA(`M`\|`S`) |
|---|---|---|---|
| 2 | 0.00 | 0.00 | 0.01 \| 0.01 |
| 3 | 0.47 | 0.00 | 0.02 \| 0.02 |
| 4 | $> 1800$ | 0.00 | 0.04 \| 0.02 |
| 5 | ? | 0.01 | 0.03 \| 0.03 |
| 6 | ? | 0.05 | 0.07 \| 0.05 |
| 7 | ? | 0.45 | 0.08 \| 0.06 |
| 8 | ? | 3.52 | 0.15 \| 0.08 |
| 9 | ? | 43.99 | 0.15 \| 0.11 |
| 10 | ? | 720.27 | 0.22 \| 0.14 |
| 11 | ? | $> 1800$ | 0.26 \| 0.17 |
| 15 | ? | ? | 0.46 \| 0.42 |
| 20 | ? | ? | 0.79 \| 0.97 |
| 30 | ? | ? | 4.88 \| 5.63 |

solvable using an ad-hoc approach. Nevertheless, such bottlenecks may appear as a part of more complex problems where a sophisticated algorithm instead of an ad-hoc should be used.

**Profiling.** Profiling of our tool showed that the simulations used for collision detection and avoidance take a negligible part of the runtime. Instead, most of the time is spent in the SMT solver itself. If our approach was applied to benchmarks with large graphs (as discussed above), then also encoding the formula, conversion to CNF, etc., would take an additional important part of the runtime.

## 9.8 Conclusion

We have demonstrated how to solve the continuous-time MAPF problem (MAPF$_{\mathbb{R}}$) by direct translation to SAT modulo linear real arithmetic. While the approach insists only on sub-optimality up to a certain factor, it shows several advantages over state-of-the-art algorithms, especially better scaling wrt. an increasing time budget and its ability to efficiently handle bottleneck situations. Our approach also allows for easy change of the objective function to another. The downside is a certain basic translation effort, especially for problems depending on large graphs.

In future work, we will explore a lazy approach to translation that only generates the information necessary for solving the current problem instance. This will be especially relevant in practical applications where similar problems have to be solved repeatedly.

Moreover, we will generalize the method to problems with non-linear motion functions, allowing both non-linear geometry of the involved curves and the modeling of non-linear dynamical phenomena such as acceleration of agents. The method will also benefit from the fact that the efficiency of SMT solvers is currently improving with each year.

Chapter 10

# Conclusion

We introduced a new method for checking satisfiability of logical formulas that involve ordinary differential equations (ODEs), which we handle using the same semantics as in simulation tools, motivated by industrial practice. In Chapter 4, we defined a logical theory and the resulting problem, Boolean Satisfiability Modulo Differential Equation Simulations. In Chapter 6, we designed a corresponding solver that aims at deciding strong satisfiability of a formula of our theory. Our implementation of the solver, UN/-SOT, tightly integrates handling of Boolean and floating-point constraints, including ODEs.

In Chapter 7, we showed several interesting case studies with models that involve ODEs encoded into logical formulas. The models are followed by experimental results of the case studies where we used our implementation of the solver. For some of the models, we also compared the results with a state-of-the-art SAT modulo ODE solver, which handles differential equations based on classical mathematical semantics. The experiments showed promising results of our tool, where we solve the corresponding problems efficiently and much faster than the state-of-the-art tool, especially in the cases when the formula is satisfiable.

Importantly, Chapter 8 introduced an example of a planning problem. Unlike the benchmarks used in state-of-the-art approaches and in Chapter 7, the benchmark problem exhibits both non-trivial discrete and continuous phenomena. The resulting problem comes from the domain of railway scheduling, where we simulate train networks at a low level and where a number of timing and ordering constraints can appear. The experiments showed that our solver is competitive with methods based on dedicated railway simulators while being more general and extensible.

Finally, in Chapter 9 we also presented a new approach to solving a continuous-time version of the multi-agent path-finding (MAPF) problem, MAPF$_R$. In the approach, we exploit conflict generalization techniques that stem from linear real arithmetic constraints. In addition to this, collision detection and avoidance of the agents yields non-linear constraints which we handle based on simulations (i.e., floating-point computation). The simulations do not involve differential equations though and we handle them using an off-the-shelf SMT solver instead of our implementation UN/SOT. Still, we designed the model such that it allows to increase the complexity of the simulations, for example with ODEs. Computational experiments showed that our approach scales better wrt. the available computation time than state-of-the-art approaches and is usually able to avoid their exponential behavior on a class of benchmark problems modeling a typical bottleneck situation.

## 10.1 Future Work

In the future, we intend to work:

- On search heuristics for more efficient handling of satisfiable inputs, for example, based on data-driven approaches such as machine learning.
- On deduction techniques that would improve the learning mechanism of our solver and proving the unsatisfiability of formulas. A possible way is to handle interval constraints with an interval arithmetic, such as affine arithmetic.
- On a more general handling of final conditions that would allow checking satisfiability of a broader set of formulas, and not only strong satisfiability. Using an interval arithmetic is one of the possibilities how to support this.
- On a replacement of the synchronous model of the railway scheduling problem to an asynchronous model. Also, we may allow more nondeterminism in the model.
- On a lazy approach to translation of a MAPF$_R$ problem instance that only generates the information necessary for solving the current instance. This will be especially relevant in practical applications where similar problems have to be solved repeatedly.
- On a generalization of our MAPF$_R$ method to problems with non-linear motion functions, allowing both non-linear geometry of the involved curves and the modeling of non-linear dynamical phenomena such as acceleration of agents.

# Bibliography

[1] K. Ahnert and M. Mulansky. "Odeint – Solving Ordinary Differential Equations in C++". In: *AIP Conf. Proc. 1389* (2011), pp. 1586–1589. URL: https://doi.org/10.1063/1.3637934.

[2] Thomas Albrecht. "Railway timetable and Traffic". In: *Eurailpress: Hamburg, Germany* (2008).

[3] Matthias Althoff. "An Introduction to CORA 2015". In: *Proc. of the 1st and 2nd Workshop on Applied Verification for Continuous and Hybrid Systems*. EasyChair, Dec. 2015, pp. 120–151. URL: https://easychair.org/publications/paper/xMm.

[4] Anton Andreychuk, Konstantin Yakovlev, et al. "Multi-agent pathfinding with continuous time". In: *Artificial Intelligence* 305 (2022), p. 103662. ISSN: 0004-3702. URL: https://doi.org/10.1016/j.artint.2022.103662.

[5] Leni Aniva, Haniel Barbosa, and Clark Barrett. *CVC5*. Stanford University and the University of Iowa, 2023. URL: https://cvc5.github.io/.

[6] Kendall Atkinson, Weimin Han, et al. *Numerical Solution of Ordinary Differential Equations*. John Wiley & Sons, Inc., Feb. 2009, p. 272. ISBN: 978-0-470-04294-6.

[7] Dor Atzmon, Roni Stern, et al. "Probabilistic Robust Multi-Agent Path Finding". In: *Proc. of the Thirtieth International Conference on Automated Planning and Scheduling, 2020*. AAAI Press, 2020, pp. 29–37.

[8] Kyungmin Bae, Soonho Kong, and Sicun Gao. "SMT Encoding of Hybrid Systems in dReal". In: *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*. Ed. by Goran Frehse and Matthias Althoff. Vol. 34. EPiC Series in Computing. EasyChair, 2015, pp. 188–195. URL: https://doi.org/10.29007/s3b9.

[9] Stanley Bak and Parasara Sridhar Duggirala. "HyLAA: A Tool for Computing Simulation-Equivalent Reachability for Linear Systems". In: *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*. HSCC '17. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2017, pp. 173–178. ISBN: 9781450345903. URL: https://doi.org/10.1145/3049797.3049808.

[10] Haniel Barbosa, Clark Barrett, et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9.

[11] Clark Barrett, Morgan Deters, et al. "6 Years of SMT-COMP". In: *Journal of Automated Reasoning* 50.3 (2013), pp. 243–277. URL: https://doi.org/10.1007/s10817-012-9246-5.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. 2021. URL: http://www.SMT-LIB.org.

[13] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. 2021. URL: https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-04-02.pdf.

[14] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Vol. 10. Springer, 2018.

[15] Armin Biere. "Bounded Model Checking". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, et al. 2nd. IOS Press, 2009. Chap. 14, pp. 457–481. URL: https://doi.org/10.3233/978-1-58603-929-5-457.

[16] Armin Biere, Marijn Heule, et al., eds. *Handbook of Satisfiability*. 2nd. IOS Press, 2021.

[17] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "$\nu$Z - An Optimizing SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 194–199. ISBN: 978-3-662-46681-0.

[18] Miquel Bofill, Robert Nieuwenhuis, et al. "The Barcelogic SMT Solver". In: *Computer Aided Verification*. Ed. by Aarti Gupta and Sharad Malik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 294–298. ISBN: 978-3-540-70545-1.

[19] Aysu Bogatarkan, Volkan Patoglu, and Esra Erdem. "A Declarative Method for Dynamic Multi-Agent Path Finding". In: *GCAI 2019. Proc. of the 5th Global Conference on Artificial Intelligence*. Vol. 65. EPiC Series in Computing. EasyChair, 2019, pp. 54–67.

[20] Sergiy Bogomolov, Marcelo Forets, et al. "JuliaReach: A Toolbox for Set-Based Reachability". In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. HSCC '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 39–44. ISBN: 9781450362825. URL: https://doi.org/10.1145/3302504.3311804.

[21] Olivier Bouissou, Samuel Mimram, and Alexandre Chapoutot. "Hyson: Set-based simulation of hybrid systems". In: *23rd IEEE International Symposium on Rapid System Prototyping (RSP)*. IEEE. 2012, pp. 79–85.

[22] Olivier Bournez and Manuel L. Campagnolo. "A Survey on Continuous Time Computations". In: *New Computational Paradigms*. Ed. by S.Barry Cooper, Benedikt Löwe, and Andrea Sorbi. Springer New York, 2008, pp. 383–423. ISBN: 978-0-387-68546-5. URL: https://doi.org/10.1007/978-0-387-68546-5_17.

[23] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation. Decision Procedures with Applications to Verification*. Springer-Verlag Berlin Heidelberg, 2007, p. 366. ISBN: 978-3-540-74112-1.

[24] Martin Brain, Cesare Tinelli, et al. "An automatable formal semantics for IEEE-754 floating-point arithmetic". In: *22nd IEEE Symposium on Computer Arithmetic*. IEEE. 2015, pp. 160–167. URL: https://doi.org/10.1109/ARITH.2015.26.

[25] D. Bresolin, P. Collins, et al. "A computable and compositional semantics for hybrid automata". In: *7th Int. Wireless Communications and Mobile Computing Conf. (IWCMC)*. Apr. 2020. URL: https://doi.org/10.1145/3365365.3382202.

[26] R. Bruttomesso, E. Pek, et al. "The OpenSMT Solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 6015. Springer. Paphos, Cyprus: Springer, 2010, pp. 150–153. URL: https://doi.org/10.1007/978-3-642-12002-2_12.

[27] Roberto Bruttomesso and Antti Hyvarinen. *OpenSMT*. University of Lugano, 2023. URL: https://verify.inf.usi.ch/opensmt.

[28] Shaowei Cai. *Z3++*. Chinese Academy of Sciences, 2022. URL: https://z3-plus-plus.github.io/.

[29] Shaowei Cai, Bohan Li, and Xindi Zhang. "Local Search for SMT on Linear Integer Arithmetic". In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer International Publishing, 2022, pp. 227–248. ISBN: 978-3-031-13188-2.

[30] Michael Cashmore, Daniele Magazzeni, and Parisa Zehtabi. "Planning for hybrid systems via satisfiability modulo theories". In: *Journal of Artificial Intelligence Research* 67 (2020), pp. 235–283.

[31] Sanjian Chen, Matthew O'Kelly, et al. "An intraoperative glucose control benchmark for formal verification". In: *IFAC-PapersOnLine* 48.27 (2015), pp. 211–217.

[32] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. "Flow*: An Analyzer for Non-linear Hybrid Systems". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 258–263. ISBN: 978-3-642-39799-8.

[33] Alessandro Cimatti, Alberto Griggio, et al. "The MathSAT5 SMT Solver". In: *Proceedings of TACAS*. Ed. by Nir Piterman and Scott Smolka. Vol. 7795. LNCS. Springer, 2013.

[34] Liron Cohen, Tansel Uras, et al. "Optimal and Bounded-Suboptimal Multi-Agent Motion Planning". In: *Proc. of the Twelfth International Symposium on Combinatorial Search, SOCS 2019*. AAAI Press, 2019, pp. 44–51.

[35] David R. Cok. *The SMT-LIBv2 Language and Tools: A Tutorial*. GrammaTech, Inc., 2013. URL: http://www.grammatech.com/resource/smt/SMTLIBTutorial.pdf.

[36] Francesco Contaldo, Patrick Trentin, and Roberto Sebastiani. "From MINIZINC to Optimization Modulo Theories, and Back". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Ed. by Emmanuel Hebrard and Nysret Musliu. Cham: Springer International Publishing, 2020, pp. 148–166. ISBN: 978-3-030-58942-4.

[37] Stephen A. Cook. "The Complexity of Theorem-proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. URL: http://doi.acm.org/10.1145/800157.805047.

[39] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, Apr. 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0. URL: https://doi.org/10.1007/978-3-540-78800-3_24.

[40] Julien Alexandre dit Sandretto and Alexandre Chapoutot. "Validated Explicit and Implicit Runge–Kutta Methods". In: *Reliable Computing* 22.1 (July 2016), pp. 79–103. URL: https://hal.archives-ouvertes.fr/hal-01243053.

[41] Julien Alexandre dit Sandretto, Alexandre Chapoutot, and Olivier Mullier. *DynIbex*. 2023. URL: https://perso.ensta-paris.fr/~chapoutot/dynibex.

[42] Tommaso Dreossi, Thao Dang, and Carla Piazza. "Parallelotope Bundles for Polynomial Reachability". In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. HSCC '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 297–306. ISBN: 9781450339551. URL: https://doi.org/10.1145/2883817.2883838.

[43] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 737–744. ISBN: 978-3-319-08867-9.

[44] Bruno Dutertre, Dejan Jovanović, and Stéphane Graham-Lengrand. *The Yices SMT Solver*. SRI International, 2022. URL: https://yices.csl.sri.com/.

[45] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-24605-3.

[46] Niklas Eén and Niklas Sörensson. *MiniSAT*. 2008. URL: http://minisat.se.

[47] A. Eggers, N. Ramdani, et al. "Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods". In: *International Conference on Software Engineering and Formal Methods (SEFM)* 9 (2011). URL: https://doi.org/10.1007/978-3-642-24690-6_13.

[48] Chuchu Fan, Yu Meng, et al. "Verifying nonlinear analog and mixed-signal circuits with inputs". In: *IFAC-PapersOnLine* 51.16 (2018), pp. 241–246. ISSN: 2405-8963. URL: https://www.sciencedirect.com/science/article/pii/S2405896318311571.

[49] Chuchu Fan, Bolun Qi, et al. "Automatic Reachability Analysis for Nonlinear Hybrid Models with C2E2". In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 531–538. ISBN: 978-3-319-41528-4.

[50] Ariel Felner, Roni Stern, et al. "Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges". In: *Proc. of the Tenth International Symposium on Combinatorial Search, SOCS 2017*. AAAI Press, 2017, pp. 29–37.

[51] M. Fränzle, C. Herde, et al. "Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure". In: *JSAT—Journal on Satisfiability, Boolean Modeling and Computation, Special Issue on SAT/CP Integration* 1 (2007), pp. 209–236.

[52] Goran Frehse and Matthias Althoff, eds. *Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23)*. Vol. 96. EPiC Series in Computing. EasyChair, 2023.

[53] Goran Frehse, Colas Le Guernic, et al. "SpaceEx: Scalable Verification of Hybrid Systems". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 379–395. ISBN: 978-3-642-22110-1. URL: https://doi.org/10.1007/978-3-642-22110-1_30.

[54] Nathan Fulton, Stefan Mitsch, et al. “KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems”. In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 527–538. ISBN: 978-3-319-21401-6.

[55] Graeme Gange, Daniel Harabor, and Peter J. Stuckey. “Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation”. In: *Proc. of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019*. AAAI Press, 2019, pp. 155–162.

[56] Sicun Gao, Jeremy Avigad, and Edmund Clarke. “$\delta$-Complete Decision Procedures for Satisfiability over the Reals”. In: vol. 7364. Apr. 2012. URL: https://doi.org/10.1007/978-3-642-31365-3_23.

[57] Sicun Gao, Soonho Kong, and Edmund Clarke. “Satisfiability Modulo ODEs”. In: *Formal Methods in Computer-Aided Design (FMCAD)* (2013).

[58] Sicun Gao, Soonho Kong, and Edmund M. Clarke. “dReal: an SMT Solver for Nonlinear Theories over the Reals”. In: *Proceedings of the 24th International Conference on Automated Deduction*. CADE’13. Lake Placid, NY: Springer-Verlag, 2013, pp. 208–214. ISBN: 978-3-642-38573-5. URL: https://doi.org/10.1007/978-3-642-38574-2_14.

[59] Alexandre Goldsztejn, Olivier Mullier, et al. “Including Ordinary Differential Equations Based Constraints in the Standard CP Framework”. In: *Principles and Practice of Constraint Programming – CP 2010*. Ed. by David Cohen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 221–235. ISBN: 978-3-642-15396-9. URL: https://doi.org/10.1007/978-3-642-15396-9_20.

[60] Alberto Griggio, Alessandro Cimatti, and Roberto Sebastiani. *The MathSAT 5 SMT Solver*. Fondazione Bruno Kessler and DISI-University of Trento, 2023. URL: https://mathsat.fbk.eu/.

[61] Rebecca Haehn, Erika Ábrahám, and Nils Nießen. “Freight Train Scheduling in Railway Systems”. In: *Measurement, Modelling and Evaluation of Computing Systems*. Ed. by Holger Hermanns. Cham: Springer International Publishing, 2020, pp. 225–241. ISBN: 978-3-030-43024-5. URL: https://doi.org/10.1007/978-3-030-43024-5_14.

[62] Rebecca Haehn, Erika Ábrahám, and Nils Nießen. “Symbolic Simulation of Railway Timetables Under Consideration of Stochastic Dependencies”. In: *LNCS*. Vol. 12846. Springer, 2021, pp. 257–275. URL: https://doi.org/10.1007/978-3-030-85172-9_14.

[63] Ernst Hairer, Syvert Paul Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I*. Springer-Verlag Berlin Heidelberg, 1993. ISBN: 978-3-540-56670-0. URL: https://doi.org/10.1007/978-3-540-78862-1.

[64] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II*. Springer-Verlag Berlin Heidelberg New York, 1996. ISBN: 3-540-60452-9.

[65] Alan C Hindmarsh, Peter N Brown, et al. "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 363–396.

[66] Daisuke Ishii, Kazunori Ueda, and Hiroshi Hosobe. "An interval-based SAT modulo ODE solver for model checking nonlinear hybrid systems". In: *International Journal on Software Tools for Technology Transfer (STTT)* 13.5 (2011), pp. 449–461.

[67] Matti Järvisalo, Daniel Le Berre, et al. "The International SAT Solver Competitions". In: *AI Magazine* 33.1 (Mar. 2012), pp. 89–92. URL: https://doi.org/10.1609/aimag.v33i1.2395.

[68] Jekyll and Skinny Bones. *dReal*. 2021. URL: http://dreal.github.io.

[78] Soonho Kong, Sicun Gao, et al. "dReach: $\delta$-Reachability Analysis for Hybrid Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 200–205. ISBN: 978-3-662-46681-0. URL: https://doi.org/10.1007/978-3-662-46681-0_15.

[79] Edward Lam, Pierre Le Bodic, et al. "Branch-and-cut-and-price for multi-agent path finding". In: *Comput. Oper. Res.* 144 (2022), p. 105809.

[80] Adrien Le Coënt, Julien Alexandre dit Sandretto, et al. "An improved algorithm for the control synthesis of nonlinear sampled switched systems". In: *Formal Methods in System Design* 53.3 (2018), pp. 363–383. ISSN: 1572-8102. URL: https://doi.org/10.1007/s10703-017-0305-8.

[81] Francesco Leofante. "OMTPlan: a tool for optimal planning modulo theories". In: *Journal on Satisfiability, Boolean Modeling and Computation* 14.1 (2023), pp. 17–23.

[82] Jiaoyang Li, Zhe Chen, et al. "Anytime Multi-Agent Path Finding via Large Neighborhood Search". In: *Proc. of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021*. ijcai.org, 2021, pp. 4127–4135.

[83] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. "EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.14 (May 2021), pp. 12353–12362. URL: https://doi.org/10.1609/aaai.v35i14.17466.

[84] Yangge Li, Haoqing Zhu, et al. *Verse: A Python library for reasoning about multi-agent hybrid system scenarios*. 2023. URL: https://doi.org/10.48550/arXiv.2301.08714.

[85] Hai Lin and Panos J. Antsaklis. *Hybrid Dynamical Systems: Fundamentals and Methods*. Springer Cham, 2021. ISBN: 978-3-030-78729-5. URL: https://doi.org/10.1007/978-3-030-78731-8.

[86] Enrico Lipparini and Stefan Ratschan. “Satisfiability of Non-linear Transcendental Arithmetic as a Certificate Search Problem”. In: *NASA Formal Methods*. Ed. by Kristin Yvonne Rozier and Swarat Chaudhuri. Cham: Springer Nature Switzerland, 2023, pp. 472–488. ISBN: 978-3-031-33170-1.

[87] Bing Liu, Soonho Kong, et al. “Parameter identification using delta-decisions for biological hybrid systems”. In: (2014).

[88] R. J. Lohner. “Enclosing the Solutions of Ordinary Initial and Boundary Value Problems”. In: *Computer Arithmetic: Scientific Computation and Programming Languages*. Stuttgart: Teubner, 1987.

[89] Ryan Luna and Kostas E. Bekris. “Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees”. In: *IJCAI 2011, Proc. of the 22nd International Joint Conference on Artificial Intelligence, 2011*. IJCAI/AAAI, 2011, pp. 294–300.

[90] Jan Lunze, ed. *Handbook of Hybrid Systems Control: Theory, Tools, Applications*. Cambridge University Press, 2009. URL: https://doi.org/10.1017/CBO9780511807930.

[91] B. Luteberget, K. Claessen, and C. Johansen. “SAT modulo discrete event simulation applied to railway design capacity analysis”. In: *Formal Methods in System Design* 57 (2021), pp. 211–245. URL: https://doi.org/10.1007/s10703-021-00368-2.

[92] Hang Ma. “Intelligent Planning for Large-Scale Multi-Agent Systems”. In: *AI Mag.* 43.4 (2022), pp. 376–382.

[93] Guillaume Melquiond. “Floating-point arithmetic in the Coq system”. In: *Information and Computation* 216 (2012), pp. 14–23.

[94] M. Montigel. “Formal representation of track topologies by double vertex graphs”. In: *Proceedings of Railcomp 92 held in Washington DC, Computers in Railways 3*. Vol. 2. Computational Mechanics Publications, 1992.

[95] P. J. Mosterman. “An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages”. In: *Lecture Notes in Computer Science*. Vol. 1569. HSCC 1999. Berlin, Heidelberg: Springer-Verlag, 1999. URL: https://doi.org/10.1007/3-540-48983-5_17.

[96] Pieter J Mosterman, Justyna Zander, et al. “A computational model of time for stiff hybrid systems applied to control synthesis”. In: *Control Engineering Practice* 20.1 (2012), pp. 2–13.

[97] Leonardo de Moura, Nikolaj Bjørner, et al. *Z3 Theorem Prover*. Microsoft® Research, 2018. URL: https://github.com/Z3Prover/z3.

[98] Nedialko S Nedialkov. “Implementing a rigorous ODE solver through literate programming”. In: *Modeling, Design, and Simulation of Systems with Uncertainties*. Springer, 2011, pp. 3–19.

[99] Rober Nieuwenhuis and Albert Oliveras. “On SAT Modulo Theories and Optimization Problems”. In: *Theory and Applications of Satisfiability Testing*. Ed. by Armin Biere and Carla P. Gomes. SAT 2006. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 156–169. ISBN: 978-3-540-37207-3. URL: https://doi.org/10.1007/11814948_18.

[100] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann-Loveland procedure to DPLL(T)”. In: *Journal of the ACM (JACM)* 53.6 (2006), pp. 937–977. URL: https://doi.org/10.1145/1217856.1217859.

[101] Cyber-Physical Systems Virtual Organization. *Applied Verification for Continuous and Hybrid Systems*. 2023. URL: https://cps-vo.org/group/ARCH.

[103] Stefan Ratschan and Zhikun She. “Safety Verification of Hybrid Systems by Constraint Propagation Based Abstraction Refinement”. In: *ACM Transactions in Embedded Computing Systems* 6.1 (2007).

[104] Jussi Rintanen. “Planning and SAT”. In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, et al. 2nd. IOS Press, 2021.

[105] Malcolm Ryan. “Constraint-based multi-robot path planning”. In: *IEEE International Conference on Robotics and Automation, ICRA 2010*. IEEE, 2010, pp. 922–928.

[106] Malcolm R. K. Ryan. “Exploiting Subgraph Structure in Multi-Robot Path Planning”. In: *J. Artif. Intell. Res.* 31 (2008), pp. 497–542.

[107] Mauricio Salerno, Raquel Fuentetaja, et al. “Train Route Planning as a Multi-agent Path Finding Problem”. In: *Conference of the Spanish Association for Artificial Intelligence*. Springer. 2021, pp. 237–246. URL: https://doi.org/10.1007/978-3-030-85713-4_23.

[108] Thomas Schlechte, Ralf Borndörfer, et al. “Micro–macro transformation of railway networks”. In: *Journal of Rail Transport Planning & Management* 1.1 (2011), pp. 38–48. ISSN: 2210-9706. URL: https://doi.org/10.1016/j.jrtpm.2011.09.001.

[109] Wulf Schwanhäußer. “Die Bemessung der Pufferzeiten im Fahrplangefüge der Eisenbahn”. PhD thesis. 1974. URL: https://www.via.rwth-aachen.de/downloads/Dissertation_Schwanhaeusser_2te_Auflage_Text.pdf.

[110] Roberto Sebastiani and Silvia Tomasi. “Optimization Modulo Theories with Linear Rational Costs”. In: *ACM Transactions Computational Logic* 16.2 (2015). ISSN: 1529-3785. URL: https://doi.org/10.1145/2699915.

[111] Roberto Sebastiani, Silvia Tomasi, and Patrick Trentin. *The OptiMathSMT OMT Solver*. Fondazione Bruno Kessler and DISI-University of Trento, 2022. URL: https://optimathsat.disi.unitn.it/.

[112] Roberto Sebastiani and Patrick Trentin. "OptiMathSAT: A Tool for Optimization Modulo Theories". In: *Proc. International Conference on Computer-Aided Verification, CAV 2015*. Vol. 9206. LNCS. Springer, 2015.

[113] Mohamed Saad Ibn Seddik, Jeremy Nicola, et al. *IBEX*. 2023. URL: https://github.com/ibex-team/ibex-lib.

[114] Guni Sharon, Roni Stern, et al. "Conflict-based search for optimal multi-agent pathfinding". In: *Artif. Intell.* 219 (2015), pp. 40–66.

[115] David Silver. "Cooperative Pathfinding". In: *Proc. of the First Artificial Intelligence and Interactive Digital Entertainment Conference, 2005*. AAAI Press, 2005, pp. 117–122.

[116] Pavel Surynek. "Bounded Sub-optimal Multi-Robot Path Planning Using Satisfiability Modulo Theory (SMT) Approach". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 11631–11637. URL: https://doi.org/10.1109/IROS45743.2020.9341047.

[117] Pavel Surynek. "Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories". In: *Proc. of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*. ijcai.org, 2019, pp. 1177–1183.

[118] Pavel Surynek, Ariel Felner, et al. "Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective". In: *ECAI 2016 - 22nd European Conference on Artificial Intelligence*. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, 2016, pp. 810–818.

[119] Patrick Trentin and Roberto Sebastiani. "Optimization Modulo the Theories of Signed Bit-Vectors and Floating-Point Numbers". In: *Journal of Automated Reasoning* 65 (2021). ISSN: 1573-0670. URL: https://doi.org/10.1007/s10817-021-09600-4.

[120] Thayne T. Walker, Nathan R. Sturtevant, and Ariel Felner. "Extended Increasing Cost Tree Search for Non-Unit Cost Domains". In: *Proc. of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018*. ijcai.org, 2018, pp. 534–540.

[121] Reyk Weiß, Jens Opitz, and Karl Nachtigall. "A Novel Approach to Strategic Planning of Rail Freight Transport". In: *Operations Research Proceedings 2012*. Ed. by Stefan Helber, Michael Breitner, et al. Cham: Springer International Publishing, 2014, pp. 463–468. ISBN: 978-3-319-00795-3. URL: https://doi.org/10.1007/978-3-319-00795-3_69.

[122] Boris de Wilde, Adriaan ter Mors, and Cees Witteveen. "Push and Rotate: a Complete Multi-agent Pathfinding Algorithm". In: *J. Artif. Intell. Res.* 51 (2014), pp. 443–492.

# List of Reviewed Publications of the Author

[74] Tomáš Kolárik and Stefan Ratschan. "Railway Scheduling Using Boolean Satisfiability Modulo Simulations". In: *Formal Methods*. Ed. by Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker. Cham: Springer International Publishing, 2023, pp. 56–73. ISBN: 978-3-031-27481-7. URL: https://doi.org/10.1007/978-3-031-27481-7_5.

[75] Tomáš Kolárik and Stefan Ratschan. "SAT Modulo Differential Equation Simulations". In: *Tests and Proofs*. Ed. by W. Ahrendt and H. Wehrheim. Vol. 12165. LNCS. Springer, 2020. URL: https://doi.org/10.1007/978-3-030-50995-8_5.

[77] Tomáš Kolárik, Stefan Ratschan, and Pavel Surynek. "Multi-Agent Path Finding with Continuous Time Using SAT Modulo Linear Real Arithmetic". In: *International Conference on Agents and Artificial Intelligence*. Ed. by Ana Paula Rocha, Luc Steels, and Jaap van den Herik. SCITEPRESS, 2024.

# List of Remaining Publications of the Author

[38] Tomáš Kolárik. *UN/SOT Core Input Language Specification*. 2019. URL: https://gitlab.com/Tomaqa/unsot/-/blob/master/doc/lang/core.pdf.

[69] Tomáš Kolárik. *MiniSAT with callbacks for lazy online approaches*. 2023. URL: https://gitlab.com/Tomaqa/minisat.

[70] Tomáš Kolárik. *Multi-Agent Path Finding with Continuous Time Solver*. 2023. URL: https://gitlab.com/Tomaqa/mapf_r.

[71] Tomáš Kolárik. *Multi-Agent Path Finding with Continuous Time Visualizer*. 2023. URL: https://github.com/Tomaqa/mapf_r-visualizer.

[72] Tomáš Kolárik. *UN/SOT (UN/SAT modulo ODES Not SOT)*. 2020. URL: https://gitlab.com/Tomaqa/unsot.

[73] Tomáš Kolárik and Stefan Ratschan. "Railway Scheduling Using Boolean Satisfiability Modulo Simulations". In: (2022). Extended version of the paper. URL: https://arxiv.org/abs/2212.05382.

[76] Tomáš Kolárik, Stefan Ratschan, and Pavel Surynek. "Multi-Agent Path Finding with Continuous Time Using SAT Modulo Linear Real Arithmetic". In: (2023). Full version of the paper. URL: https://arxiv.org/abs/2312.08051.

[102] Tomáš Kolárik. *UN/SOT Preprocessing Language*. 2022. URL: https://gitlab.com/Tomaqa/unsot/-/blob/master/doc/lang/preprocess.pdf.