CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
INFORMATIK, WIRTSCHAFTS- UND RECHTSWISSENSCHAFTEN
Department für Informatik

Direct Handling of Ordinary Differential Equations in Constraint-Solving-Based Analysis of Hybrid Systems

Dissertation zur Erlangung des Grades
eines Doktors der Ingenieurwissenschaften

vorgelegt von

M.Sc. Andreas Eggers

Einreichung am 23. April 2014
Disputation am 23. Juli 2014

Gutachter:

Prof. Dr. Martin Fränzle
Prof. Dr. Nacim Ramdani (Université d'Orléans, IUT de Bourges)

weitere Mitglieder der Prüfungskommission:

Prof. Dr. Ernst-Rüdiger Olderog (Vorsitz)
PD Dr. Elke Wilkeit

Abstract

We encode the behavior of hybrid discrete-continuous systems, their initial conditions, and the target states in whose reachability we are interested in a bounded model checking (BMC) formula comprising boolean connectives, arithmetic constraints, and ordinary differential equations (ODEs). These Satisfiability (SAT) modulo ODE formulae are the input to our solver, which combines reasoning for boolean combinations of non-linear arithmetic constraints over discrete and continuous variables in the form of the iSAT algorithm with validated numerical enclosures for ODEs, implemented in the VNODE-LP library. Our scientific contribution (made beforehand in our publications on which this thesis is based) lies primarily in introducing the SAT modulo ODE formalism and the combination of iSAT with VNODE-LP into iSAT-ODE, a solver for SAT modulo ODE formulae. This thesis describes in detail these formalisms, related work, and the components we use. The tight integration that is necessary for an efficient solver requires significant effort to avoid unnecessary yet costly recomputations, the optimization of the computed enclosures to achieve tight bounds and thereby powerful pruning of the search space, and several acceleration techniques. To overcome some of the limitations of VNODE-LP on non-linear differential equations with large initial value sets, we implement bracketing systems from the literature, which exploit monotonicity properties of the ODE systems under analysis, and embed them into the enclosure computation. We validate our approach experimentally on case studies from the literature and of our own design and report results and performance measurements. Our findings show that the direct satisfiability-based analysis of hybrid systems is feasible without the need of rewriting the ODE constraints, that describe these systems' continuous evolutions, into simpler theories. Experimental results, however, also demonstrate the limited scalability that must be expected for a problem of undecidable nature and when using algorithms that may have to pave a search space of exponentially many small boxes in the worst case. While contributing to the state of the art in satisfiability solving for rich theories and in the analysis of hybrid systems, we unfortunately cannot provide the ultimate solution to analyzing arbitrary hybrid systems that play an ever increasingly important role in our lives.

Übersetzung der englischen Zusammenfassung

Wir kodieren das Verhalten diskret-kontinuierlicher hybrider Systeme, ihrer Initialbedingungen und die Zielzustände, an deren Erreichbarkeit wir interessiert sind, in einer Formel mit tiefenbeschränkter Abrollung des Transitionssystems (engl. BMC), die neben booleschen Konnektiven und arithmetischen Ausdrücken somit auch Differentialgleichungen (engl. ODEs) enthält. Diese Satisfiability (SAT) modulo ODE Formeln (sinngemäß: Erfüllbarkeit unter Erfüllung der Differentialgleichungsbedingungen) sind die Eingabe zu unserem Erfüllbarkeitsprüfer (Solver), der das Lösen boolescher Kombinationen von nichtlinearen arithmetischen Bedingungen über diskreten und kontinuierlichen Variablen in Form des iSAT-Algorithmus kombiniert mit der validierten numerischen Integration von Differentialgleichungen, die durch die VNODE-LP Programmbibliothek implementiert wird. Unser wissenschaftlicher Beitrag, den wir im Rahmen der dieser Dissertation zu Grunde liegenden Publikationen geleistet haben, liegt primär in der Einführung des SAT modulo ODE Formalismus und der Kombination von iSAT mit VNODE-LP zum iSAT-ODE-Solver für SAT modulo ODE Formeln. Diese Dissertation beschreibt detailliert diese Formalismen, angrenzende Forschungsarbeiten und die Komponenten, die wir benutzen. Die enge Integration, die für einen effizienten Solver nötig ist, erfordert signifikante Anstrengungen, um unnötige aber kostspielige Wiederholungen von Einschließungsberechnungen zu vermeiden, um die Genauigkeit der berechneten Einschließungen zu optimieren und dadurch ein wirkungsvolles Beschneiden des Suchraumes zu erreichen und erfordert weitere Beschleunigungstechniken. Um einige der Beschränkungen VNODE-LPs auf nichtlinearen Differentialgleichungen mit großen Anfangswertmengen zu überwinden, implementieren wir Systeme oberer und unterer Schranken (bracketing systems) aus der Literatur, die Monotonieeigenschaften des zu analysierende Differentialgleichungssystems ausnutzen, und betten diese in die Berechnung der Einschließungen ein. Wir validieren unseren Ansatz experimentell auf Fallstudien aus der Literatur und aus unserem eigenen Entwurf und berichten die Ergebnisse und Leistungsmessungen. Unsere Ergebnisse zeigen dass die direkte erfüllbarkeitsbasierte Analyse hybrider Systeme machbar ist, ohne Differentialgleichungsbedingungen, welche die kontinuierliche Entwicklung dieser Systeme beschreiben, in einfachere Theorien übersetzen zu müssen. Die experimentellen Ergebnisse demonstrieren aber auch die begrenzte Skalierbarkeit, die man erwarten muss bei einer unentscheidbaren Problemklasse und der Verwendung von Algorithmen, die im schlimmsten Fall einen Raum exponentiell vieler kleiner Suchboxen überdecken müssen. Während die Arbeit also zum Stand der Forschung im Bereich der Erfüllbarkeitsprüfung über reichhaltigen Theorien und in der Analyse hybrider Systeme beiträgt, können wir leider keine ultimative Lösung des Problems der Analyse beliebiger hybrider Systeme liefern, welche eine immer größere werdende Rolle in unserem Alltag spielen.

Acknowledgements

Without Martin Fränzle, this thesis would not exist. I came to his working group before he came back from Lyngby to Oldenburg, when I was hired by Christian Herde as a student assistant. Since then I wrote my Bachelor's and Master's thesis under his supervision and he offered me a job in AVACS to delve deeper into the topics that had emerged from the open questions raised by my Master's thesis. Throughout all these many years he offered his guidance and advice, but it never felt like being steered into one direction. It was much more like having an open ear and brilliant mind available when I felt that I got stuck or could not decide on the best way to proceed. We would then analyze together where our previous steps had taken us, discuss the many possible directions, and thereby distill the decision, which course to set. I am deeply grateful for all the freedom he gave me in this endeavor and, despite the many times I may have felt not up to it, for offering me this great challenge.

It seems to me that in science it is quite common to meet people at conferences or workshops, discuss the great potential of cooperating and joining different approaches, only to later find that the cooperation somehow did not materialize. I am glad that in the case of Nacim Ramdani, this was not the case. He invited me to give a talk at the Small Workshop on Interval Methods in Lausanne in 2009, we invited him to Oldenburg half a year later, and since then, we have had numerous virtual and luckily also a number of real meetings, working hard to combine our approaches and achieving publishable results. This thesis has greatly benefited from this cooperation and I am very happy that Nacim agreed to co-referee my thesis, as much as I am grateful for the welcoming atmosphere and his hospitality during my stays at Bourges.

Thanks must definitely also go to Ned Nedialkov. Writing our papers together, exchanging our progress across the time zones—so that paper writing could follow the sun over the Atlantic—still somewhat amazes me. His insights into VNODE-LP and his support when I felt mathematically unsure, his correction sweeps over our papers, and the good questions he asked all have an important impact also on this thesis.

Of the many roads not traveled during these years, one would have led towards Taylor Models and I am happy that Markus Neher accompanied me on the first steps on that road. We had a number of interesting discussions and meetings, especially one in Prague with Stefan Ratschan, but ultimately the decision to use VNODE-LP and not Taylor Models for handling ODEs in our approach separated our paths. Nonetheless, I am grateful for his help in understanding Taylor Models and their potential, which I am sure will play an important role also in the analysis of hybrid systems.

I cannot start to thank all my colleagues from AVACS, the Hybrid Systems research group, OFFIS, and the university for the many many hours of fruitful or funny, scientific or social conversations. Regarding this thesis itself, the close cooperation with Stefan Kupferschmid, Karsten Scheibler, Natalia Kalinnik, and Tobias Schubert from Bernd Becker's group in Freiburg on the development of iSAT was fundamental. In Oldenburg, Christian Herde and Tino Teige deserve the greatest thanks not only for their scientific contribution to the iSAT solver, but also for the research atmosphere we shared. Christian was my mentor when I started as a student assistant, and his experience, his advice, his opinions, and his passion for research and teaching have been inspiring. With Tino, I did not only have the great pleasure to work together on many topics related to his Stochastic SMT research, but he colored up our routine. Life could never be dull with all these great people around, and I am grateful for this diversity of personalities that I was allowed to enjoy during these years.

My life outside the office was enriched by my family and friends. They allowed me to occasionally get all the research problems out of my mind—which is so important to liberate one's perspective on the scientific questions and thereby to gain a chance for a fresh start come Monday morning. I am deeply grateful to my parents for their encouragement and example to never give up from early on in my life, my mother in her struggle to live, which she lost far too early, and my father to keep our family going ever since, and to give me the chance to study and find my way. When I started my research, my sister's daughter was born, and experiencing her and her siblings' steps of growing up is truly wonderful, albeit also a bit frustrating if one compares the fact that a child can learn everything from mastering its arms and legs to walk and climb trees to even the acquisition of language with the little amount of things that one has learned in the same time and that can be summarized in this thesis. Quite a lesson in humility.

Only a few years back, Joke came into my life and has made it better by so much with his love and support. I am most thankful that I may experience our love and deeply hope that it never ends.

Scientists far too seldom thank the true supporters of our profession—the artists who rekindle our spirits with their music or acting, their writing, their melodies, their wit and humor. They had a tremendous impact in keeping me going when my mind could not find answers to the scientific questions in front of it or could not really relax after thinking too long about something. Musicians and actors, screen writers and poets of all professions: be thanked for your service to keeping scientists go on.

Nearly last, but by no means least, I would like to thank Ernst-Rüdiger Olderog and Elke Wilkeit for serving on my thesis committee, for taking the time to get involved with the ideas presented in this thesis, and for the friendly atmosphere they brought to my thesis defense.

Formally, I shall add thanks at the end in the following way: This work has been supported by the German Research Council DFG within SFB/TR 14 "Automatic Verification and Analysis of Complex Systems". But I would like to add that I am truly grateful to the DFG for offering me a temporary home in science, one which I enjoyed living in, and for giving me the opportunity to make—hopefully meaningful—contributions to the state of the art.

Contents

1 Introduction **13**
- 1.1 Context and Motivation . . . 13
- 1.2 Structure of this Thesis . . . 15
- 1.3 Contributions and Prior Publication . . . 15

2 Foundations and Problem Statement **19**
- 2.1 Hybrid Systems . . . 19
- 2.2 Encoding by Formulae Involving Ordinary Differential Equations 31
- 2.3 Satisfiability Modulo ODE . . . 37
- 2.4 Related Work . . . 43

3 Algorithmic Approach **55**
- 3.1 The Core iSAT Algorithm: Learning Conflicts by Branching and Pruning Intervals . . . 55
  - 3.1.1 Interval Constraint Propagation . . . 56
  - 3.1.2 Input Language and Preprocessing . . . 62
  - 3.1.3 Unit Propagation . . . 66
  - 3.1.4 Branching by Splitting Intervals . . . 67
  - 3.1.5 Implication Graph and Conflict Analysis . . . 68
  - 3.1.6 The iSAT Solver . . . 70
- 3.2 Enclosing Solution Sets of Ordinary Differential Equations . . . 77
  - 3.2.1 From Approximations to Enclosures . . . 80
  - 3.2.2 Fighting the Wrapping Effect . . . 84
  - 3.2.3 Automatic Differentiation . . . 91
  - 3.2.4 VNODE-LP . . . 91
- 3.3 Embedding ODE Enclosures into iSAT . . . 94
  - 3.3.1 Structure of the iSAT-ODE Solver . . . 94
  - 3.3.2 Extended Solver Input . . . 97
  - 3.3.3 Deduction for ODE Constraints . . . 101
  - 3.3.4 Computation of Enclosures and Refinements . . . 105
  - 3.3.5 Backward Deduction . . . 115
  - 3.3.6 Utilizing Deduction Results in iSAT-ODE . . . 118
  - 3.3.7 Acceleration Techniques . . . 121
  - 3.3.8 Direction Deduction . . . 124
- 3.4 Avoiding Repeated Enclosures . . . 127
- 3.5 Building Bracketing Systems Automatically . . . 130

4 Experimental Evaluation **137**
4.1 Bounded Model Checking Case Studies 137
4.1.1 Two Tank System 137
4.1.2 A Conveyor Belt System 145
4.1.3 Comparison with hydlogic 152
4.2 Range Estimation for Hybrid Systems 161

5 Conclusions **167**
5.1 Summary 167
5.2 Topics for Future Work and Open Questions 169

Bibliography **177**

Imagine, you design a machine and you want to know before building it from actual materials whether it will work. So you build a *virtual model*. Your model contains the relevant *physics*, maybe some mechanics or electrical components, and with some probability, it also contains some form of *control*, likely in the form of a *digital* controller, potentially mixed with some *analog* circuits that realize low-level control functions. From the innocent onset to build a model for your machine, you have slipped into creating a *mixture of discrete and continuous* components, a *hybrid system*, and you are lucky that the tool you created your model with supports *simulation*, since otherwise you would not have the faintest idea about whether all the interactions in your model behave in the way you intended. *But does it really work?* Does your system really behave as stated in the *specification* your machine is requested to fulfil? You simulate and you do not find any bugs anymore after changing and tweaking the model a lot, but that does not mean that the system will be *working correctly in every possible situation*—after all, the continuous world offers an *infinite number of starting points and influences*, some of which might lead to disastrous consequences. You have a problem and this thesis does *not* offer the solution. It offers a *glimpse* at what might be possible if a lot more time and energy is spent on continuing this research. It pioneers a *combination* of *safe mathematical evaluation* of the *continuous dynamics described by ordinary differential equations* with automatic *satisfiability search* for formulae consisting of boolean, integer, and real-valued variables—all of which are the *natural ingredients of hybrid system models*.

Chapter 1

Introduction

1.1 Context and Motivation

Hybrid systems are everywhere. Wherever machines are controlled by computers, discrete and continuous components interact, sensors sample physical quantities, digital controllers undergo mode changes and define set points for actuators, which finally manipulate the physical world and whose effects are then measured again. But also some biological systems can be thought of as a combination of continuous evolution and discrete mode changes. Water crystallizing into ice. Chemicals interacting and forming products whose dynamics differ from the reactants. A teacup's continuous movement stopped when it comes into contact with the surface of a table: a mode change, different dynamic laws.

Hybrid systems allow a combined approach at modeling the world. The world becomes discrete if you go far enough down to the smallest relevant pieces, if you model individual atoms or split a deforming piece of metal into thousands of interacting objects. The world becomes continuous if you go far enough up such that all individual pieces blur away and only large-scale effects stay observable. Models can also be made finite or be made continuous by explicitly accepting imprecision, by sampling time, by quantizing continuous variables, by smoothing abrupt changes such that they become continuous.

Accepting imprecision actually is a core principle in modeling: models are built to serve a particular purpose, to make particular observations about the system that is modeled without the need for access to the actual system. Models need only to be good enough such that these observations can be made. Precision however comes in different flavors: merely approximating an observed behavior will lead to a model that will only give approximate answers, whose reliability may be too low or in fact unknown for the purpose of the model. However, if imprecision is taken into account explicitly in the model, if unknown parameters are e.g. represented by intervals covering the entire range of their possible values, models are still imprecise, but at the same time, they cover all behavior of the actual instance, they not only approximate, but instead *overapproximate* the actual system. Introducing a mode change where the actual system evolves with complex continuous dynamics may be imprecise, but if the conditions, under which the dynamic change happens, are overapproximated by the added

discrete mode change, this approach of modeling a complex continuous system by a hybrid system may even allow the generation of a significantly simpler model.

Not only the modeling, but also the analysis of the model is imprecise for most kinds of continuous and hybrid models. Numerical simulation introduces imprecision stemming from rounding in floating-point arithmetics or truncation errors in the integration of differential equations. Sometimes simulators use fixed step sizes and thereby may perform mode changes at a later time point than they would actually occur. All good gained from overapproximating the model may be counteracted by the approximations made e.g. in a traditional simulation tool.

If adherence to a specification is the goal of the analysis, simulations are most likely to be incomplete, since neither all possible initial conditions nor all possible inputs can be tested explicitly with a finite number of simulation runs. No matter how close samples are chosen, there is still a chance that a value in between may cause undesired behavior. In practice, systems are often designed to be robust such that slightly different initial conditions and inputs will lead to comparable system states, avoiding chaotic behavior that would make system design unmanageable. While formal notions of robustness have been exploited to reach completeness, in today's practice a lot of the perceived certainty about the results from simulation and testing may be a form of misguided trust in a fuzzy notion of robustness and may therefore be without measurable degrees of completeness or coverage. A rigorous mathematical analysis would allow a much cleaner certificate of correctness, especially if the consequences of failures may be fatal for life, health, a mission, or a company's business.

The problem however is, that it is a hard problem. The combination of discrete and continuous aspects, their interaction, the infinite search space stemming from the continuous domains, and the huge amount of branching stemming from the non-determinism that is used to overapproximate the system behavior, they all complicate the analysis of hybrid systems. Not much is needed to make the problem even theoretically undecidable, rendering any attempt at building a general algorithm for the automatic analysis of hybrid systems a futile endeavor clearly bound to fail in at least some cases. The computational complexity—even the simplest building block, a propositional satisfiability search, is already well-known to be NP-complete—additionally causes concern about the practical applicability of any incomplete algorithm that one may think of. Why try it anyway?

If successful, automatic analysis of hybrid systems, even if confined to quite abstract models and certainly requiring significant effort for building suitable models, may be a game changer in the development process of complex systems. It may allow new design loops where automatic verification analyzes models built by automatic synthesis tools and gives feedback about errors. It may be the single most helpful tool during engineering, when a debugger complains about an interaction overlooked by the system designer—far in advance of any expensive prototyping. Thinking of ever improving compilers and their help in detecting programming errors, an analysis engine that could be plugged into a tool like Matlab to automatically check a specification on the fly, would certainly be highly supportive. Checking the most abstract, most high-level models of the system early in the design process will save time and cost, reduce redesign loops, will allow engineers to be more audacious about their designs and thereby

open new, more efficient, design options to them previously thought to be too dangerous because they could not be explored sufficiently deeply. Having a tool at intermediate stages that checks refined models of subsystems against the relevant contracts from the specification will be equally helpful. This is why computer scientists have made steps into this direction despite their knowledge that theoretical results may stand in their way at some point and that hardware may be too limited to get results for real-world models. This is why in this thesis an approach is explored, that attempts to combine techniques which each handle individual aspects of hybrid system's analysis. This is why even limited success with regard to the system sizes that can be handled today, means a step forward towards a great goal on an adventurous journey that may take a few more decades of traveling time.

1.2 Structure of this Thesis

This first chapter is dedicated to the motivation of and introduction into the topic. In the second chapter, the problem is developed in more detail, introducing hybrid systems formally and their encoding by constraint systems involving ordinary differential equations (ODEs). The second chapter also briefly sketches the surrounding research landscape. The core of this thesis lies in chapters three and four which present the algorithmic approach, a combination of interval-based constraint solving with safe numerical enclosures for ODEs and the experimental evaluation of this algorithm's implementation on a number of case studies. Chapter five finally concludes the thesis with a summary of the findings and an outlook on potential future work.

1.3 Contributions and Prior Publication

This thesis as a document contains little that has not been published beforehand in journals or workshop and conference proceedings by the author together with other researchers, who contributed mostly their thoughts in discussion, sometimes their guidance, concrete software in the case of Ned Nedialkov in the form of the VNODE-LP library [Ned06], its documentation and insights into its implementation, and who participated in the interpretation of results and the writing of the papers. In the scientific context of a DFG-funded research project, an over-six-year research without intermediate and joint publications would have been impossible and therefore clearly the desired goal of producing doctoral dissertations not attainable without accepting the fact that these theses will mostly combine research that has been peer-reviewed and published before the thesis is written. This is a well-accepted state of affairs in computer science and those who grade this dissertation are well aware of it. It needs to be stated clearly and prominently at this point nonetheless to make it visible to every reader that this document is known to consist to its largest degree of material that has been published before—though mostly in a less detailed manner than possible here. The greatest efforts have been made to clearly name all these publications in the respective chapters, but no attempt is made to explicitly quote these excerpts and highlight all the small and larger changes that need to be done for the sake of readability in this larger context. The proposed referees

of this thesis have been coauthors of the relevant publications and are therefore the best to judge on the author's contribution to those original papers and hence to also judge on whether these contributions to the state of the art are suitable as doctoral dissertation.

Research Question. Before discussing the necessary formal notions and fundamentals in the next chapter, an abstract definition of the research question is already possible and useful: the goal of this thesis is to examine a satisfiability-based approach to hybrid systems verification, i.e. to find a way to analyze hybrid systems automatically via checking the satisfiability of formulae that represent all their possible behavior. An explicit goal of this approach is to avoid those abstraction steps during modeling that have to be employed when the formula class does not directly support ordinary differential equations. The thesis therefore strives to answer the research question, how existing satisfiability solving can be extended to cover a class of formulae that contain ODEs and thereby allow a direct encoding of hybrid systems.

Contributions to the State of the Art. In the original papers on which this thesis is based, the author of this thesis made the following scientific contributions with the support as stated above by the respective coauthors.

We pioneered the combination of safe enclosure methods for ODEs with a solver for constraint-systems over boolean, integer, and real-valued variables [EFH08] and were the first to describe the Satisfiability (SAT) modulo ODE problem that has later been taken up by other researchers [IUH11, GKC13]. This pioneering work was still based on an earlier work by the author [Egg06] using an own implementation of a Taylor-series based safe enclosure mechanism for the enclosures of ODEs based on the description of the AWA algorithm [Loh88] embedded prototypically into the HySAT-II tool [FHR+07].

Apart from the subsequently named points which clearly provide a significant difference between the earlier work done for the author's Master's thesis [Egg06] and this thesis, also the algorithmic basis has been entirely redone in a different way: by not using a limited own implementation for the enclosure generation, but instead integrating and extending a state-of-the art interval numerical library; by not discarding learned facts during the backtracking search, but instead preserving them by a learning scheme and a complex caching structure to recognize previously encountered queries; and by not prototypically embedding the ODE-related parts into the HySAT-II tool, but modularly adding them to its successor iSAT to whose development the author contributed and later extracting them into a stand-alone library which has been made available to the scientific community.

Based on theoretical work and manually constructed examples by Ramdani et al. [RMC09], we extended our algorithm with an automatic on-the-fly generation of bracketing systems and evaluated the implementation on a well-known academic case study from the literature [ERNF11].

Subsequently, the latter publication has been significantly extended with an implementation of a more efficient evaluation of the enclosures generated by VNODE-LP, a tighter coupling of the direct and bracketing methods, and an evaluation on a larger number of case studies including a comparison with the closest competitor tool [IUH11] on their examples and a larger own case study

of a system of parallel hybrid automata with non-linear ODEs and non-linear flow invariants [ERNF12a].

The author has contributed to the development of the core iSAT solver and has done all algorithm and data structure design and all implementations that were necessary to combine the VNODE-LP library with iSAT into iSAT-ODE. Also all implementations of the bracketing approach have been done by the author. The evaluation, including the modeling of the hybrid systems in the tool's input language and the debugging of these models, the setup and execution of the experiments and the extraction and visualization of the results from the solver's output has been done entirely and solely by the author. The interpretation of results has been done in cooperation with the coauthors. The author has written the significant central parts of the publications including the description of the iSAT-ODE algorithm and the evaluation. The proposed referees of this thesis have coauthored these relevant publications and are witnesses of these facts.

Chapter 2

Foundations and Problem Statement

2.1 Hybrid Systems

The most wide-spread type of of hybrid system models can probably be found in the form of graphical modeling languages like the ones from Matlab's Simulink and Stateflow tools. The ability to simulate the model and to organize large models hierarchically are certainly as much key selling points as a (seeming) simplicity of the graphical language, which hides much of its complexity and subtleties behind an easy-to-use interface.

Yet, from an academic perspective, the richness of many practically-used modeling languages and their often incompletely described formal semantics complicate reasoning about the properties of the models and the language itself. Therefore, theoretical results about hybrid systems, which are now as old as two decades, have been obtained using a more confined formalism given in the form of *hybrid automata* like the one used in [Hen96]. However, with proper restrictions and sufficient care about semantics, also models created for example with Simulink and Stateflow can be converted into a form that is suitable for rigorous analysis.

In this thesis, most examples are illustrated by hybrid automata or parallel compositions thereof. The actual basis for the analysis, however, are formulae that encode the systems' behaviors. Whatever original models are used, the relevant property is therefore that they have formal semantics that allow the generation of these formulae. We first recall the notion of hybrid automata from the literature and throughout this chapter develop an encoding using this base formalism.

Definition 1 (expression, constraint, predicate)**.** Given a set of boolean, integer, and real-valued variables $\{x_1, \ldots, x_n\}$, an *expression* is composed of the variables and constants from the rational numbers ($\mathbb{Q}$) using standard arithmetic operators like $+, \cdot, -, /, \sin, \cos, \exp, \log, \sqrt{}, \ldots$ with the usual precedence and bracketing rules. *Constraints* are formed over these expressions by combining two expressions with relational operators ($\leq, <, =, \neq, >, \geq$). Finally, boolean variables and constraints form the most basic *predicates*, which may then be

combined into a larger predicate $p(x_1, \ldots, x_n)$ by using boolean connectives, e.g.

$$p(a,x,y,z) \equiv \underbrace{\neg \underbrace{a}_{\text{boolean variable}} \Rightarrow \underbrace{\underbrace{(x \geq 0.2 \vee y \geq \underbrace{2 \cdot \sqrt{z} - x}_{\text{expression}})}_{\text{constraint}}}_{\text{predicate}} \wedge \frac{x}{z} \leq 0}_{\text{predicate}}.$$

We additionally allow to write e.g. $x \in [3, 3.2)$ as an abbreviation for the predicate $x \geq 3 \wedge x < 3.2$.

Definition 2 (hybrid automaton)**.** We loosely follow the definition from [Hen96], but deviate in notation and some details. A *hybrid automaton* $\mathcal{H}$ consists of a vector $\vec{x} = (x_1, \ldots, x_n)^T$ of *continuous variables*, a set $M = \{m_1, \ldots, m_q\}$ of *modes*, and a multiset $J \subseteq (M \times M) \cup (M)$ of *jumps (discrete transitions)*, with elements of J written with intermediate arrows in the form $m_O \longrightarrow m_D$ or $\epsilon \longrightarrow m_I$—calling m_O the *origin* of a jump, m_D the *destination* of the jump, and m_I an *initial mode* of $\mathcal{H}$. Each mode $m \in M$ is decorated by a *mode invariant* (also called a *flow invariant*) predicate $I_m(\vec{x})$, which describes the admissible values for $\vec{x}$ in m, and whose constraints over $\vec{x}$ are of the form $x \sim c$, with x a continuous variable, $\sim \in \{\leq, \geq\}$, and $c \in \mathbb{Q}$. Most importantly, each mode is attributed by a system of *time invariant ordinary differential equations* $\dot{\vec{x}} = \frac{d\vec{x}}{dt} = \vec{f}_m(\vec{x})$, with $\dot{x}$ denoting the derivative of variable x with respect to time t and the right-hand side $\vec{f}_m : \mathbb{R}^n \to \mathbb{R}^n$ (with $\mathbb{R}$ the real numbers) of the ODE being restricted to functions that are analytic (i.e. infinitely often differentiable)[1] over the entire subset of $\mathbb{R}^n$ that satisfies the invariant I_m (and in practice over a sufficiently large region around it to accommodate for numerical overapproximation). Each jump $j \in J$ is attributed by a combined *guard and action* predicate p_j over variables $\vec{x}$ denoting the state before the jump and primed variables $\vec{x}'$ denoting their state after the jump. For *initial conditions*, i.e. jumps leading from ϵ to an initial mode, the prime-decoration of variables can be (and usually is) omitted since only their state after the jump is relevant. To distinguish guards, which define the condition on the state before the jump is taken, from actions, which define how the variables may change during the jump, guard and action predicates may be split, such that only the guard is written as a predicate and the actions are written (following a "/") as a list of assignments with individual (now unprimed) variables on the left-hand sides and expressions over the variables on the right-hand sides. In this case, assignments, that do not change a variable, i.e. those derived from $x' = x$ constraints, are allowed to be omitted for brevity. As an example for such a split display of a predicate $p_j(x,y,z) \equiv x \leq 5 \wedge z \geq 2 \wedge x' = x \wedge y' = y \wedge z' = -z$ consider the equivalent rewriting $(x \leq 5 \wedge z \geq 2)/(z := -z)$.
To facilitate *parallel composition* of automata, an additional vector $\vec{c}$ of boolean *communication variables* can be added to the system of parallel automata. Communication variables can be used as boolean variables in mode invariants as well as in jump predicates of all automata to flexibly realize different types

[1] Our method actually supports ODEs whose right-hand sides are *sufficiently often* differentiable, with the meaning of "sufficiently" depending on a solver parameter typically chosen to be in the order of $20 - 30$, and locally Lipschitz continuous. Depending on the reader's taste, one or the other restriction may be used.

of synchronization behavior: forcing e.g. a communication variable to be true in the jump predicates of two automata and forcing it to false during all mode invariants will synchronize the jumps of these two automata. Similarly, they can be used to allow jumps in one automaton only if another is currently in a specific mode.
Additionally, all variables of each component automaton may occur within predicates and right-hand sides of ODEs of all other components, but may not be be left-hand-side variables of ODE constraints in any other automaton than the one they belong to. Having hybrid automata $\mathcal{H}_1, \ldots, \mathcal{H}_z$ with respective variable vectors $\vec{x}_{\mathcal{H}_1}, \ldots, \vec{x}_{\mathcal{H}_z}$, the jump predicate for a jump j can therefore be given over the entire set of unprimed and primed variables: $p_j(\vec{x}_{\mathcal{H}_1}, \vec{x}'_{\mathcal{H}_1}, \ldots, \vec{x}_{\mathcal{H}_z}, \vec{x}'_{\mathcal{H}_z}, \vec{c}, \vec{c}')$ (for an initial jump j of course only over unprimed variables), similarly the flow invariant for a mode m: $I_m(\vec{x}_{\mathcal{H}_1}, \ldots, \vec{x}_{\mathcal{H}_z}, \vec{c})$, and finally the ODE system in mode m of automaton $\mathcal{H}_i$ may define the behavior of $\vec{x}_{\mathcal{H}_i}$ using all continuous variables, instead of just the ones belonging to $\mathcal{H}_i$: $\dot{\vec{x}}_{\mathcal{H}_i} = \vec{f}_m(\vec{x}_{\mathcal{H}_1}, \ldots, \vec{x}_{\mathcal{H}_z})$.
Hybrid automata can be and almost exclusively are represented in *graphical form*. Modes are then drawn as vertices and discrete transitions as edges of a directed multigraph. Edges are also introduced for jumps into initial modes, but no ϵ-location is added, i.e. these edges are drawn without a predecessor. Edges are labeled with the guard and action predicates of the corresponding jump, while vertices are labeled with the ODE and flow invariant predicate of the mode they represent.

Flow Invariants. Although our definition restricts flow invariants to the form $x \sim c$, more general conditions on flows that may be desirable in practice can still be modeled with this formalism. Assuming to find in a mode m's invariant a more general flow invariant constraint $\gamma(\vec{x}(t)) \sim c$ (retaining for clearer illustration the explicit t that we drop most of the time in accordance to the usual practice for ODEs), we only require $\gamma(\vec{x}(t))$ to be *differentiable* with respect to t and the derivative to be analytic since it will be used inside the system of differential equations. We can then introduce a new variable $y(t) := \gamma(\vec{x}(t))$ and replace the original flow invariant constraint $\gamma(\vec{x}(t)) \sim c$ in I_m with $y(t) \sim c$. Additionally, we add $y := \gamma(\vec{x})$ to all incoming jumps for the mode m. Using the assumption of differentiability, we can add in mode m to the system of differential equations

$$\dot{y} = \frac{d\gamma(\vec{x}(t))}{dt} = \sum_{i=1}^{n} \frac{\partial\gamma(\vec{x}(t))}{\partial x_i(t)} \cdot \frac{dx_i(t)}{dt}.$$

Making use of all the known right-hand sides for $\dot{x}_i$, we can rewrite this into

$$\dot{y} = \sum_{i=1}^{n} \frac{\partial\vec{x}(t)}{\partial x_i(t)} \cdot f_{(m,i)}(\vec{x}),$$

with $\vec{f}_m = (f_{(m,1)}, \ldots, f_{(m,n)})$. The fundamental theorem of integral calculus together with the semantics that will be introduced below then guarantees that the solution function for y will hold the same value as the original expression $\gamma(\vec{x}(t))$ for the entire duration that the automaton dwells in mode m. For all other modes, we can simply force the new variable y to be constant by adding $\dot{y} = 0$ or add the same dynamic law as for mode m. Of course, y then becomes one of the normal continuous variables, hence increasing the dimensionality of the system by one.

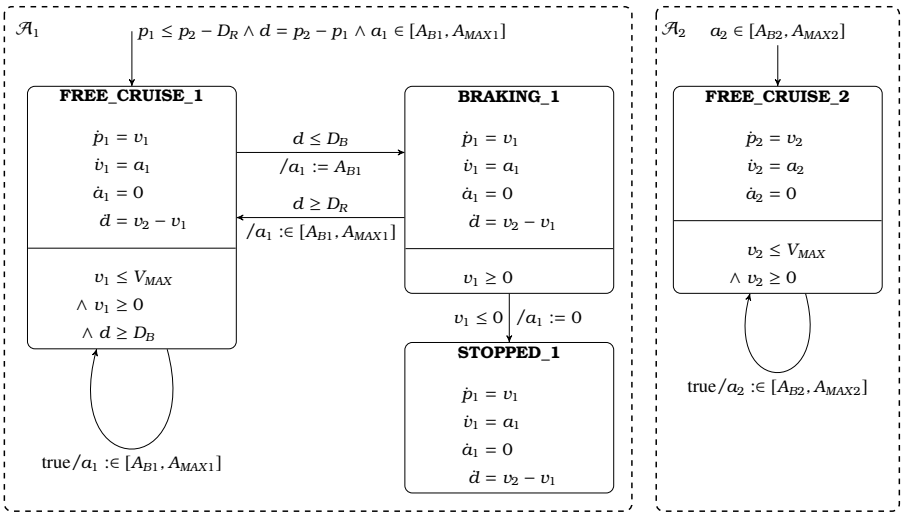


Figure 2.1: System of two simple hybrid automata modeling the behavior of two vehicles and an emergency braking system in the follower car.

This rewriting occurs entirely on the automaton level and is used in some of our the examples.

Remark. An important difference of our definition with respect to the definition from [Hen96] is the restriction to ODEs instead of allowing more general inequality constraints over the variables and their derivatives. While this more general kind of constraint is certainly desirable from a modeling perspective (especially due to its usefulness for formulating overapproximations), it significantly complicates the analysis: instead of "just" dealing with ODEs, a direct approach for the more general class would have to also cover the handling of differential algebraic equations and differential inequalities. For the purpose of this thesis, we will mostly ignore these richer formalisms, and will only occasionally refer to them when discussing related work.

Running Example. As an example, consider the two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ depicted in Figure 2.1. Their parallel composition models the longitudinal movement of two vehicles driving behind each other on the same lane. While $\mathcal{A}_2$ always allows jumps at which the acceleration variable a_2 can be freely chosen to have a new value, the automaton $\mathcal{A}_1$, which models the following vehicle, implements an emergency braking protocol: whenever the distance between the vehicles decreases below D_B (a constant parameter which needs to be instantiated with a rational constant prior to analysis), the acceleration a_1 is forced to a (negative) braking velocity A_{B1} and the vehicle may return to its *FREE_CRUISE_1* mode only if the distance has grown above the brake-release distance D_R. If the velocity reaches zero while the vehicle is forced to brake and the distance has not grown sufficiently, the *STOPPED_1* mode is entered and cannot be left again.

This model can also be considered an example for rewriting a differentiable complex flow invariant into the simpler structure allowed by our formalism. Consider the flow invariant $p_2 - p_1 \ge D_B$, which requires position p_2 to be at least

D_B distance units in front of position p_1. First, we introduce a variable d, whose value represents the distance and hence fortunately has an intuitive real-world semantics. Subsequently, we rewrite the invariant into the form $d \geq D_B$. As initialization on the incoming edge $\epsilon \longrightarrow$ *FREE_CRUISE_1*, we add $d := p_2 - p_1$ and since all other incoming transitions do not change the value of p_1 and p_2, we can omit explicit reinitializations on these edges. For the dynamics of d, we compute the symbolic partial derivatives with respect to p_1 and p_2 and obtain:

$$\dot{d} = \frac{\partial p_2 - p_1}{\partial p_1} \cdot \dot{p_1} + \frac{\partial p_2 - p_1}{\partial p_2} \cdot \dot{p_2} = -1 \cdot v_1 + 1 \cdot v_2 = v_2 - v_1,$$

which also intuitively makes sense since the distance simply grows or shrinks according to the velocity difference between the two vehicles.

We use this simple model to illustrate the necessary definitions and to exemplify the subsequent conversion steps needed for a satisfiability-based analysis. This conversion is based on an encoding of the possible *traces* of a hybrid system.

Definition 3 (valuation)**.** A *valuation* σ maps variables to values. Given a vector $\vec{x} = (x_1, \ldots, x_n)^T$ of n variables (or a set $\{x_1, \ldots, x_n\}$ of n variables with an ordering for that matter) and a point $\vec{y} = (y_1, \ldots, y_n) \in \mathrm{dom}(x_1) \times \cdots \times \mathrm{dom}(x_n)$ from the variables' domains, the valuation σ maps all x_i to their respective values $\sigma(x_i) = y_i$, i.e. $\sigma(\vec{x}) = \vec{y}$. We allow valuations to be concatenated into larger valuations: having vectors $\vec{a} = (a_1, \ldots, a_n)$ and $\vec{b} = (b_1, \ldots, b_m)$ and having valuations $\sigma_1(\vec{a}) = (\sigma_1(a_1), \ldots, \sigma_1(a_n))$ and $\sigma_2(\vec{b}) = (\sigma_2(b_1), \ldots, \sigma_2(b_m))$, the concatenation is defined as $(\sigma_1(\vec{a}), \sigma_2(\vec{b})) := (\sigma_1(a_1), \ldots, \sigma_1(a_n), \sigma_2(b_1), \ldots, \sigma_2(b_m))$.

Definition 4 (satisfaction of predicates and constraints)**.** Given a predicate $p(\vec{x})$ and a valuation $\sigma(\vec{x})$ for the variables occurring in the predicate, we call the predicate *satisfied* by the valuation, $\sigma \models p$, if and only if p evaluates to true under the standard semantics of the boolean connectives, relations, and arithmetic operators, when for each variable x_i in p the value $\sigma(x_i)$ is assumed. Similarly, we call constraints (and occasionally even boolean variables) satisfied, if under σ they evaluate to true.

Definition 5 (trace semantics of one hybrid automaton)**.** Again loosely following the definitions from [Hen96], which explicitly describe (infinite-state) *labeled* and *timed transition systems*, we define the semantics of hybrid automata by their transitions. To simplify the exposition, we consider the *current* (or *active*) *mode* of an automaton $\mathcal{H}$ a variable m with $\mathrm{dom}(m) = M$ and define σ_M, a valuation, that assigns a mode from M to m. We call $\mathcal{S} = (\sigma_M(m), \sigma(\vec{x})) \in M \times \mathbb{R}^n$ a *state* of $\mathcal{H}$, which consists of the active mode $\sigma_M(m)$ and the valuation $\sigma(\vec{x})$ of the automaton's continuous variables $\vec{x}$. A *trace* of $\mathcal{H}$ is a possibly infinite sequence of states $\mathcal{S}_0, \mathcal{S}_1, \ldots$ with the following properties:

- $\mathcal{S}_0 = (m_0, \sigma_0)$ satisfies the initial condition, i.e. the mode of $\mathcal{S}_0$ is initial, i.e. there is $j \in J$ such that $j \equiv \epsilon \rightarrow m_0$, and the state's valuation σ_0 satisfies the initial predicate of the jump which makes the mode initial, i.e. $\sigma_0(\vec{x}) \models p_j(\vec{x})$ (again considering all variables to occur in the unprimed version in this jump's predicate even though they describe the state after the initial jump has been taken).

- Each pair of subsequent states $\mathcal{S}_i = (m_i, \sigma_i(x_1), \ldots, \sigma_i(x_n))$ and $\mathcal{S}_{i+1} = (m_{i+1}, \sigma_{i+1}(x_1), \ldots, \sigma_{i+1}(x_n))$ is connected by either a flow or a jump:
  - $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ are connected by a jump if there exists a jump $j \in J$ such that $j \equiv m_i \longrightarrow m_{i+1}$ and the valuations satisfy this jump's guard and action predicate $p_j(\vec{x}, \vec{x}')$, while the mode invariant of the origin of the jump I_{m_i} holds in $\mathcal{S}_i$ and the invariant $I_{m_{i+1}}$ of the destination of the jump holds in $\mathcal{S}_{i+1}$, i.e.

$$\begin{aligned} & (\sigma_i(\vec{x}), \sigma_{i+1}(\vec{x})) \models p_j(\vec{x}, \vec{x}') \\ \wedge\ & \sigma_i(\vec{x}) \models I_{m_i}(\vec{x}) \\ \wedge\ & \sigma_{i+1}(\vec{x}) \models I_{m_{i+1}}(\vec{x}). \end{aligned}$$

  - $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ are connected by a flow of duration δ if there is no mode change, i.e. $m_i = m_{i+1}$, and the valuations of the continuous variables are connected by a δ-long solution trajectory of the system of ordinary differential equations active in m_i and this solution trajectory does not leave the region that is admissible by the mode's flow invariant. More precisely, given ODEs and flow invariants from mode m_i, $\dot{\vec{x}} = \vec{f}_{m_i} = ((\vec{f}_{m_i})_1(x_1, \ldots, x_n), \ldots, (\vec{f}_{m_i})_n(x_1, \ldots, x_n))^T$ and I_{m_i}, this condition is satisfied if there exists a differentiable function $\vec{y} : \mathbb{R} \to \mathbb{R}^n$, such that $\vec{y}(0) = \sigma_i(\vec{x})$, $\vec{y}(\delta) = \sigma_{i+1}(\vec{x})$, and for all $j = \{1, \ldots, n\}$ and all $t \in [0, \delta]$ the slope of the solution trajectory satisfies the ODE: $\dot{y}_j(t) = (\vec{f}_{m_i})_j(y_1(t), \ldots, y_n(t))$, while all of the solution trajectory's intermediate and end-point values satisfy the flow invariant, i.e. for all $t \in [0, \delta]$, a valuation at time t, $\sigma(\vec{x})(t)$, given by the value $\vec{y}(t)$ satisfies $I_{m_i}(\vec{x})$: $\sigma(\vec{x})(t) = \vec{y}(t) \models I_{m_i}(\vec{x})$.

Concurrency. For the parallel execution of multiple automata or other types of transition systems (like first and foremost parallel programs for example), traditionally, there are two major models: *synchronous jumps* of all components, also called *true concurrency*, and on the other hand *asynchronous jumps* (often referred to as *interleaving*) under the assumption of a *scheduler* that selects one component at a time for making its jump (see e.g. Chapter 2 of [BK08] for a more nuanced overview of concurrent transition systems). These models of concurrency reflect fundamental paradigms of computation: single units (e.g. processors) requiring sequentialization and interleaving versus parallel units allowing truly parallel execution of commands.

For parallel compositions of *hybrid* automata, additionally the time that is spent between mode switches is particularly important, due to the evolution of the continuous variables during this time. However, the coupling of components via shared variables representing physical (and hence naturally truly concurrent) entities like e.g. velocities, positions, temperatures, or forces would render an asynchronous model of evolutions not a very suitable choice for an execution model of continuous flows. Between jumps, time is hence understood as passing synchronously for all component automata.

Since the predicative encoding we will later use for the analysis easily supports both paradigms and many intermediate forms, the semantics of the concurrency model chosen for the automata is less relevant than knowing that there are different options to choose from. For the sake of clarity of the

representations, we nonetheless have to define, which model of concurrency we will use in the examples.

In this thesis, we opt for a model which allows synchronous jumps of all components and *implicit* stuttering, i.e. each mode can be understood as having an invisible self-loop that can be taken whenever a jump is performed by another component automaton and which does not change the automaton's state. A jump of the parallel system therefore interrupts all continuous evolutions, but only one component automaton really needs to take one of its jumps, while the others are allowed to perform a stuttering jump or to also take a real jump simultaneously.

Note that while a stutter jump has no other guard than true and no other action than to keep all continuous variables of the stuttering automaton at their previous value, it does in fact observe the mode invariant of the mode in which stuttering occurs. In particular, these mode invariants can enforce a particular valuation of the communication variables, which may cause a jump of one component to be incompatible with another component dwelling in a particular mode, thereby effectively "communicating" the current mode of one component into the guard of another component's jump.

By introducing a special communication variable that is forced to true in some or all of the modes of one automaton and required to be false in some or all guards of another automaton, stuttering can even be forbidden selectively or entirely for the first automaton. Making the assumption that stuttering jumps are implicitly allowed therefore does not impede our model's generality.

Definition 6 (trace semantics for the parallel composition of hybrid automata)**.** For parallel composition of automata $\mathcal{H}_1, \ldots, \mathcal{H}_z$, the notion of a state $\mathcal{S}$ introduced in Definition 5 is extended to consist of a valuation σ_M of the currently active mode variables $m_{\mathcal{H}_1}, \ldots, m_{\mathcal{H}_z}$ of all component automata, of a valuation σ for all continuous variables $\vec{x}_{\mathcal{H}_1}, \ldots, \vec{x}_{\mathcal{H}_z}$, and of a valuation σ_C for the boolean communication variables $\vec{c}$: $\mathcal{S} = (\sigma_M(m_{\mathcal{H}_1}), \ldots, \sigma_M(m_{\mathcal{H}_z}), \sigma(\vec{x}_{\mathcal{H}_1}), \ldots, \sigma(\vec{x}_{\mathcal{H}_z}), \sigma_C(\vec{c}))$. To simplify the exposition, we introduce a joint vector of current modes $\vec{m} := (m_{\mathcal{H}_1}, \ldots, m_{\mathcal{H}_z})^T$ and a vector $\vec{x}$ of all $n = n_{\mathcal{H}_1} + \ldots + n_{\mathcal{H}_z}$ continuous variables as the concatenation of the component variable vectors

$$\vec{x} := ((\vec{x}_{\mathcal{H}_1})_1, \ldots, (\vec{x}_{\mathcal{H}_1})_{n_{\mathcal{H}_1}}, (\vec{x}_{\mathcal{H}_2})_1, \ldots, (\vec{x}_{\mathcal{H}_z})_{n_{\mathcal{H}_z}})^T.$$

For these, a state then includes valuations $\sigma_M : \vec{m} \to (M_{\mathcal{H}_1} \times \cdots \times M_{\mathcal{H}_z})$ and $\sigma : \vec{x} \to \mathbb{R}^n$. We can now abbreviate the representation of a state significantly: $\mathcal{S} = (\sigma_M(\vec{m}), \sigma(\vec{x}), \sigma_C(\vec{c}))$.

A (potentially infinite) sequence of states $\mathcal{S}_0, \mathcal{S}_1, \ldots$ is a trace of the parallel composition of automata $\mathcal{H}_1, \ldots, \mathcal{H}_z$ if and only if the following properties hold.

- The state $\mathcal{S}_0 = (\sigma_{(M,0)}(\vec{m}), \sigma_0(\vec{x}), \sigma_{(C,0)}(\vec{c}))$ is initial, i.e. there are initial jumps $j_1 \in \mathcal{J}_{\mathcal{H}_1}, \ldots, j_z \in \mathcal{J}_{\mathcal{H}_z}$ with $j_1 \equiv \epsilon \to \sigma_{(M,0)}(m_{\mathcal{H}_1}), \ldots, j_z \equiv \epsilon \to \sigma_{(M,0)}(m_{\mathcal{H}_z})$ and the valuation of the continuous variables and of the communication variables satisfies the respective initial conditions belonging to these initial jumps, i.e. $(\sigma_0(\vec{x}), \sigma_{(C,0)}(\vec{c})) \models (p_{j_1} \wedge \ldots \wedge p_{j_z})$.

- Each pair of subsequent states $\mathcal{S}_i = (\sigma_{(M,i)}(\vec{m}), \sigma_i(\vec{x}), \sigma_{(C,i)}(\vec{c}))$ and $\mathcal{S}_{i+1} = (\sigma_{(M,i+1)}(\vec{m}), \sigma_{i+1}(\vec{x}), \sigma_{(C,i+1)}(\vec{c}))$ is connected by a jump or by a flow.

– $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ are connected by a jump if at least one component automaton $\mathcal{H}_a$ with $a \in \{1, \dots, z\}$ performs a true jump and all other components either stutter or perform jumps as well—all while satisfying the conjunction of the jumps' guard and action predicates and relevant mode invariants. Formally, the system of automata makes a jump if and only if

$$
\begin{aligned}
&\exists a \in \{1,\dots,z\} : && \text{there is } \mathcal{H}_a\\
&\quad (\exists j \in J_{\mathcal{H}_a} : j \equiv \sigma_{(M,i)}(m_{\mathcal{H}_a}) \to \sigma_{(M,i+1)}(m_{\mathcal{H}_a}) && \text{with a jump } j, \text{ whose}\\
&\quad \wedge\, (\sigma_i(\vec{x}), \sigma_{i+1}(\vec{x}), \sigma_{(C,i)}(\vec{c}), \sigma_{(C,i+1)}(\vec{c})) \models p_j && \text{predicate is satisfied}\\
&\quad \wedge\, (\sigma_i(\vec{x}), \sigma_{(C,i)}(\vec{c})) \models I_{\sigma_{(M,i)}(m_{\mathcal{H}_a})} && \text{as are the invariants}\\
&\quad \wedge\, (\sigma_{i+1}(\vec{x}), \sigma_{(C,i+1)}(\vec{c})) \models I_{\sigma_{(M,i+1)}(m_{\mathcal{H}_a})})\\
&\wedge(\forall b \in \{1,\dots,z\}\backslash\{a\} : && \text{and all other } \mathcal{H}_b\text{'s}\\
&\quad (\exists j \in J_{\mathcal{H}_b} : j \equiv \sigma_{(M,i)}(m_{\mathcal{H}_b}) \to \sigma_{(M,i+1)}(m_{\mathcal{H}_b}) && \text{also do true jumps}\\
&\quad \wedge(\sigma_i(\vec{x}), \sigma_{i+1}(\vec{x}), \sigma_{(C,i)}(\vec{c}), \sigma_{(C,i+1)}(\vec{c})) \models p_j\\
&\quad \wedge(\sigma_i(\vec{x}), \sigma_{(C,i)}(\vec{c})) \models I_{\sigma_{(M,i)}(m_{\mathcal{H}_b})}\\
&\quad \wedge(\sigma_{i+1}(\vec{x}), \sigma_{(C,i+1)}(\vec{c})) \models I_{\sigma_{(M,i+1)}(m_{\mathcal{H}_b})})\\
&\quad \vee\, (\sigma_{(M,i)}(m_{\mathcal{H}_b}) = \sigma_{(M,i+1)}(m_{\mathcal{H}_b}) && \text{or stutter,}\\
&\quad \wedge\sigma_i(\vec{x}_{\mathcal{H}_b}) = \sigma_{i+1}(\vec{x}_{\mathcal{H}_b}))) && \text{keeping } \vec{x}_{\mathcal{H}_b} \text{ constant}\\
&\quad \wedge(\sigma_i(\vec{x}), \sigma_{(C,i)}(\vec{c})) \models I_{\sigma_{(M,i)}(m_{\mathcal{H}_b})} && \text{and satisfying the}\\
&\quad \wedge(\sigma_{i+1}(\vec{x}), \sigma_{(C,i+1)}(\vec{c})) \models I_{\sigma_{(M,i+1)}(m_{\mathcal{H}_b})}) && \text{mode invariant.}
\end{aligned}
$$

– $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ are connected by a flow of duration δ if there is no mode change, i.e. $\sigma_{(M,i)}(\vec{m}) = \sigma_{(M,i+1)}(\vec{m})$, and the valuations σ_i and σ_{i+1} are connected by a δ-long solution trajectory of the system of ordinary differential equations active in the modes $\sigma_{(M,i)}(\vec{m})$ and this solution trajectory does not leave the region that is admissible by the flow invariants in these modes. Note that the flow invariants may depend on the communication variables, which however do not change during the flow and which can therefore be substituted by their valuations, i.e. occurrences of a communication variable c are replaced by $\sigma_{(C,i)}(c)$. Hence, having the ODE system

$$
\dot{\vec{x}} = \begin{pmatrix} (\dot{\vec{x}}_{\mathcal{H}_1})_1 \\ \vdots \\ (\dot{\vec{x}}_{\mathcal{H}_1})_{n_{\mathcal{H}_1}} \\ (\dot{\vec{x}}_{\mathcal{H}_2})_1 \\ \vdots \\ (\dot{\vec{x}}_{\mathcal{H}_2})_{n_{\mathcal{H}_2}} \\ \vdots \\ (\dot{\vec{x}}_{\mathcal{H}_z})_{n_{\mathcal{H}_z}} \end{pmatrix} = \begin{pmatrix} (\vec{f}_{\sigma_{(M,i)}(m_{\mathcal{H}_1})})_1(\vec{x}) \\ \vdots \\ (\vec{f}_{\sigma_{(M,i)}(m_{\mathcal{H}_1})})_{n_{\mathcal{H}_1}}(\vec{x}) \\ (\vec{f}_{\sigma_{(M,i)}(m_{\mathcal{H}_2})})_1(\vec{x}) \\ \vdots \\ (\vec{f}_{\sigma_{(M,i)}(m_{\mathcal{H}_2})})_{n_{\mathcal{H}_2}}(\vec{x}) \\ \vdots \\ (\vec{f}_{\sigma_{(M,i)}(m_{\mathcal{H}_z})})_{n_{\mathcal{H}_z}}(\vec{x}) \end{pmatrix} =: \vec{f}(\vec{x})
$$

and conjunction of flow invariants with substituted valuations for

the communication variables

$$I(\vec{x}) = \bigwedge_{j\in\{1,\ldots,z\}} \left(I_{\sigma_{(M,i)}(m_{\mathcal{H}_j})}(\vec{x},\vec{c})\Big|_{(\vec{c}/\sigma_{(C,i)}(\vec{c}))} \right),$$

a flow of duration δ is given by the existence of a differentiable solution function $\vec{y} : \mathbb{R} \to \mathbb{R}^n$, such that $\vec{y}(0) = \sigma_{(M,i)}(\vec{x})$, $\vec{y}(\delta) = \sigma_{(M,i+1)}(\vec{x})$, and $\dot{\vec{y}}(t) = \vec{f}(y(t))$ for all $t \in [0,\delta]$, while the flow invariant is never left, i.e. for each $t \in [0,\delta]$, the valuation $\sigma(\vec{x})(t) = \vec{y}(t) \models I(\vec{x})$.

Running Example (continued). We apply the above definition to illustrate the meaning of a trace using the running example. Automaton $\mathcal{A}_1$ has three possible modes $m_{\mathcal{A}_1} \in M_{\mathcal{A}_1} = \{FREE\_CRUISE\_1, BRAKING\_1, STOPPED\_1\}$ and four continuous variables $\vec{x}_{\mathcal{A}_1} = (p_1, v_1, a_1, d)$ representing the vehicle's position, velocity, acceleration, and distance from the second car, which drives in front of it. The second automaton, $\mathcal{A}_2$, has only one possible mode, i.e. $m_{\mathcal{A}_2} \in M_{\mathcal{A}_2} = \{FREE\_CRUISE\_2\}$, and continuous variables $\vec{x}_{\mathcal{A}_2} = (p_2, v_2, a_2)$. There is no communication variable since the system does not require any synchronization. The system has a number of parameters, whose instantiation has a significant impact on the traces that are possible. For this illustration, we choose the distance at which braking is initiated to be $D_B := 10$, the distance at which the brake is released to be $D_R := 15$, the braking accelerations to be $A_{B1} = A_{B2} := -2$, the maximum accelerations to be $A_{MAX1} = A_{MAX2} := 2$, and the maximum velocity to be $V_{MAX} := 5$.

A state of this system must give a value for each of the mode and continuous variables. Table 2.1 contains a sequence of states $\mathcal{S}_0, \ldots, \mathcal{S}_6$ that shows a trace of this system. We have added a variable t representing the *global time*, to make it easier to distinguish flows and jumps—the latter happening without passage of time.

	t	m_1	p_1	v_1	a_1	d	m_2	p_2	v_2	a_2
$\mathcal{S}_0$	0	*FREE_CRUISE_1*	0	4	1	20	*FREE_CRUISE_2*	20	4	0
$\mathcal{S}_1$	0.8	*FREE_CRUISE_1*	3.52	4.8	1	19.68	*FREE_CRUISE_2*	23.2	4	0
$\mathcal{S}_2$	0.8	*FREE_CRUISE_1*	3.52	4.8	0	19.68	*FREE_CRUISE_2*	23.2	4	0
$\mathcal{S}_3$	12.9	*FREE_CRUISE_1*	61.6	4.8	0	10	*FREE_CRUISE_2*	71.6	4	0
$\mathcal{S}_4$	12.9	*BRAKING_1*	61.6	4.8	-2	10	*FREE_CRUISE_2*	71.6	4	0
$\mathcal{S}_5$	15.3	*BRAKING_1*	67.36	0	-2	13.84	*FREE_CRUISE_2*	81.2	4	0
$\mathcal{S}_6$	15.3	*STOPPED_1*	67.36	0	0	13.84	*FREE_CRUISE_2*	81.2	4	0

Table 2.1: Tabular representation of a trace for the running example.

In this trace, $\mathcal{S}_0$ is an initial state, i.e. its modes are initial and the valuation of the continuous variables satisfies the conjunction of the conditions of the initial jumps:

$$\begin{aligned} &\sigma_0(p_1, v_1, a_1, d, p_2, v_2, a_2) \\ &= (0, 4, 1, 20, 20, 4, 0) \\ &\models p_1 \le p_2 - D_R \wedge d = p_2 - p_1 \wedge a_1 \in [A_{B1}, A_{MAX1}] \wedge a_2 \in [A_{B2}, A_{MAX2}] \end{aligned}$$

Subsequently, a flow of duration $\delta = 0.8$ occurs and leads to state $\mathcal{S}_1$. Easily to be seen, this flow does not change the modes of the component automata. Hidden by the trace semantics is, however, the behavior of the continuous

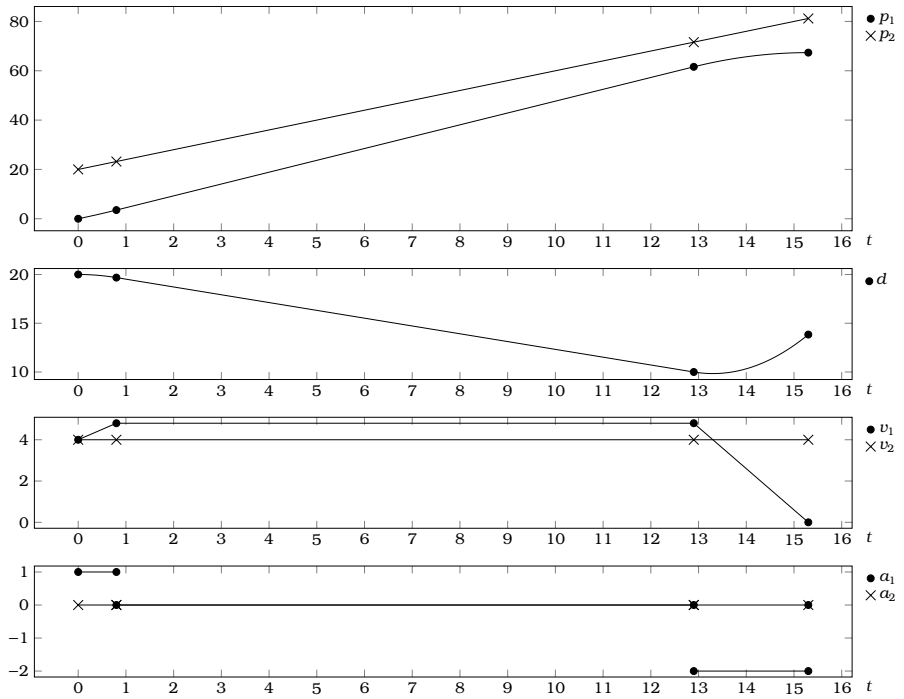


Figure 2.2: Trace from Table 2.1 with visualization of intermediate flows.

variables between $t = 0$ and $t = 0.8$. A trace is only a witness for the existence of a solution trajectory of the ODEs of the system's active modes. In contrast to the general case, for this example we can easily compute closed-form solution functions for $p_1, v_1, a_1, d, p_2, v_2, a_2$ by sorting the ODEs and performing symbolic integration:

$$
\begin{aligned}
p_1(\delta) &= p_1(0) + \delta v_1(0) + {}^1\!/_2\ \delta^2 a_1(0) \\
v_1(\delta) &= v_1(0) + \delta a_1(0) \\
a_1(\delta) &= a_1(0) \\
d(\delta) &= d(0) + (p_2(\delta) - p_2(0)) - (p_1(\delta) - p_1(0)) \\
p_2(\delta) &= p_2(0) + \delta v_2(0) + {}^1\!/_2\ \delta^2 a_2(0) \\
v_2(\delta) &= v_2(0) + \delta a_2(0) \\
a_2(\delta) &= a_2(0)
\end{aligned}
$$

In Figure 2.2, we visualize the example trace over dense time using these solution functions to plot the intermediate flows.

By inserting the valuation $\sigma_0(p_1, v_1, a_1, d, p_2, v_2, a_2)$ as starting point for the flow $(p_1(0), v_1(0), a_1(0), d(0), p_2(0), v_2(0), a_2(0))$ and by setting $\delta = 0.8$, we compute with the closed-form solution $(p_1(\delta), v_1(\delta), a_1(\delta), d(\delta), p_2(\delta), v_2(\delta), a_2(\delta)) = (3.52, 4.8, 1, 19.68, 23.2, 4, 0) = \sigma_1(p_1, v_1, a_1, d, p_2, v_2, a_2)$, i.e. the valuation of state $\mathcal{S}_1$. The flow invariant conditions $v_1(t) \leq V_{MAX} \wedge v_1(t) \geq 0 \wedge d(t) \geq D_B \wedge v_1(t) \leq V_{MAX} \wedge v_2(t) \geq 0$ hold for all $t \in [0, \delta]$—in this example easily deducible from the monotonicity of the solution functions.

A more intriguing question about this flow may be, why its duration is $\delta = 0.8$ time units and the answer may be unsatisfying: because it is allowed; nothing prevents the system from performing a jump after an arbitrary flow duration while it resides in the *FREE_CRUISE_1* mode, as long as it satisfies the constraints on the valuation originating from the guard and action predicates and mode invariants. Similary, also the choice of the initial valuation was by no means unique. This system, albeit looking quite simple, contains a number of sources for non-determinism, which cause the existence of an infinite number of traces, all of which satisfy the conditions we have laid out before.

Following the flow, the transition from state $\mathcal{S}_1$ to $\mathcal{S}_2$ is a jump along the self-loop from $\mathcal{A}_1$'s *FREE_CRUISE_1* mode. Again, non-determinism can be observed in the choice of the new valuation for a_1, which is set from 1 to 0. There is no change in $\mathcal{A}_2$, i.e. it stutters. This pair of states represents a valid jump, since the taken jump's predicate is satisfied

$$\begin{aligned}&(\sigma_1(p_1, v_1, a_1, d, p_2, v_2, a_2), \sigma_2(p_1, v_1, a_1, d, p_2, v_2, a_2))\\ &\models \text{true} \wedge a_1' \in [A_{B1}, A_{MAX1}]\end{aligned}$$

along with the invariants at the origin of the jump by the valuation before the jump is taken

$$\sigma_1(p_1, v_1, a_1, d, p_2, v_2, a_2) \models v_1 \leq V_{MAX} \wedge v_1 \geq 0 \wedge d \geq D_B$$

and the invariant of the target (which is the same mode) by the valuation after the jump is taken

$$\sigma_2(p_1, v_1, a_1, d, p_2, v_2, a_2) \models v_1 \leq V_{MAX} \wedge v_1 \geq 0 \wedge d \geq D_B$$

and the stuttering in the other automaton $\mathcal{A}_2$ does not change the valuation of its variables and also there the currently active mode's invariant is not violated by the valuation neither before

$$\sigma_1(p_1, v_1, a_1, d, p_2, v_2, a_2) \models v_2 \leq V_{MAX} \wedge v_2 \geq 0$$

nor after the jump

$$\sigma_2(p_1, v_1, a_1, d, p_2, v_2, a_2) \models v_2 \leq V_{MAX} \wedge v_2 \geq 0.$$

The subsequent transition from $\mathcal{S}_2$ to $\mathcal{S}_3$ is a flow of duration $\delta = 12.1$. While non-deterministic interruptions of this flow with jumps performing acceleration changes would again have been possible, its maximum duration is determined by reaching the border of the states admitted by the flow invariants. When $\delta = 12.1$, the distance $d = 10$ is reached and therefore any continuation of this flow would cause the flow invariant $d \geq D_B = 10$ of mode *FREE_CRUISE_1* to be violated, since d would decrease further. In $\mathcal{S}_3$, the system could still perform an arbitrary number of self-loop jumps, but due to being on the border of the admissible state space, it could not perform a flow of any non-zero duration.

Jumping from *FREE_CRUISE_1* to *BRAKING_1* in the transition from $\mathcal{S}_3$ to $\mathcal{S}_4$, however, opens up the possibility of a flow, which takes $\delta = 2.4$ time units and leads to the first car's velocity reaching zero ($\mathcal{S}_5$)—at which point a jump into the *STOPPED_1*-mode in $\mathcal{S}_6$ concludes this example trace. Again, there could have been arbitrarily many branches off this trace, e.g. by $\mathcal{A}_2$ performing jumps changing its acceleration.

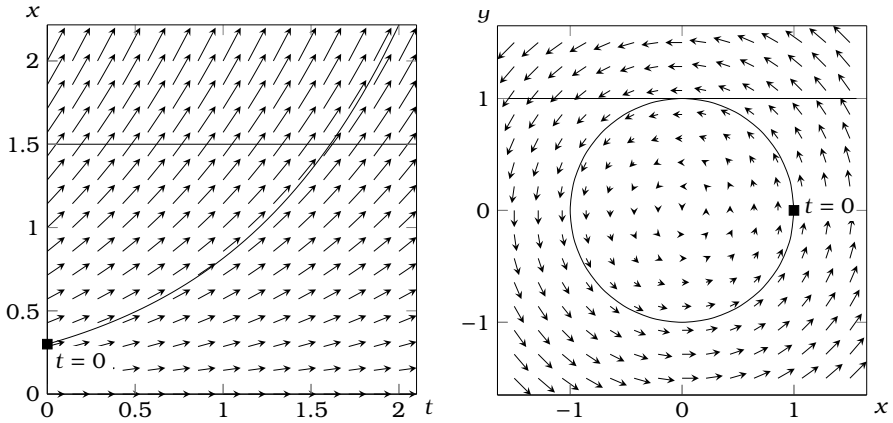


Figure 2.3: On the left: vector field for $\dot{x} = x$ and solution trajectory for $x(0) = 0.3$; on the right: vector field as phase plot (x, y) for harmonic oscillator $\dot{x} = -y, \dot{y} = x$, with a solution trajectory for $(x, y)(0) = (1, 0)$. For the dynamics on the left, a flow invariant like $x \leq 1.5$ can easily be used to model urgency, while for the system on the right, the example trajectory and flow invariant $y \leq 1$ provide an example, where this approach would not suffice to limit the duration to the first point of time, when the switching surface at $y = 1$ is reached.

Urgency. As can be seen in the example, flow invariants can be used to force flows to stop. By using a relaxed version of the negated guard condition, this stop of the flow can be timed to occur exactly at the point, when a guard condition is satisfied. This modeling approach is actually quite classic, instances of it can e.g. already be found in [Hen96]. In our example, the flow invariant $d \geq D_B$ of *FREE_CRUISE_1* is such a relaxed negation of the guard $d \leq D_B$: both constraints share a common value at $d = D_B$, but movement in one direction will invalidate either the flow invariant or the guard condition. In this system, we use this approach to model a behavior, that is often described as *urgency*: a jump shall be taken as soon as its guard is satisfied. While this is sometimes incorporated directly into the semantics, we need to model it explicitly by adding to flow invariants the conjunction of all relaxed negated guard conditions of all outgoing urgent jumps. In this way, as soon as one of the guards becomes true, also the border of the admissible state space for the mode, as defined by the flow invariant, is reached and therefore a flow cannot continue into a region violating the flow invariant.

Note however, that this modeling of urgency via flow invariants is weaker than an approach where an explicit notion of urgency is added to the semantics. In Figure 2.3, we compare two cases: one where flow invariants model urgency exactly, and one, where they are not sufficient to force trajectories to end when the guard of the urgent jump becomes satisfied. The left part of the figure shows the directional field of an ODE and an initial condition for which urgency can easily be modeled by a flow invariant. The trajectory shown on the right side of the figure, however, only touches the *switching surface*, i.e. the common values of guard and flow invariant. Its direction does not force it to leave the region admissible under the flow invariant and it is therefore not forced to end when

first satisfying the guard condition. In such cases only additional modeling tricks may still allow to achieve the desired behavior (e.g. by splitting a mode into multiple modes to explicitly handle regions in the state space where trajectories could slide along switching surfaces).

Overapproximation. The modeling of urgency via flow invariants is one instance of *overapproximation*: while in an "original" system, a particular urgent jump has to be taken as soon as a guard is satisfied, in a system where urgency is modeled by flow invariants, all these trajectories, whose flows end when the guard is reached, are present, too, but alongside there are additional *spurious* trajectories, which do not correspond to any trajectory of the original system.

A formal understanding of overapproximations is often based on the notion of *simulation relations*, see e.g. [BK08, Chapter 7.4]. Having two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$, a simulation relation is a (potentially partial) mapping of states from $\mathcal{T}_1$ to $\mathcal{T}_2$. Furthermore, $\mathcal{T}_2$ is called an *abstraction* of $\mathcal{T}_1$, if for each transition between states in $\mathcal{T}_1$ there exists a transition in $\mathcal{T}_2$ between the states mapped to by the simulation relation. For each trace (i.e. sequence of states connected by transitions) of $\mathcal{T}_1$, an abstraction therefore contains a trace—simply obtained by replacing the states of the sequence with their images under the simulation relation. Traces in $\mathcal{T}_1$ can thus be *simulated* by traces in $\mathcal{T}_2$. However, unless the two transition systems differ merely by renaming of states, there are traces of $\mathcal{T}_2$, which do not have a counterpart in $\mathcal{T}_1$, i.e. they exist in the abstraction, but cannot be *concretized*. The abstract transition system $\mathcal{T}_2$ is therefore called an *overapproximation* of $\mathcal{T}_1$. Note, that in some contexts, abstractions are used with respect only to a specific property of a system. In that case, an overapproximation needs not necessarily contain abstract versions for all concrete traces, but only for those which are relevant with respect to the property.

In this thesis, we will use overapproximations over a wide range of objects. We have seen an example above for their application to hybrid systems, but we will also call a formula an overapproximation of a hybrid system or an interval evaluation of a constraint an overapproximation of its real-valued interpretation. In all these cases, the general intuition is the same: an overapproximation allows the same behavior as the concrete object, but may contain additional spurious solutions or behaviors that cannot be concretized.

2.2 Encoding by Formulae Involving Ordinary Differential Equations

So far, we have only gathered the ingredients for modeling the behavior of hybrid systems. For an analysis, and often also for the process of creating a suitable model, however, we also require an *analysis goal*. In the general case, this can be as complex as a (timed) temporal logic property or the question whether the system stabilizes towards an attractor (see e.g. [KGG+09] for a broad overview of classes of analysis goals). For the purpose of this thesis, we focus on *step-bounded reachability*, with the reachability target being encoded by a *target predicate* over the mode, continuous, and communication variables.

Bounded Model Checking. In [BCCZ99], Biere et al. introduced *Bounded Model Checking* (BMC) for finite-state Kripke structures as an alternative to model checking based on Binary Decision Diagrams (BDDs).[2] A good intuition about BMC is that a finite unwinding of the transition system is encoded by a formula in such a way that this formula is satisfiable if and only if a trace (path) of a length equal (or in some variants less than or equal) to the unwinding depth exists. By adding to this formula an encoding of a bounded linear temporal logic (LTL) property, a solution only exists if a path satisfies this property. Biere et al. also give conditions on the paths and LTL properties such that unbounded results can be deduced from bounded unwindings. They also stress the importance of finding counter examples, e.g. violations of a safety property, in practical applications of model checking. The simplest form of a target condition, showing that an error state is eventually reached (as the negation of a property that holds globally), combined with a progressively increased number of transition unwindings (sometimes called *incremental BMC*), leads to an algorithmic approach that can find error traces with minimal length—well-suited for identifying errors in a system. However, if applied without any knowledge about a maximum path length, BMC is inherently incomplete because it misses error traces that have a length larger than the investigated unwinding depth.

By extending the approach to result in formulae containing not only propositional variables, but richer types of constraints, BMC has later been applied to classes of hybrid automata, whose dynamics are governed by linear expressions over the continuous variables [ABCS05] and by linear inclusion functions assigning in each mode a maximum and minimum slope to each variable [FH05]. Similarly, BMC has been used to analyze *discrete-time* hybrid automata, with post-states depending linearly on pre-states [GPB05].

Even further relaxations of the types of allowed constraints within the predicative encoding allowed the application of BMC to hybrid automata, whose dynamics could be captured by non-linear (including transcendental) expressions over the pre- and post-states of the continuous variables. These have been published for a concrete example in the form of a Simulik/Stateflow model in [HEFT08] and a generalized form of the encoding process can be found in Chapter 2.2 of [Her11].

Our encoding, which is based on what we have first published in [EFH08] and refined in [ERNF12a], relaxes the restriction on the atoms of the formula even further by introducing constraints that allow the direct encoding of the ODEs and flow invariants occurring in the automaton under investigation. This will lead to the notion of *Satisfiability modulo ODE*, which is the topic of the next section.

Definition 7 (variable instantiation)**.** Given a variable v, a k-fold *instantiation* of v is given by introducing variables $v_1, \ldots, v_k$ that all represent different *instances* of the same original variable v. This notion is directly raised to sets and vectors of variables by introduction of instances for each element / component.

[2] Even earlier, in [GvVK94], Groote et al. applied finite unwinding and encoding as a propositional tautology checking problem to the low-level safety control program of a railway system, which can be interpreted as a precursor to BMC.

Predicative Encoding of Hybrid Automata. We now apply the method of bounded model checking as described above to the trace semantics for parallel compositions of hybrid automata $\mathcal{H}_1, \ldots, \mathcal{H}_z$ from Definition 6. First, we introduce a set of variables. In addition to the obvious variables representing the mode, continuous, and communication variables, we declare variables that explicitly encode, which component performs a jump. Furthermore, a variable *flow* indicates whether the subsequent state will be connected via a continuous flow or via a discrete jump. The global time t sums up the individual durations δ of each flow and is not increased when a jump occurs.

$$\begin{aligned}
\vec{m} = (m_{\mathcal{H}_1}, \ldots, m_{\mathcal{H}_z}) &\in M_{\mathcal{H}_1} \times \ldots \times M_{\mathcal{H}_z} \\
\vec{x} = (\vec{x}_{\mathcal{H}_1}, \ldots, \vec{x}_{\mathcal{H}_z}) &\in \mathrm{dom}((\vec{x}_{\mathcal{H}_1})_1) \times \ldots \times \mathrm{dom}((\vec{x}_{\mathcal{H}_z})_{n_{\mathcal{H}_z}}) \\
\vec{c} &\in \{0,1\}^{|\vec{c}|} \\
(jump_1, \ldots, jump_z) &\in \{0,1\}^z \\
flow &\in \{0,1\} \\
t &\in [0, T] \\
\delta &\in [0, \Delta]
\end{aligned}$$

Like BMC for other types of systems, our encoding utilizes predicates over variable instantiations. Let $s := (\vec{m}, \vec{x}, \vec{c}, jump_1, \ldots, jump_z, flow, t, \delta)$. Each state $\mathcal{S}_i$ of the trace, that is to be identified or whose existence is to be disproved by BMC, is encoded by one instance s_i of the variables introduced above. The predicate $init(s)$, encoding the initial state $\mathcal{S}_0$, is applied to the instance s_0. For a k-fold unwinding of the transition system (i.e. jumps and flows), we introduce a predicate $trans(s, s')$, which is applied to pairs (s_i, s_{i+1}) for $i \in \{0, \ldots, k-1\}$. And finally, since a target condition must hold in the final state of the trace, the target predicate $target(s)$ must hold for the instance s_k. The bounded model checking formula is thus:

$$\Phi = init(s_0) \wedge trans(s_0, s_1) \wedge \ldots \wedge trans(s_{k-1}, s_k) \wedge target(s_k). \tag{2.1}$$

Definition 6 requires that in the initial state $\mathcal{S}_0$ of a trace, each component automaton is in an initial mode and the valuations of the continuous and communication variables are compatible with the initial jumps that lead to the corresponding modes. Consequently, a predicate $init(s)$ must encode these two properties, i.e.

$$init(s) \equiv \underbrace{\bigwedge_{i \in \{1,\ldots,z\}} \bigvee_{j=(\epsilon \longrightarrow m) \in J_{\mathcal{H}_i}} \Big(m_{\mathcal{H}_i} = m \;\wedge\; p_j(\vec{x}, \vec{c}) \underbrace{\;\wedge\; I_m(\vec{x}, \vec{c})}_{\ldots \text{as well as the invariant of the initial mode resulting from taking this jump.}} \Big)}_{\text{For each automaton, one initial jump must be taken and its predicate must hold...}} \wedge \underbrace{t = 0}_{\text{Global time starts at 0.}}$$

The $target(s)$ predicate, that is applied to the last instance s_k, must be chosen in such a way that it characterizes a state $\mathcal{S}_k$ of the trace, whose reachability is under investigation. Such a predicate can e.g. consist of just a particular

mode—like in classical reachability for finite state automata—or can also describe a region in the state space spanned by the mode and continuous variables, including conditions on the global time.

For all pairs of consecutive instances s_i and s_{i+1} between the initial and target instance, the transition predicate $trans(s, s')$ encodes the discrete jumps and continuous flows, which connect pairs of consecutive states $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ in the trace.

$$
\begin{aligned}
trans(s, s') \equiv \quad & \underbrace{t' = t + \delta}_{\text{Sum of durations.}} \\
\wedge \; & \underbrace{(\neg flow \Rightarrow \delta = 0)}_{\text{Jumps do not take time.}} \\
\wedge \; & \underbrace{\left(\neg flow \Rightarrow \left(\bigvee_{i \in \{1,\dots,z\}} jump_i \right)\right)}_{\text{A step is either a flow or at least one component must do a jump.}} \\
\wedge \; & \underbrace{\bigwedge_{i \in \{1,\dots,z\}} \Big(jump_i \Rightarrow}_{\text{If a component performs a discrete transition, } \dots} \\
& \Big(\underbrace{\bigvee_{j=(m \longrightarrow m') \in J_{\mathcal{H}_i}}}_{\dots\text{one of its jumps } \dots} \underbrace{(m_{\mathcal{H}_i} = m \wedge m'_{\mathcal{H}_i} = m' \wedge p_j(\vec{x}, \vec{x}', \vec{c}))}_{\dots\text{must be taken and its predicate must hold.}} \Big) \Big) \qquad (2.2) \\
\wedge \; & \underbrace{\left(\neg flow \Rightarrow \bigwedge_{i \in \{1,\dots,z\}} (\neg jump_i \Rightarrow (\vec{x}'_{\mathcal{H}_i} = \vec{x}_{\mathcal{H}_i}))\right)}_{\text{All stuttering components must keep their valuations constant.}} \\
\wedge \; & \underbrace{\bigwedge_{i \in \{1,\dots,z\}} \bigwedge_{m \in M_{\mathcal{H}_i}} \Big((m'_{\mathcal{H}_i} = m) \Rightarrow I_m(\vec{x}', \vec{c}') \Big)}_{\text{In all automata, in all modes, the mode invariants must always hold.}} \qquad (2.3) \\
\wedge \; & \underbrace{\Bigg(flow \Rightarrow \Bigg(\bigwedge_{i \in \{1,\dots,z\}} \bigwedge_{m \in M_{\mathcal{H}_i}} \Big((m_{\mathcal{H}_i} = m) \Rightarrow}_{\text{During a flow, the active mode's } \dots} \\
& \underbrace{\dot{\vec{x}}_{\mathcal{H}_i} = f_m(\vec{x})}_{\dots\text{ODE and } \dots} \wedge \underbrace{I_m|_{\vec{x}}(\vec{x}(t))}_{\substack{\dots\text{projection of its} \\ \text{flow invariant to } \vec{x} \text{ must hold.}}} \Big) \Bigg) \Bigg) \qquad (2.4) \\
\wedge \; & \underbrace{\left(flow \Rightarrow \bigwedge_{i \in \{1,\dots,z\}} m'_{\mathcal{H}_i} = m_{\mathcal{H}_i} \right)}_{\text{Modes are constant during a flow.}}
\end{aligned}
$$

Note that for the predicate (2.2) the unabbreviated jump predicates p_j contain constraints of the form $x' = x$ for all non-changing variables. This is important for a predicative encoding, since not constraining a variable implies that its value can be chosen freely.

The predicate (2.3) is given only over the primed state s' because of the BMC unwinding (2.1). For the instance s_0, $init(s_0)$ already enforces that invariants

hold. The predicate (2.3) in the transition relation $trans(s_i, s_{i+1})$ thus only has to enforce that the invariant also holds for all future instances, but does not need to duplicate it for the instance s_0.

Finally, the predicate (2.4) introduces two kinds of constraints, that have so far not been covered in our definition of constraints and predicates: ODE constraints in which the occurring left-hand-side continuous variable is marked by the derivation operator, e.g. $\dot{x}$, and flow invariant constraints, in which the continuous variable is marked by $x(t)$. These constraint types directly import their semantics from Definition 6. Before formally defining satisfaction for ODE and flow invariant constraints, we apply the approach to the running example.

Running Example (continued). First, we choose a concrete instance of the model from Figure 2.1 by selecting values for the constants $D_B = 10, D_R = 15, A_{MAX1} = A_{MAX2} = 2, A_{B1} = A_{B2} = -2, V_{MAX} = 5$. To begin the encoding, we introduce the following variables and domains:

$$\begin{aligned}
m_{\mathcal{A}_1} &\in \{FREE_CRUISE_1, BRAKING_1, STOPPED_1\},\\
m_{\mathcal{A}_2} &\in \{FREE_CRUISE_2\},\\
p_1, p_2 &\in \mathbb{R},\\
v_1, v_2 &\in [0, V_{MAX}],\\
a_1, a_2 &\in [A_{B1}, A_{MAX}],\\
d &\in [0, \infty),\\
jump_1, jump_2, flow &\in \{0, 1\},\\
t &\in [0, T], \text{and}\\
\delta &\in [0, \Delta],
\end{aligned}$$

hence $s = (m_{\mathcal{A}_1}, m_{\mathcal{A}_2}, p_1, v_1, a_1, d, p_2, v_2, a_2, jump_1, jump_2, flow, t, \delta)$. Later, our approach will require us to introduce bounded domains for all variables. This will require some additional modeling insight or explicit choices, which we postpone for the moment. We have, however, already chosen tight domains for the velocities and accelerations, since these can be deduced quite easily from the model in this case. Also the maximum duration Δ of a flow and the maximum aggregated duration T must be chosen before an analysis can take place. Since these are not relevant for the encoding, we also postpone a decision about appropriate values for these parameters.

Over these variables, we define $init(s)$ as instructed above:

$$\begin{aligned}
init(s) \equiv \quad & m_{\mathcal{A}_1} = FREE_CRUISE_1\\
& \qquad \wedge (p_1 \leq p_2 - D_R \wedge d = p_2 - p_1 \wedge a_1 \in [A_{B1}, A_{MAX1}]\\
& \qquad \wedge v_1 \leq V_{MAX} \wedge v_1 \geq 0 \wedge d \geq D_B)\\
& \wedge m_{\mathcal{A}_2} = FREE_CRUISE_2\\
& \qquad \wedge a_2 \in [A_{B_2}, A_{MAX2}] \wedge v_2 \leq V_{MAX} \wedge v_2 \geq 0\\
& \wedge t = 0
\end{aligned}$$

Applying the encoding strategy to the continuous and discrete transitions of

the system, yields the following $trans(s, s')$ predicate:

$$
\begin{aligned}
trans(s, s') \equiv \quad & t' = t + \delta \\
& \wedge (\neg flow \Rightarrow \delta = 0) \\
& \wedge (\neg flow \Rightarrow (jump_1 \vee jump_2)) \\
& \wedge (jump_1 \Rightarrow (\\
& \qquad (m_{\mathcal{A}_1} = FREE\_CRUISE\_1 \wedge m'_{\mathcal{A}_1} = FREE\_CRUISE\_1 \\
& \qquad \wedge a'_1 \in [A_{B1}, A_{MAX1}] \wedge p'_1 = p_1 \wedge v'_1 = v_1 \wedge d' = d) \\
& \qquad \vee (m_{\mathcal{A}_1} = FREE\_CRUISE\_1 \wedge m'_{\mathcal{A}_1} = BRAKING\_1 \\
& \qquad \wedge d \leq D_B \wedge a'_1 = A_{B1} \wedge p'_1 = p_1 \wedge v'_1 = v_1 \wedge d' = d) \\
& \qquad \vee (m_{\mathcal{A}_1} = BRAKING\_1 \wedge m'_{\mathcal{A}_1} = FREE\_CRUISE\_1 \\
& \qquad \wedge d \geq D_R \wedge a'_1 \in [A_{B1}, A_{MAX1}] \wedge p'_1 = p_1 \wedge v'_1 = v_1 \wedge d' = d) \\
& \qquad \vee (m_{\mathcal{A}_1} = BRAKING\_1 \wedge m'_{\mathcal{A}_1} = STOPPED\_1 \\
& \qquad \wedge v_1 \leq 0 \wedge a'_1 = 0 \wedge p'_1 = p_1 \wedge v'_1 = v_1 \wedge d' = d))) \\
& \wedge (jump_2 \Rightarrow (\\
& \qquad (m_{\mathcal{A}_1} = FREE\_CRUISE\_2 \wedge m'_{\mathcal{A}_1} = FREE\_CRUISE\_2 \\
& \qquad \wedge a'_2 \in [A_{B2}, A_{MAX2}] \wedge p'_2 = p_2 \wedge v'_2 = v_2))) \\
& \wedge (\neg flow \Rightarrow (\\
& \qquad (\neg jump_1 \Rightarrow (p'_1 = p_1 \wedge v'_1 = v_1 \wedge a'_1 = a_1 \wedge d' = d)) \\
& \qquad \wedge (\neg jump_2 \Rightarrow (p'_2 = p_2 \wedge v'_2 = v_2 \wedge a'_2 = a_2)))) \\
& \wedge (m'_{\mathcal{A}_1} = FREE\_CRUISE\_1 \Rightarrow (v_1 \leq V_{MAX} \wedge v_1 \geq 0 \wedge d \geq D_B)) \\
& \wedge (m'_{\mathcal{A}_1} = BRAKING\_1 \Rightarrow (v_1 \geq 0)) \\
& \wedge (m'_{\mathcal{A}_2} = FREE\_CRUISE\_2 \Rightarrow (v_2 \leq V_{MAX} \wedge v_2 \geq 0)) \\
& \wedge (flow \Rightarrow \\
& \qquad (m_{\mathcal{A}_1} = FREE\_CRUISE\_1 \Rightarrow (\\
& \qquad \dot{p}_1 = v_1 \wedge \dot{v}_1 = a_1 \wedge \dot{a}_1 = 0 \wedge \dot{d} = v_2 - v_1 \\
& \qquad \wedge v_1(t) \leq V_{MAX} \wedge v_1(t) \geq 0 \wedge d(t) \geq D_B)) \\
& \qquad \wedge (m_{\mathcal{A}_1} = BRAKING\_1 \Rightarrow (\\
& \qquad \dot{p}_1 = v_1 \wedge \dot{v}_1 = a_1 \wedge \dot{a}_1 = 0 \wedge \dot{d} = v_2 - v_1 \\
& \qquad \wedge v_1(t) \geq 0)) \\
& \qquad \wedge (m_{\mathcal{A}_1} = STOPPED\_1 \Rightarrow (\\
& \qquad \dot{p}_1 = v_1 \wedge \dot{v}_1 = a_1 \wedge \dot{a}_1 = 0 \wedge \dot{d} = v_2 - v_1)) \\
& \qquad \wedge (m_{\mathcal{A}_2} = FREE\_CRUISE\_2 \Rightarrow (\\
& \qquad \dot{p}_2 = v_2 \wedge \dot{v}_2 = a_2 \wedge \dot{a}_2 = 0 \\
& \qquad \wedge v_2(t) \leq V_{MAX} \wedge v_2(t) \geq 0))) \\
& \wedge (flow \Rightarrow (m'_{\mathcal{A}_1} = m_{\mathcal{A}_1} \wedge m'_{\mathcal{A}_2} = m_{\mathcal{A}_2}))
\end{aligned}
$$

The predicate $target(s)$ can be chosen arbitrarily to define the state of the system, whose reachability is to be analyzed. A most simple choice could be $target(s) \equiv$ true, which would mean that for any trace $\mathcal{S}_0, \ldots, \mathcal{S}_k$, there would exist a solution $s_0, \ldots, s_k$ for the k-fold unwinding of the transition system as

shown in Equation (2.1). Conversely, choosing $target(s) \equiv$ false would make the BMC formula trivially unsatisfiable and hence there would not be a single trace of the system, for which we could find a corresponding solution. A much more interesting choice would be $target(s) \equiv d \leq 1$. If this target condition is used in (2.1), solutions would only exist for traces of length k which finally reach a near-collision distance $d \leq 1$ in their last step. Such traces, which identify undesired situations, are obviously useful to falsify the safety of the system and help fixing e.g. the choice of parameters that allows the undesired behavior. Again, we should stress that unsatisfiability of (2.1) means that no trace of length k exists, that ends in a near-collision situation. However, nothing can be deduced about traces of length $k+1$, unless by some other argument it can be shown that no new (undesired) behavior can occur for larger unwindings than k.

Size of the Encoding for Parallel Compositions. As can be seen from the general rules and their application to our example, an important quality of the encoding is that there is no need for an explicit construction of a *product automaton*. More importantly, nowhere within the encoding is there a need to build any cross-product over modes or variables of the component automata. In fact, it is often argued (e.g. in [FHR+07, Her11]) that BMC encodings allow parallel composition by mere conjunction of the individual transition systems. For the parallel composition of independent programs (i.e. no communication, no shared state, and no continuous evolution) it has already been known since at least 1984 that parallel composition can be represented by mere conjunction of the predicates that encode the behavior of the component programs [Heh84]. While this may not be true in the strictest sense for our encoding (we still need to extend e.g. the predicate that enforces $\neg flow \Rightarrow \bigvee_{i=1}^{z} jump_i$), the fundamental message that an explosion of the representation size—that would be expected for a parallel composition based on explicit product-automaton generation—can be avoided. Unfortunately, but rather naturally, and this will later be addressed in more detail, the price for the additional complexity that comes with the interaction and interleaving of parallel components still has to be paid. In our case, it comes in the form of a larger number of variables and the state space they span for the solver to search for solutions in.

2.3 Satisfiability Modulo ODE

Starting with hybrid automata and introducing an encoding that creates formulae which are satisfiable by valuations that represent traces of the encoded automata has led us to the essence of the problem this thesis tries to address: solving formulae over discrete and continuous variables involving non-linear arithmetics and ODEs. We have introduced this problem with the same motivation in [EFH08], calling it *SAT modulo ODE*, and extended it among others by the introduction of flow invariant constraints in [ERNF12a].

The propositional satisfiability problem (SAT) can safely be considered “one of the classical problems in computer science” [CESS08]. Within that paper, Claessen et. al. first give a brief overview of SAT as a theoretical problem and then focus on the “SAT revolution” that was caused by the advent of bounded model checking coinciding with the availability of a first “high-performance SAT

solver" and a decade-long history of improvements of SAT solving algorithms. While SAT deals with formulae consisting of the usual boolean connectives over propositional variables and their negation as *atoms* (often called *literals*), *SAT modulo Theories (SMT)* lifts the restriction on what constitutes an atom in the formula. In [BSST09], Barret et. al. provide a survey of SMT over a wide variety of theories and of the underlying theoretical foundations. Intuitively, like a SAT formula, an SMT formula is satisfiable if and only if there exists a valuation that satisfies a set of atoms from the formula which together satisfy the boolean structure and hence the formula as such. However, while the question, whether an individual atom is satisfied, is close to trivial in the case of propositional formulae, it becomes far more intriguing when the formula consists of atoms from one or several *background theories*. Among the "theories of interest" enumerated by Barret et. al. are reasoning about *equality with uninterpreted functions* (e.g. deducing that when the two atoms $f(x) = y$ and $f(z) = y$ hold, so must $x = z$), various types of restricted *arithmetic*, most prominently linear constraints like $5x + 7y - 2.1z \leq 4.2$, and *bit-vectors*, which allow more compact representations of operations over boolean variables. The focus of the survey thus lies on combining SAT solving with *decidable* theories.

In contrast to these decidable theories, in [FHR+07], the problem of satisfiability modulo non-linear arithmetics over boolean, integer, and real-valued variables involving transcendental functions has been investigated, accepting the incompleteness of the method due to the undecidability introduced by the rich arithmetics. Adding to this problem class ODE constraints and flow invariants will obviously not bring us back into the domain of a decidable SMT problem. However, theoretical research in the direction of *robustness* for hybrid systems [Frä99], *approximation metrics* [GP07] exploiting similarity of neighboring trajectories to simplify verification, *quasi-decidability* [FRZ11] for systems of equations over real analytic functions, and most recently *δ-completeness* [GKC13] has introduced notions that are only slightly weaker than traditional decidability.

In [GKC13], Gao et al. have explicitly targeted the class of Satisfiability modulo ODE problems. They define a δ-complete "decision" procedure as one that only needs to detect unsatisfiability if a formula stays false even if the right-hand-side constants of all constraints (normalized such that the relational operators are from $\{>, \geq\}$ and the right-hand side is zero) are changed to $-\delta$. Due to this modification, a valuation still satisfies this relaxed constraint system if it violates each original constraint by at most δ. By allowing the solver to give an arbitrary answer if the relaxed constraint system becomes satisfiable while the original formula was not, the notion of δ-completeness is weaker than actual decidability. The price that needs to be paid for achieving δ-completeness is, however, that all pruning operators, which are used to remove non-solutions from the search space, must be able to provide δ accuracy. If rewriting of a formula's complex expressions into simpler constraints is taken into account, this may require even significantly higher accuracy to mitigate the effect that also the constraints added for freshly introduced auxiliary variables will be relaxed by δ. This requirement of being able to produce arbitrarily fine overapproximations of arithmetic and of ODE constraints is a significant burden for a practical implementation of δ-complete algorithms.

We expect that future work in this direction will further bridge the gap between a theoretically desirable decidability notion and limitations induced by

practical numerics.

Definition 8 (SAT modulo ODE formula)**.** With only cosmetic modifications, we repeat our definition from [ERNF12a]. A SAT modulo ODE formula is a quantifier-free boolean combination of arithmetic constraints over real-, integer-, and boolean-valued variables with bounded domains, simple bounds, ODE constraints over real variables, and flow invariants with the following properties:

- arithmetic constraints over variables x, y, and z are of the form $x \sim \circ(y, z)$ or $x \sim \circ(y)$, where $\sim$ is a relational operator from $\{<, \leq, =, \geq, >\}$, and $\circ$ is a total unary or binary operator from $\{+, -, \cdot, \sin, \cos, \mathrm{pow}_{\mathbb{N}}, \exp, \min, \max\}$, with $\mathrm{pow}_{\mathbb{N}}$ denoting a power with constant positive integer exponent and the other operators having their usual meanings;
- simple bounds are of the form $x \sim c$ with $\sim$ as above a relational operator, x a variable, and $c \in \mathbb{Q}$ a constant;
- ODE constraints are given by $\dot{x}_i = dx_i/dt = f(x_1, \ldots, x_n)$ with all occurring variables x_i themselves again being defined by ODE constraints and f being a function composed of $\{+, -, \cdot, /, \mathrm{pow}_{\mathbb{N}}, \exp, \ln, \sqrt[\mathbb{N}]{}, \sin, \cos\}$ (with $\sqrt[\mathbb{N}]{}$ denoting roots with positive integer index); and
- flow invariant constraints are of the form $x(t) \leq c$ or $x(t) \geq c$ with x being an ODE-defined variable and $c \in \mathbb{Q}$ being a constant.

A SAT modulo ODE formula is specified by a variable declaration, which introduces the domain for each variable, and by an initial, transition, and a target predicate over the declared variables. The actual formula is then obtained by a k-fold instantiation of the variables and unwinding of the transition predicate in the BMC fashion as introduced in Equation (2.1). ODE constraints and flow invariants must only occur within the transition predicate and only under an even number of negations, allowing e.g. an implication like $m_1 \Rightarrow ((\dot{x} = \sin(y)) \wedge (\dot{y} = -x))$, but forbidding e.g. $(\dot{x} = \sin(y)) \Rightarrow m_1$ to avoid subtleties in the semantics of the formula. A special variable t denotes the global time and a special variable δ denotes the duration of each transition.

The semantics of a Satisfiability modulo ODE formula is anchored in the notion of a satisfying valuation for it, and this notion of satisfiability is naturally chosen to be compatible with the trace semantics for hybrid automata, that has dominated the previous sections.

Definition 9 (satisfiability of SAT modulo ODE formulae)**.** A SAT modulo ODE formula Φ over variables $\vec{v}$ is satisfiable if and only if there exists a point in $\mathrm{dom}(\vec{v})$, which satisfies Φ. For a valuation σ to satisfy Φ, there must be a combination of atoms in Φ that is satisfied by σ, such that the boolean structure of Φ is satisfied. An arithmetic atom is satisfied by σ if the evaluation of the left- and right-hand-side expressions under σ satisfies the constraint's relational operator or—in case the atom only consists of a boolean literal—this literal evaluates to true under σ. We can safely consider the details of this part of the semantics to be well known and standard, and refer e.g. to [Her11] for a more in-depth definition.

Essential for the extended problem class is the question of satisfying ODE constraints and flow invariants. As defined above, the variables of Φ are actually

k instances that were introduced for the unwinding of the transition system. We call these sub-vectors of variables $(\vec{v}_0, \ldots, \vec{v}_k) = \vec{v}$ and split σ accordingly into $\sigma_0, \ldots, \sigma_k$. Additionally, we call a set of ODE constraints from the same unwinding depth $i \in \{0, \ldots, k\}$ *definitionally closed* if and only if each variable occurring on a right-hand side of one of the ODE constraints in the set occurs exactly once as a left-hand-side variable, i.e. its slope is defined by an ODE constraint as well. Such a set of ODE constraints over variables $\vec{x}$ can then easily be rewritten as an ODE system $\dot{\vec{x}} = \vec{f}(\vec{x})$. For depth i, we use σ_i as valuation; similarly, we use σ_{i+1} as valuation for the successor depth. This depth $i+1$ is always less or equal to k since we have restricted ODEs to occur only within the transition system and not within the target predicate.

We then call an ODE system of dimensionality n on unwinding depth i satisfied if and only if there exists a solution function $\vec{y} : [0, \delta_i] \to \mathbb{R}^n$ of duration δ_i with the right starting point $\sigma_i(\vec{x}) = \vec{y}(0)$, the right slope for all points in between, i.e. $\forall \tau \in [0, \delta] : \vec{y}(\tau) = \vec{f}(\vec{y}(\tau))$, and the right end point $\sigma_{i+1}(\vec{x}) = \vec{y}(\delta)$.

A flow invariant constraint $x_j(t) \sim c$ on the same depth i is satisfied if and only if the j-th dimension of the solution function never violates the constraint, i.e. $\forall \tau \in [0, \delta] : y_j(t) \sim c$ holds.

By construction, this formula class contains sufficiently powerful atoms to directly perform the encoding that we have introduced in the previous section. Obviously, some of the choices made in these definitions are tailored specifically to the intended usage scenario, especially the requirement that the formula be written as a combination of initial, transition, and target predicate. A more general definition could easily be made if variables were explicitly marked as being connected by ODEs instead of relying on unwindings and the resulting pre- and post-instances. Such a generalization might in fact be useful in other contexts or even for model checking approaches that do not follow the strict unwinding scheme, we have pursued in this thesis.

Example 1 (SAT modulo ODE formula [ERNF12a])**.** In Figure 2.4, we illustrate the encoding and the formula written in the syntax of our iSAT-ODE tool on an originally non-BMC-oriented model from [GMEH10]. This problem can be stated as follows: find two points A and B on a circle with radius 1 around $(1, 0)$ and from the box $[-1, 1] \times [-1, 1]$, such that a trajectory of a harmonic oscillator around $(0, 0)$ with fixed temporal length (here, we choose 1), starting in A ends in a point X, forming an equilateral triangle A, B, X. The special variables t and δ are represented by variables *time* and *delta_time* in the input syntax. On the right side, the figure also contains a graphical depiction of a satisfying valuation—represented by the three points. A simulated trajectory leading from point A to point X validates the solution. In the example, no flow invariants have been used, not even to enforce the variable domains. As a result, the connecting trajectory between A and X is not forced to stay within $[-1, 1] \times [-1, 1]$. Were these flow invariants added, the illustrated valuation would no longer be a solution of the formula.

Running Example (continued). We provide the actual encoding of the running example as a SAT modulo ODE formula in the input language used by our tool in Figure 2.5. In the first part (lines 1–32), some symbolic names for numerical constants and the variables of the system with their domains are declared. Also the declarations of the special variables *time* for the global time t and *delta_time*

```
DECL
 float [−1, 1] ax, ay, bx, by;
 float [−10, 10] x, y;
 float [0, 10] time;
 float [1, 1]  delta_time;
INIT
 −− A and B on circle around (1,0).
 (ay−0)^2 + (ax−1)^2 = 1; (by−0)^2 + (bx−1)^2 = 1;
 −− A and B must be distinct points.
 ax != bx or ay != by;
 −− Trajectory must start in A.
 x = ax; y = ay;
 time = 0;
TRANS
 −− A and B stay the same.
 ax' = ax; ay' = ay; bx' = bx; by' = by;
 −− Trajectory.
 (d.x / d.time =  y); (d.y / d.time = −x);
 time' = time + delta_time;
TARGET
 −− Equilateral triangle: equal distances between points.
 (ay−by)^2 + (ax−bx)^2 = (ax−x)^2 + (ay−y)^2;
 (ay−by)^2 + (ax−bx)^2 = (bx−x)^2 + (by−y)^2;
```

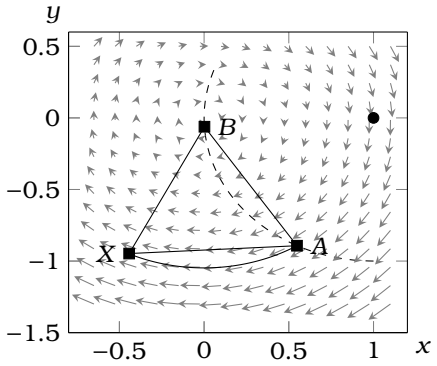


Figure 2.4: Example of an iSAT-ODE input (before being automatically rewritten into the solver's internal format by its frontend). The right graph shows a candidate solution that has been found by the solver, illustrating a satisfying valuation of a one-step unwinding of this constraint system. (Based on our previous publication in [ERNF12a].)

for the duration δ introduce bounds, which limit the duration of each BMC step and the total duration of any trajectory that may be a solution. Choosing these bounds may be a non-trivial task in general, but if no a-priori choice is possible, the model can be extended by self-loops in such a way that it is always possible to glue together two shorter flows by an intermediate jump that does not change anything. As a minor deviation from the strict encoding scheme applied in the previous section, no mode variable for the second automaton is introduced since its value would never change.

In lines 33–41, the initial predicate encodes what constitutes an admissible initial state of the system. Again, some constraints have been left out since they are redundant, e.g. $a_2 \in [A_{B_2}, A_{MAX2}]$, which is already implied by the chosen variable domain.

The largest part of the encoding is spent on the transition relation. It begins with general constraints (lines 43–49), which enforce that the global time t accumulates the durations δ_i, jumps do not take time, and modes are kept constant during a flow. A deviation from the encoding scheme lies in the constraint in line 46, which requires that flows and jumps are strictly alternating. This condition is sometimes useful, especially if it is known that trajectories without strict alternation of discrete and continuous transitions do not lead to any additional interesting system states. In lines 51–60, the continuous evolution is encoded by ODE constraints. It is beneficial for technical and for readability reasons to write each ODE constraint only once and guard it with a disjunction of all the modes in which it is active—or, as is possible in this example, to even enforce them globally for each flow if they are the same in all modes. Lines 63–76 constrain the continuous trajectories by mode-dependent flow invariants. The predicate $\neg \mathit{flow} \Rightarrow \mathit{jump}_1 \vee \mathit{jump}_2$ in line 83 explicitly

```
DECL
    define BRAKE_DIST = 10;
    define RELEASE_BRAKE_DIST = 15;
    define MAX_ACCELERATION = 2;
    define MAX_SPEED = 5;
    define BRAKE_ACC = -2;

    define FREE_CRUISE = 0;
    define BRAKING = 1;
    define STOPPED = 2;

    -- Modes, posititions, speeds,
    -- and accelerations.
    int [0,2] m1;
    float [-1000, 1000] pos1, pos2;
    float [0, MAX_SPEED] speed1, speed2;
    float [BRAKE_ACC,
           MAX_ACCELERATION] acc1, acc2;

    float [0, 2000] distance;

    define MAX_TIME = 10;
    define MAX_DELTA_TIME = MAX_TIME;

    float [0, MAX_TIME] time;
    float [0, MAX_DELTA_TIME] delta_time;

    boole flow;
    -- If one of the parallel components performs
    -- a mode switch, its jump variable must be
    -- true during !flow.
    boole jump1, jump2;
INIT
    time = 0;
    -- The follower car1 starts in FREE_CRUISE
    -- mode with safe distance to the lead car.
    m1 = FREE_CRUISE;
    pos2 >= pos1 + RELEASE_BRAKE_DIST;
    -- Leading car2 is always in FREE_CRUISE
    -- mode and is ahead of the follower car.
    distance = pos2 - pos1;
TRANS
    -- Time progress.
    time' = time + delta_time;
    -- Strict alternations of jumps and flows.
    flow <-> !flow';
    !flow -> delta_time = 0;
    -- Mode does not change during flow.
    flow -> m1' = m1;

    -- Dynamics of system dimensions.
    flow -> (d.pos1 / d.time = speed1);
    flow -> (d.speed1 / d.time = acc1);
    flow -> (d.acc1 / d.time = 0);
    flow ->
       (d.distance / d.time = speed2 - speed1);

    flow -> (d.pos2 / d.time = speed2);
    flow -> (d.speed2 / d.time = acc2);
    flow -> (d.acc2 / d.time = 0);


    -- May stay in FREE_CRUISE as long as
    -- neither braking distance nor maximum
    -- speed are reached.
    flow and m1 = FREE_CRUISE
        -> (    distance(time) >= BRAKE_DIST
            and speed1(time) <= MAX_SPEED
            and speed1(time) >= 0);

    -- Mode BRAKING as long as speed >= 0.
    flow and m1 = BRAKING -> speed1(time) >= 0;

    -- Flow invariants for car2.
    flow -> (    speed2(time) >= 0
            and speed2(time) <= MAX_SPEED);

    -- Jumps.

    -- Note that multiple jumps can occur
    -- simultaneously or components can perform
    -- jumps while the other ones do not change.
    !flow <-> jump1 or jump2;

    -- Common for all jumps.
    !flow ->    distance' = distance
            and pos1' = pos1
            and pos2' = pos2
            and speed1' = speed1
            and speed2' = speed2;

    -- Keep all remaining variables constant if
    -- component does not perform jump.
    !flow and !jump1 -> acc1' = acc1 and m1' = m1;
    !flow and !jump2 -> acc2' = acc2;

    -- Individual jumps for car1. Self-loop at
    -- FREE_CRUISE: allow arbitrary acceleration.
    jump1 and m1 = FREE_CRUISE
          and m1' = FREE_CRUISE
        ->          acc1' >= BRAKE_ACC
              and acc1' <= MAX_ACCELERATION;

    -- Jump from FREE_CRUISE to BRAKING.
    jump1 and m1 = FREE_CRUISE
          and m1' = BRAKING
        ->    distance <= BRAKE_DIST
          and acc1' = BRAKE_ACC;

    -- Jump from BRAKING to STOPPED.
    jump1 and m1 = BRAKING
          and m1' = STOPPED
        ->    speed1 = 0
          and acc1' = 0;

    -- Jump from BRAKING to FREE_CRUISE: again
    -- arbitrary acceleration.
    jump1 and m1 = BRAKING
          and m1' = FREE_CRUISE
        ->    distance >= RELEASE_BRAKE_DIST
          and acc1' >= BRAKE_ACC
          and acc1' <= MAX_ACCELERATION;

    -- Define allowed jumps.
    jump1 ->   m1 = FREE_CRUISE
             and m1' = FREE_CRUISE
      or m1 = FREE_CRUISE and m1' = BRAKING
      or m1 = BRAKING and m1' = FREE_CRUISE
      or m1 = BRAKING and m1' = STOPPED;

    -- Car2 is always in FREE_CRUISE and can
    -- therefore change its acceleration freely.
    -- The flow invariant will avoid negative or
    -- excessive positive velocities.
    jump2 ->  (   acc2' >= BRAKE_ACC
              and acc2' <= MAX_ACCELERATION);
TARGET
    -- Find a near-collision situation.
    distance <= 1;
```

Figure 2.5: Encoding of the model from Figure 2.1 as an iSAT-ODE input.

encodes that at least one component must actually perform a jump in each non-flow step. Extracting from the jumps all variables that are kept constant (lines 85–90) compactifies the remaining individual jump encodings up to the end of the transition predicate (line 136).

Finally, the target predicate at the end describes a state, in which the distance variable reaches a critically small value. A solution of a k-fold unwinding of this formula would hence have to reach such a small distance after having started from a much larger initial separation between the vehicles. Such a solution—if not an artifact of a modeling error—would expose an interesting behavior of the analyzed system, that probably should be investigated further and may be used to tune some of the system's parameters.

2.4 Related Work

Before looking at our algorithmic approach to solving SAT modulo ODE problems and at experimental results in the chapters to come, a step back to a wider perspective is now in order and helps to understand the work related to this thesis. Our core motivation is the prospect of a tool for the automatic analysis of hybrid systems. Clearly, we are neither the first nor the last to pursue this goal. *Simulation* has reached the stadium of industrial maturity as has of course *testing of physical prototypes*, which has been in use much longer than people have been thinking of their systems as hybrid. Both play an important practical role in building up a developer's and a client's trust in the actual system under construction (cf. e.g. [Mar11]).

Reachability Computation. *Model checking* approaches, which aim more closely at being fully automatic and at yielding mathematically-rigorous and complete results, have existed for hybrid systems at least since the development of HyTech [HHWT97]. HyTech is restricted to *linear hybrid automata* in the sense that the solution functions are linear, i.e. ODEs must be of the form $\dot{x} = c$ or $\dot{x} \in [c - \epsilon, c + \epsilon]$. The core algorithm in HyTech and many other model checkers iteratively constructs the entire space of reachable states until it either contains a target state or no new states can be added to the identified reachable state space because a fixed point has been reached. In [HPWT01] reasons for non-termination in practice are given as the size of the reachable state space exceeding available memory, impractical running times, countably infinite or non-linear reachable state sets, and arithmetic overflows caused by finite-precision representations. In the case of HyTech, the algorithm is based on polyhedra and their manipulation by the linear ODE solution functions and discrete jumps. Obviously, having a characterization of the entire reachable state space grants deep insight into a hybrid system's behavior since it allows e.g. to immediately decide for multiple target conditions whether they can be reached or by how large a margin they are missed. In some cases, it may even suffice to identify some form of stabilization or oscillation behavior. If a fixed point is actually reached by the analysis, the characterization is complete in the sense that it allows reasoning about all trajectories, including *unbounded* ones. Such desirable qualities have been the reason why a multitude of different geometric objects have been explored over nearly two decades for capturing reachable-states of hybrid systems.

In [Gir05] an approach was presented to compute the set of reachable states for linear ODEs (and the hybrid systems in which they occur) using *zonotopes*, a form of polytopes, which are closed under linear transformation and Minkowski sum. This approach makes use of the matrix exponential to compute near-overapproximations of the solution sets, which are represented by sequences (or unions) of zonotopes in what is often called a *flowpipe*. The actual computations however are done via standard approximative numerics, leaving potential to miss some solutions in the enclosure. Old [OM88, BM89] and more recent [Gol09] work on computing the matrix exponential for interval matrices could however mitigate this shortcoming.

We note briefly, that more general *polytopes* have successfully been used in [HK06] and so-called *ellipsoidal approximations* even earlier in [BT00]. The variety of methods is further diversified by *Bézier curves* [Dan06] adapted from computer graphics and applied to the analysis of hybrid systems with polynomial ODEs. More recently in [DT12], their basis, *Bernstein expansions*, have been combined with *template polyhedra* and used to improve the scalability of the Bézier approach in the context of polynomial discrete-time dynamical systems—with future applications to hybrid systems to be expected.

Hybridization. One core argument, why an analysis tool's restrictions of the continuous dynamics are not a major obstacle to the verification of more complex hybrid systems, is given by Asarin et al. in [ADG03]. Replacing a system of non-linear ODEs in one mode by a number of modes in which the original dynamic is overapproximated e.g. by linear ODEs and a sufficiently-large error term, the original dynamics is subjected to what is called *hybridization*. Such an introduction of additional modes shifts some of the analysis complexity from the continuous flows to the discrete mode changes. Additionally, it may be hard to find a suitable trade-off between the degree of conservativeness in the overapproximation and the number of modes that need to be introduced. Clever strategies for the division of the continuous state space are necessary in all but the lowest-dimensional systems to avoid the exponential blow-up, which is caused by naively subdividing each variable's domain.

One approach to mitigate this problem is proposed in [DLGM09] as *dynamic hybridization*. The central idea is to construct the hybridization during the reachability analysis: whenever the reachable successor states computed by the post-operator have partially left the cell of the hybridization, the last step is taken back, the cell shifted into the direction where it has been left, and the reachability computation repeated with the newly computed (linear) overapproximation for this shifted cell. The underlying idea of first computing the reachable states and a-posteriori checking the computation's validity is in fact also used in our approach for computing bracketing systems (cf. Sect. 3.5).

In [Her11, Chapter 6], a method is suggested to perform *hybridizations on the fly inside bounded model checking* with an iSAT-like solver. This would allow to automatize the trade-off between overestimation and discrete state-space complexity. A general problem with this approach in the context of bounded model checking would be the increase of trace lengths, which is induced by the introduction of additional modes, through which a previously uninterrupted continuous trajectory needs to pass in the hybridized system.

Application of Hybridization in Model Checking. In the PHAVer [Fre05] model checker, automatic hybridization is used to generate piecewise-linear solution functions for *affine dynamics*, i.e. ODEs with linear right-hand sides. This approach is based on an earlier understanding of hybridization, called *linear phase-portrait approximation* [HHWT98], which overapproximates the solution functions of linear ODEs like e.g. $\dot{x} = f(x) = 2x$—which naturally may contain trigonometric and exponential terms—by linear enclosure functions like $x(t_0 + h) \geq x(t_0) + h \cdot a$ and $x(t_0 + h) \leq x(t_0) + h \cdot b$, which are the solutions of constant differential inclusions of the form $\dot{x} \in [a, b]$. By splitting the continuous state space into cells and finding in each cell appropriate a and b such that they cover the possible values of $f(x) = 2x$, the original solution function is enclosed within that cell. PHAVer performs the necessary partitioning recursively according to user-defined parameters and combines this hybridization approach with a fixed-point computation similar to the one of HyTech. One of its major improvements is to limit the precision of the computed representations of rational coefficients, whose growth was found to be problematic in HyTech. An additional improvement lies in the deliberate introduction of imprecision by combining computed elements of the reach set into fewer—hence larger—and simpler polyhedra, which leads to more compact representations of the reachable state set and at the same time may avoid infinitesimally small progress during the fixed-point computation. Both advantages come at the cost of a more conservative estimate of the reachable state space, potentially introducing spuriously non-empty intersections with unsafe target states.

PHAVer's successor, SpaceEx [FLGD+11], combines *template polyhedra* with *support functions* as an alternative representation of reachable states. In this context, support functions can be used as exact representations of convex sets with finitely many directions stemming from the template polyhedra representation and allow for efficient applications of linear maps, Minkowski sum, and the computation of the convex hull of several sets, whereas template polyhedra allow efficient computation of intersections and deciding containment. SpaceEx therefore uses conversions between the two representations—again incurring some overapproximation—to have efficient versions of all these operations, from which its reachable state computation for hybrid systems with affine dynamics and nondeterministic inputs is composed.

Comparison with our Approach. Contrasting these reachability-based methods to our approach of solving a SAT modulo ODE formula, we observe the following central differences. While the goal of tools like SpaceEx is to characterize the *entire* reachable state space, our approach is focused on finding only *one* step-bounded trajectory of the hybrid system under analysis or proving its non-existence. While the latter may amount to the same work—after all, the entire state space may have to be analyzed to prove that no trajectory leads to a target state—in principle, our goal is significantly less ambitious. Having found one trajectory, by luck or by the virtue of well-crafted search heuristics, we can stop. Also the *step-boundedness* of BMC is a significant difference: without an additional argument, our approach is unable to prove anything about trajectories that are longer than the threshold for the number of steps that has been chosen. On the positive side, from the solution of a SAT modulo ODE formula, it is trivial to extract the values of each variable at each step,

including e.g. uncertain inputs. Constructing an actual trajectory after having found a non-empty intersection between reachable and target states may, on the other hand, require additional computations to reconstruct the concrete values which need to be chosen at each step.

The technical differences are even more striking. Reachable state computation for hybrid systems is about computing successor states from what is known to be reachable, intersecting them with guards, and applying actions to the intersection result. Our approach starts with the entire state space as a possible valuation and tries to prune off parts that are known to be inconsistent under the given formula. Splitting the remaining consistent state space and generalizing observed inconsistencies, the solver tries to close-in on a solution and uses a backtracking search to escape non-satisfying parts of the state space. While the one is like constructing step-by-step a reachability frontier, the other can better be described as carving a single trajectory from a large domain full of inconsistent points.

Due to these unequal goals and different technical approaches, the direct comparison of results is complicated. While it is possible to say which models and specifications can be analyzed by the one tool or the other, the results of a successful analysis differ significantly. If a target is unreachable, the reachability-based approaches give a more detailed result by showing the entire reachable state space instead of just the proof that the analyzed formula is unsatisfiable. Otherwise, if the target can actually be reached, the satisfiability-based approach computes a concrete trace leading to the target, instead of just showing that the intersection of reachable and target states is non-empty. While in some applications either result may be acceptable, often one would be preferable over the other.

In the following, after introducing some more context and background for our own approach, we discuss more direct competitor tools. Part of our experimental evaluation is concentrated on a comparison with these, which indirectly also yields a comparison with the reachability-computation-based model checking algorithms above.

Lyapunov Stability Proofs. An important class of real-world problems involves the question of whether a system stabilizes towards equilibrium points or attractor regions. To address this question, the *direct Lyapunov method*, which is used to prove stabilization in the domain of continuous systems, has been lifted to hybrid systems [Oeh11]. One central issue is the automatic generation of suitable Lyapunov functions, for which template candidates and algorithmic fitting of coefficients can be employed—successfully at least in the case of linear ODEs (and generalized to some degree beyond this class). To prove stability in case of hybrid systems, the approach extends the Lyapunov argument from the continuous trajectories along which the Lyapunov function needs to decrease, also to the relevant jumps of the hybrid system, such that they can be shown to equally reduce the system's "energy". Scalability of the approach is improved by the use of hierarchical proof schemes based on automatized system decompositions into smaller components.

In one of our experiments (cf. Section 4.1.1), we apply our method to prove stabilization of a hybrid system towards an attractor region. In general, however, Lyapunov methods are a more suitable tool if stabilization is to be shown

rather than reachability, since they allow reasoning without having to solve or overapproximate the differential equations. On the other hand, if stabilization properties of a system are known, e.g. from a proof based on Lyapunov methods, this information can be used to improve the system's encoding (for example by making it explicit that a trajectory can never leave a certain mode after it has entered it while being close to an attractor that is catching the trajectory).

Validated Numerical Integration of ODEs. In an earlier attempt to improve HyTech, HyperTech [HHMWT00] was introduced as a successor tool, which utilized *interval arithmetics* to compute enclosures for the solution sets of ODEs, enabling the tool to compute overapproximations of the set of reachable states for *hybrid systems with non-linear ODEs*. To the best of our knowledge, HyperTech was the first tool to apply methods from the domain of *validated numerical integration of ODEs* in the context of hybrid systems model checking. In particular, HyperTech used ADIODES (Automatic Differentiation Interval Ordinary Differential Equation Solver) [Sta97] to compute enclosures of the ODE solution sets. While our approach differs significantly from HyperTech in that we try to find solutions of constraint systems instead of computing the entire set of reachable states, the idea of using methods from interval arithmetics has significantly influenced our work.

In the 1960s, (the historically most recent incarnation of) interval analysis has been pioneered by Moore [Moo66] and has even then found one of its first applications in the computation of enclosures for set-valued initial problems of ODEs. Moore already detected that stepwise integration and use of axis-parallel boxes to enclose the solution sets could cause significant overapproximation. This *wrapping effect*, which is the result of having to compute not only the actual solution points, but also the successors of all those points, which were "wrapped" in together with the solution in each intermediate integration step, has been the topic of further research. Moore himself identified that coordinate transformations would help to reduce wrapping. One of Lohner's contributions [Loh88] was the additional use of the "QR-method" to orthogonalize the transformation matrix used for these coordinate transformations. Stauning [Sta97], whose work has been used in HyperTech, has provided a more modern implementation, which heavily uses C++ templates for *automatic differentiation*, parts of which (the FADBAD++ library) are still found in VNODE-LP [Ned06], in which the Taylor series expansion of the (unknown) solution is complemented by a Hermite Obreschkoff series. VNODE-LP also solves the long-standing problem of finding an a-priori solution for larger integration step sizes by using *higher order enclosures* (HOE). We will describe these techniques in more detail in the next chapter.

It may be noteworthy that the goal of these enclosure methods is similar to the goal pursued by the model checking algorithms that are based on reachability computations. Being able to compute over a finite temporal horizon an enclosure of all trajectories that emerge from a set of initial values, the validated numerics methods could adequately be called reachability tools for continuous systems. This similarity makes it less astonishing that the choice of geometric shapes is as important as in reachability computation. While the most natural shape for an interval-based tool is simply a box consisting of n intervals in the n dimensions of the problem, representations based on coordinate transformations implicitly

form (regular) polytopes—represented by a coordinate transformation matrix and intervals with respect to the sheared and rotated axes.

Taylor Models. A second line of methods to mitigate the wrapping effect addresses the shortcoming that whenever non-convex solution sets are enclosed by convex shapes, wrapping cannot be avoided. Real-world examples include the determination of object paths in particle accelerators [BM98] and checking for decades into the future, whether an asteroid may impact the earth [ADLBZB10]. These instances of non-linear ODE enclosure problems cause significant overapproximation when boxes with coordinate transformation are used.

In [MB03, p. 8ff], *Taylor models* implemented in the COSY-VI ("validated integrator") tool, are compared with Lohner's AWA (for "Anfangswertaufgabe" – initial value problem) [Loh88] using a *single use expression (SUE)* of the ODEs to reduce the *dependency problem*[3] and highlight issues caused by the wrapping effect. A detailed comparison is made on the example of the non-linear Volterra equations. While the Taylor-model-based enclosures can be used to tightly follow the solution sets over an entire oscillation period, the enclosure computed by AWA quickly deteriorates—making it impossible to compute an enclosure for an entire period. The detailed analysis includes graphical depictions of the Taylor-model enclosures at various time points, which show the non-convexity of the solution sets.

In a survey paper [Neu03], Neumaier puts Taylor models into their historical context (stressing that they are actually a reinvention of *Taylor forms / arithmetics* from the 1980s). More importantly, he also highlights that "Taylor models in themselves are as prone to wrapping as other naive approaches such as simple integration with a centered form, since wrapping in the error term cannot be avoided" [Neu03, p. 52]. However, also in his view, their curved boundaries offer benefits in the case of highly non-linear ODEs, if e.g. the *shrink wrapping* technique is used, which Neumaier characterizes as a "slightly modified nonlinear version of the parallelepiped method". While allowing tighter bounds, Taylor models have been found to come at high computational cost. Neumaier, in the same survey, quotes runtimes of COSY[4] being in the order of 60–1000 times higher than those of AWA for celestial mechanics examples [Neu03, p. 53].

This high computational cost has influenced our choice of opting for VNODE-LP albeit the wrapping-induced limitations that have to be expected in non-linear systems and large initial domains. A recent development in the field of Taylor models is *Flow** [CÁS13]. In the paper, the expectable higher accuracy of Taylor models versus VNODE-LP is demonstrated, but more importantly, also a significantly improved runtime performance. In the future, the choice of Taylor models for ODE enclosures may thus be much more viable than it has been during the development of our approach.

Interval Methods in Abstraction Refinement. In HSolver [RS07], interval arithmetic is used to compute the valuation of right-hand sides of ODEs over

[3] Dependency in this context means that if the same variable occurs more than once in an expression, e.g. $y = x \cdot x$, interval evaluations with $x \in [-10, 10]$ would yield $y \in [-100, 100]$. Single use expressions avoid this effect; in the example $y = x^2$ would yield $y \in [0, 100]$, a tight representation of the range.

[4] Reported for the COSY-INFINITY package—available from http://www.bt.pa.msu.edu/index_cosy.htm.

the facets of sub-boxes, into which it partitions the state space of the hybrid system under analysis. The interval result allows to easily decide for each border, whether a trajectory can pass it. With this information, the cell graph can be completed into a discrete abstraction of the hybrid system by introducing edges between cells whenever trajectories can pass through them. *Abstraction refinement* is done not only by splitting cells, which lie on abstract paths between initial and target states, but also by pruning off parts of cells, which have been shown by interval analysis methods to be unreachable. To prune cells, HSolver uses a linear overapproximation of the possible evolution, based on the mean-value theorem.

Like reachability computation methods, also this approach overapproximates the entire set of reachable states. Instead of consecutive addition of elements to the reachable state set, however, the abstraction always is an overapproximation of the result—albeit initially a very coarse one. By refuting the concretizability of abstract paths, refinements are done until the target states can be shown to be unreachable. The size of the abstraction depends on the number of discrete modes of the original hybrid automaton and the number of cells introduced by splitting boxes during refinement. The restriction to linear overapproximations may cause a significant amount of overestimation, which may necessitate further refinement steps. Using more accurate ODE enclosure methods would theoretically be possible, but is complicated by trajectories that never leave a cell and therefore cannot be captured by any time-bounded enclosure.

Theorem Proving. The *KeYmaera* tool [PQ08] offers an interactive theorem-proving approach to the analysis of hybrid systems. While automation of proofs is a research goal for theorem provers, in practice, user interactions are required and may only be possible with deep insight into the behavior of the system under analysis. KeYmaera requires closed-form solutions of the ODEs occurring in the hybrid system under analysis. In [PQ08] Mathematica and a Java-based library, Orbital, are named as tools to obtain these solution functions. While numerical methods cover a wide range of practically-relevant ODEs, even the best symbolic approach to compute closed-form solutions will be limited to those ODEs for which they exist—with practical implementations being even more restrictive.

Constraint Programming. Closer to our approach than theorem proving, which tries to deduce new knowledge through the application of (potentially user-selected) rules on already-derived facts, is the *Constraint Logic Programming (Functions)* approach CLP(F) [HW04]. It consists of an input language which allows the encoding of arithmetic and analytic constraints over real and function variables, an interpreter for this language, and integrated interval arithmetic solvers, which deduce ranges for the solutions of the variables under the given constraints. The language then allows to submit queries, which can encode target conditions like reachability or counter-examples to an invariance property. Additionally required are user-specified choices of solver-algorithms (e.g. a simple iteration of the encoded transition system or splitting and contraction). The interpreter then deduces intervals which potentially contain a solution or proves unsatisfiability of the constraints. This approach allows the direct encoding of hybrid systems with non-linear ODEs like the approach, we explore in this thesis.

The ODEs, which can be specified in this language, are automatically translated into arithmetic constraints. This process is described in [Hic00]. The underlying idea is to introduce constraints that represent the Taylor series of the exact solution and the remainder term. For integration horizons that require more than one step, explicit encoding of the intermediate integration steps is required, exemplified by a CLP(F)-procedure that generates the appropriate constraints. Importantly, however, the handling of differential equations by mere replacement with the corresponding Taylor series constraints, does nothing to prevent the wrapping effect that so prominently dominates the research on validated numerical solving of ODEs. Hickey and Wittenberg clearly admit this shortcoming: "CLP(F) makes no attempt to handle the wrapping problem, other than the simple minded solution technique of dividing each rectangle into smaller pieces, exacerbating the performance problems." [HW04, p. 414] In [Hic00], a comparison with the literature is made and even on a simple one-dimensional non-linear example, the CLP(F) approach is found to take 220 seconds to enclose the solution with the same precision which the then state-of-the-art solvers were able to achieve within 0.12 seconds.

Similar, yet different, is more recent work by Goldsztejn et al., who investigate including ODEs in classical *constraint programming* [GMEH10]. This work is similar to the CLP(F) approach in that also here a declarative constraint language is introduced over real variables, but different in that differential equations are represented only by their role of connecting two instances of real variables and a duration by the ODE's solution instead of by general analytic constraints over function variables. An important restriction of this work with respect to its applicability to the analysis of hybrid systems is that the constraint system must be purely conjunctive and all variables must be continuous. While this easily allows the analysis of purely continuous systems, an application to hybrid systems would require a number of modeling tricks (like extracting discrete variables from the reals, e.g. through trigonometric functions, and encoding of modes by use of these discretized variables as coefficients in ODEs and arithmetic constraints). For the intended scenario of continuous conjunctive problems, the approach utilizes a combination of validated numerical integration of ODEs, using the *CAPD interval library*[5], with the *uni- and multi-variate interval Newton methods*, which are local optimization tools that converge towards solutions by making use of the known derivatives. Moreover, using interval Newton, proofs of existence can be derived, which may be important for some applications. For finding errors in (hybrid) systems, however, such proofs may not be much better than candidate solutions for which the near-satisfaction of all constraints has been shown (and hence attention is drawn to potential error trajectories that may only be slightly off).

An important observation that underlines the similarities between this approach and ours is that also within the CP framework the traditional optimization goal of validated numerics libraries to provide tight enclosures from tight initial conditions may be detrimental to their ability to quickly prune the search space when, initially, interval boxes are still rather large and enclosure computation with high accuracy hence very costly. Goldsztejn et al. suggest: "experiments have shown that at the beginning of the search, when intervals are large, the pruning is not efficient so reducing the order of the Taylor expansion would

[5] CAPD is available from http://capd.ii.uj.edu.pl—cited after [GMEH10].

certainly pay off." [GMEH10, p. 233] The addition of bracketing systems to our approach aims at the same issue: giving faster enclosures for larger domains. A totally satisfying solution has, however, not been incorporated in either approach—and might in fact consist of an entire portfolio of pruning methods with different degrees of accuracy.

Satisfiability Modulo ODE Approaches. The closest competitors to our approach are both using the same central idea as we are: to extend satisfiability search to cover the entire range of boolean combinations over arithmetic and ODE constraints. The central goal and ideas being equal, we need to present some more technical details to be able to outline the remaining differences.

As will become clearer in the next chapter, our approach is based on an extension of the iSAT solver for boolean combinations of arithmetic constraints. Essentially, iSAT performs three operations: deduction of new bounds (i.e. pruning of the search space), splitting of intervals (when deduction no longer yields progress), and learning of conflict clauses from analyzing the reasons when a branch of the search has lead to an empty box. Jumping back and undoing decisions that led to the conflict, iSAT traverses the state space until it either finds a box whose width is below a user-defined threshold and is consistent under the given deductions—hence a candidate solution—or it encounters an empty box, but during analysis of the reasons for this conflict finds that no undoing of decisions would mitigate it. In this case, the conflict is inherent to the constraint system, which is thus found unsatisfiable. While proven unsatisfiability is a conclusive answer, reporting a candidate solution leaves margin for error, since a refined analysis with more splitting or tighter deduction resolution might have ruled out an actual solution within that box. A guarantee can hence only be given for unsatisfiability, but no proofs of existence are provided in the candidate solution case, which is therefore also reported as "unknown".

This algorithm is different from classical SMT algorithms in that it does not combine a boolean SAT solver with a number of theory solvers, but instead lifts classical *conflict driven clause learning* (CDCL) from propositional SAT to SMT by incorporating *interval constraint propagation* (ICP) for deductions and using intervals as the search domain for the variables.

Our extension of the core iSAT to iSAT-ODE is based on computing, during the deduction phase, interval enclosures for the ODE constraints under the assumption of the active flow invariants and the current valuation. Its results are learned as clauses, added to the constraint system, and thereby indirectly lead to new deductions on the ICP level. Again, we have left out the details, which are the subject of the next chapter.

In [IUHG09], Ishii et al. introduced an approach to solve *hybrid constraint systems* (HCS) by combining ODE enclosures based on VNODE-LP with constraint programming. The HCS language allows the specification of non-linear ODEs and non-linear guard conditions, whose first intersection is to be computed. This approach therefore solves a sub-problem that commonly occurs in the analysis of hybrid systems: to identify switching times and the corresponding regions of the state space in which switching occurs. To achieve a higher convergence rate of the pruning operator, univariate interval Newton is applied on top of the ODE enclosures. This is achieved by first computing an enclosure

of the trajectories for one step (whose size is determined by VNODE-LP) and thereafter to compute the interval extension of the ODE's right-hand side over this enclosure at the next time point. By applying automatic differentiation on an interval extension of the guard function with which the intersection is to be determined, the ingredients for the interval Newton method can be computed. When interval Newton converges towards an intersection of guard and trajectory, this can additionally be taken as an existence guarantee for a solution inside the narrowed box.

While in this paper, Ishii et al. apply their method to examples, which require only successive computations of the trajectory-guard intersections, in a later paper [IUH11], they address the entire Satisfiability modulo ODE problem. Their solver *hydlogic* takes as input a hybrid system description and performs bounded unwindings of its transition predicate. In their paper, two theoretical approaches are presented. The simpler one is a classical SMT approach, where a SAT solver selects a set of constraints, which together would satisfy the formula's boolean structure, and a theory solver—based on the HCS algorithm from [IUHG09]—checks this set's satisfiability. In the second one, called *IncSolve*, this SMT scheme is applied in a more forward directed fashion. Starting from an initial mode and continous state given by an interval box, the HCS propagator is used to test all guard predicates that belong to jumps from the current mode. The goal is thus to find continuous evolutions that reach within bounded time the guard condition (without violating the flow invariant of the current mode in between). Using the HCS propagator's existence guarantees, the solver can detect whether at least one guard can be reached. If no such guarantee can be given, the algorithm splits an *initial* interval and retries on this smaller box. Jumps whose guard predicate cannot be reached are discarded from the set of possible successors in the trace. If a guarantee has been obtained that at least one jump is feasible, the unwinding depth is increased and the successor mode from this jump is taken as the frontier from which the next step continues. When this successor mode is labeled unsafe, a counter-example to safety has been found.

If in this case, all intermediate jumps have been proven to be guaranteed, the authors consider the resulting error trace to be guaranteed as well. For this to be correct, it is important that at all intermediate steps, the entire domain of reachable states is retained. This way, an exact error trajectory from an initial point would be known to definitely reach a point which satisfies the guard conditions. Assuming the computed domains are entirely traced through the subsequent continuous evolutions, this guarantee can be retained to the final step, when an unsafe mode is reached. It is important to note that in constraint solving based on box consistency, which aims only at satisfying each constraint individually, the individual existence guarantees are not sufficient for the existence of a point solution that satisfies all (active) constraints. The strict adherence to one direction in the IncSolve algorithm, however, lifts local guarantees to the level of the entire formula. A similar idea based on explicitly computing suitable redirections has been presented under the name "strong satisfiability" in [FHR+07] for the core iSAT solver, but has not been included in iSAT-ODE.

We understand that hydlogic is an implementation of the IncSolve algorithm, making use of a toolkit for SAT solving, supporting the construction of SMT solvers, called Decision Procedure Toolkit (DPT) and the HCSLib for solving

hybrid constraint systems as outlined above. This library makes use of an interval-based constraint solver (Elisa) "based on box-consistency" and VNODE-LP for computing the enclosures of ODEs.

We observe the following central differences with our own solver iSAT-ODE. The hydlogic IncSolve algorithm effectively enumerates abstract traces by SAT solving of a boolean abstraction (of the mode graph) and uses HCS propagation to select suitable successor modes. While our approach also supports solving with incrementally increased BMC unwinding depths and also stores learned facts like conflict clauses and ODE propagations from previous unwinding depths such that they can be used for the enlarged formula as well, in iSAT-ODE, propagation and search go in every direction. Splitting in hydlogic is restricted to the initial domains of the variables, in iSAT-ODE splitting occurs on every variable, it may e.g. also occur in the middle of a trajectory or search may actually go entirely backwards if e.g. the target state is known entirely and backward propagation from there reduces the number of possible initial states. From the hydlogic algorithm and from the examples used as benchmarks in [IUH11], it is doubtful that the algorithm can be applied to systems that allow non-determinism in their actions and later evolve differently depending on which value is chosen for a variable when such an action is performed: restricting splitting to initial variables is not sufficient to resolve this situation and following all possible states after a jump will be practically impossible if trajectories diverge. This restriction can certainly be lifted by modeling the non-deterministic choices as part of the initial state, such that they can be subjected to splitting. On the other hand, regarding the entire constraint system without a sense of direction as done in iSAT-ODE, is detrimental to our ability to prove the existence of a trajectory. The use of interval Newton in HCS solving together with the strict forward analysis allow hydlogic to give guarantees on the existence of point-valued trajectories, which iSAT-ODE cannot give in its current state. We think that such checks could however be done after finding a candidate solution by redirecting the constraints of the formula such that propagations point strictly into one direction and no part of the valuation is removed by splitting. An important technical detail is only hinted at in [IUHG09], namely that results from VNODE-LP are cached for reusing. In iSAT-ODE considerable effort has been invested in detecting reusable results and avoid recomputations, too.

In chapter 4, we give detailed comparisons with a number of benchmarks from [IUH11], which themselves have been compared by Ishii et al. with HSolver and PHAVer within the reasonable limitations we have mentioned above.

Most recently, Gao et al. have introduced *dReal* [GKC13], a solver for SAT modulo ODE formulae. The paper's theoretical contributions have already been discussed in Section 2.3. In the paper, the actual implementation is described only briefly, but it is clear that it does not deliver a method to δ-decide SAT modulo ODE formulae, since it builds on off-the-shelf libraries, which cannot provide the arbitrary precision which is required for the theoretical results to apply. More precisely, dReal uses a classical SMT scheme (with the *opensmt* library as a SAT framework), and combines interval constraint propagation (using the *realpaver* algorithm) with enclosure methods for ODEs coming from CAPD, which is also used in [GMEH10]. This solver architecture together with the theoretical algorithms described in the paper suggests that the SAT layer is used to instantiate sets of constraints such that they satisfy the boolean formula structure. This conjunction of constraints is then analyzed by the ICP layer,

which tries to find solutions by propagation (making use of ODE enclosures for the ODE constraints and flow invariants) and by splitting intervals when the image of the current interval valuation under the functions occurring in the constraints exceeds a maximum width threshold.

Chapter 3

Algorithmic Approach

We could not avoid to briefly sketch some parts of our approach in the previous chapter when trying to point out differences to the related literature. In the following sections, we will however be able to give a coherent and much more detailed presentation of how we try to solve SAT modulo ODE formulae.

First, we describe the iSAT algorithm for solving boolean combinations of non-linear arithmetic constraints over continuous and discrete variables. The second building block is given by the VNODE-LP algorithm for computing trajectory enclosures for sets of initial value problems of ODEs. These two algorithms form the basis for the contribution of this thesis: to extend iSAT into iSAT-ODE for solving SAT modulo ODE formulae, which additionally include ODE and flow invariant constraints. An important technical aspect is to avoid expensive recomputations of enclosures for ODE solution sets by learning and caching on multiple layers. When VNODE-LP is used to compute enclosures for non-linear ODEs and the initial states are given by rather large sets, the size of the enclosures may grow too strongly to be of any use. We therefore implemented the bracketing system approach from the literature, which—in some cases—provides dramatically better enclosures than using VNODE-LP directly. Its downside is its inability to fight the wrapping effect, which is why it is combined with the direct method in iSAT-ODE. The experimental evaluation in the subsequent chapter will then highlight the effects of these algorithmic improvements.

3.1 The Core iSAT Algorithm: Learning Conflicts by Branching and Pruning Intervals

The iSAT algorithm, which has been introduced in [FHR+07] is described in detail and evaluated on a number of benchmarks in [Her11, Chapter 5]. Underlying this thesis is a reimplementation of the iSAT algorithm that was done within the AVACS H1/2 project and has been made publicly available as iSAT 1.0[1]. The author of this thesis has contributed to this reimplementation, but it does not form a scientific contribution of this thesis.

The iSAT algorithm can be understood to be a tight coupling of two major

[1] https://projects.avacs.org/projects/isat/

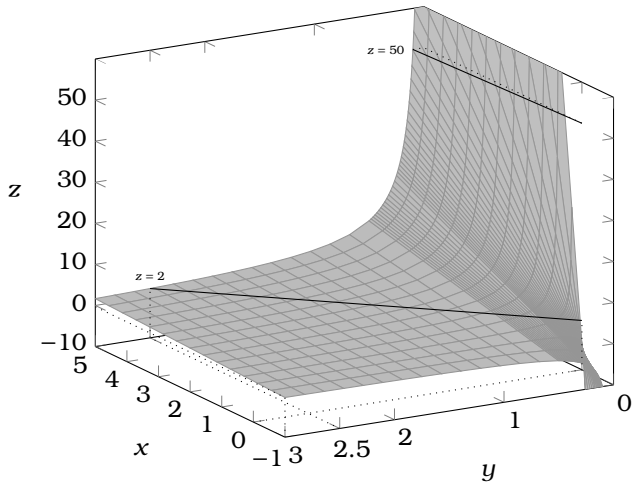


Figure 3.1: Interval pruning for $x = y \cdot z$ over given interval ranges $x \in [-1, 5]$, $y \in [0, 3]$, and $z \in [2, 50]$. Plotted are the surface $z = {}^{x}/_{y}$ and its contour lines at $z = 2$ and $z = 50$, which mark the boundary of z's given range. Pruned intervals are $x \in [0, 5]$, $y \in [0, 2.5]$, and $z \in [2, 50]$.

ideas. One is modern SAT solving as introduced in Section 2.3, the other is interval constraint propagation (ICP) (cf. e.g. [BG06]).

3.1.1 Interval Constraint Propagation

Essentially, solving a formula amounts to searching for a *point* in the variables' n-dimensional domain, which satisfies sufficiently many constraints such that their boolean combination evaluates to true. Since some of these variables are continuous, i.e. their domain is a subset of $\mathbb{R}$, there is potentially an infinity of points to choose a variable's value from. While for some theories, there are clever—though not necessarily cheap—strategies (like linear programming or cylindrical algebraic decomposition), which allow to only look at a finite number of points to decide whether a solution exists, the central idea in interval analysis and ICP is to *not* look at individual points, but instead take subranges of the variables' domains, i.e. intervals and boxes spanned by them in multiple dimensions, and evaluate the expressions over these intervals instead.

Example 2 (pruning intervals)**.** As a motivating example, consider the constraint $x = y \cdot z$ and given ranges for all three variables, $x \in [-1, 5]$, $y \in [0, 3]$, and $z \in [2, 50]$. The goal is thus to find a point (x_s, y_s, z_s) from $[-1, 5] \times [0, 3] \times [2, 50]$ such that $x_s = y_s \cdot z_s$ holds. The idea of ICP is to prune points from this box that are known not to be possible solutions. From the ranges for y and z, we can deduce that $x \in [0, 3] \cdot [2, 50] = [0, 150]$. However, we already know that $x \in [-1, 5]$ from the given domain. By simply intersecting the known and the computed bounds, we find that $x \in [-1, 5] \cap [0, 150] = [0, 5]$. Also a deduction in the other direction is possible by transforming the constraint into $y = {}^{x}/_{z}$ and computing $y \in {}^{[0,5]}/_{[2,50]} = [0, 2.5]$. Again, we intersect with the known interval for y, and receive $y \in [0, 2.5] \cap [0, 3] = [0, 2.5]$. Last, we try to

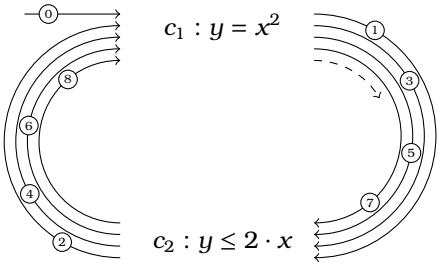
compute a refined enclosure for z via $z = {}^{x}/_{y}$, hence $z \in {}^{[0,5]}/_{[0,2.5]}$. With zero in the denominator interval, this enclosure is not very useful, it could be trivially expressed as $z \in (-\infty, +\infty)$ (or as $z \in [0, +\infty)$, since it is known in this case that the quotient of any two points from the given intervals would be non-negative), whose intersection with the known interval for z would leave us with $z \in [2, 50]$. While not having given us a point solution, this approach has actually removed infinitely many points and hence refined the search space, in which we have to look for a solution. This interval pruning is illustrated in Figure 3.1.

Definition 10 (interval extension)**.** Within the example, we implicitly lifted arithmetic operators to interval operands. We call $\mathbb{IR}$ the *set of intervals* over $\mathbb{R} \cup \{-\infty, +\infty\}$. An element of $\mathbb{IR}$ is thus an interval $[a, b]$, $(a, b]$, $[a, b)$, or (a, b) with a and b real numbers or positive / negative infinity and denoting the continuum of real numbers between the two bounds. Similarly we define $\mathbb{IZ}$ as the set of integer intervals over $\mathbb{Z}$, in which each interval contains all integer numbers between the given bounds. Following closely [BG06, Definition 16.5], we call an interval function $F : \mathbb{IR}^n \to \mathbb{IR}$ an *interval extension* of a function $f : \mathbb{R}^n \to \mathbb{R}$, if and only if for every interval box $\vec{I} \in \mathbb{IR}^n$, the set of exact values of f over the points from $\vec{I}$ is contained in the valuation $F(\vec{I})$, i.e. $\{f(\vec{x}) \mid \vec{x} \in \vec{I}\} \subseteq F(\vec{I})$.

The simplest form of interval extension (and at least as old as Moore's seminal work on the domain [Moo66]) is obtained by introducing interval extensions for each arithmetic operator and replacing the point operators in f with these interval operators to form F. Having a binary operator $\circ$ and arguments $x, y \in \mathbb{R}$, the interval extension for $X, Y \in \mathbb{IR}$ must contain the set of all point solutions $Z := \{z \in \mathbb{R} \mid z = x \circ y, x \in X, y \in Y\}$ and must at the same time be an interval from $\mathbb{IR}$. The smallest set satisfying both conditions is the interval hull, i.e. ideally $X \circ Y := [\inf(Z), \sup(Z)] \in \mathbb{IR}$. Similarly, these definitions can be made for unary operators and for discrete domains.

For practical implementations, interval borders need to be representable. In iSAT, the data type used to represent interval boundaries are floating-point numbers. These have limited precision, i.e. in practice, the above definition of an interval extension needs to be supplemented by a third condition: the lower and upper bound of the interval must be floating-point numbers. Calling $\mathbb{F}$ the set of floating-point numbers of a given precision, and $\mathbb{IF}$ the set of intervals whose boundaries can be represented by these floating-point numbers, we define $\inf_{\mathbb{F}}(Z) := \max(\{x \in \mathbb{F} \mid x \leq \inf(Z)\})$ and similarly $\sup_{\mathbb{F}}(Z) := \min(\{x \in \mathbb{F} \mid x \geq \sup(Z)\}$. These are the largest floating-point number below or the smallest above the set's lower and upper bounds respectively. The tightest possible floating-point interval extension of $x \circ y$ with $x, y \in \mathbb{R}$ for $X, Y \in \mathbb{IF}$ is then $X \circ Y := [\inf_{\mathbb{F}}(Z), \sup_{\mathbb{F}}(Z)]$. In practice, we also call any other floating-point interval that contains this tightest possible interval extension a valid enclosure of the interval extension, since it may still be usable for pruning as long as the amount of overestimation is not too large. These operations can be safely implemented by using *directed rounding*, which is offered either directly by the IEEE 754-compatible programming language, compiler, and CPU architecture for primitive operations or is constructed in software from these primitives, e.g. via libraries like MPFR [FHL+07] and filib++ [LTG+06].

Example 3 (deduction chains)**.** Figure 3.2 explains ICP on an example with a conjunction of two constraints and also illustrates that constraint systems



	X	Y
0	$[-100.00, 100.00]$	$[-100.00, 100.00]$
	$[-\sqrt{100}, \sqrt{100}]$	$[0.00, 100.00]$
1	$[-10.00, 10.00]$	$[0.00, 100.00]$
2	$[0.00, 10.00]$	$[0.00, 20.00]$
	$[0.00, \sqrt{20}]$	$[0.00, 20.00]$
3	$[0.00, 4.48]$	$[0.00, 20.00]$
4	$[0.00, 4.48]$	$[0.00, 8.96]$
	$[0.00, \sqrt{8.96}]$	$[0.00, 8.96]$
5	$[0.00, 3.00]$	$[0.00, 8.96]$
6	$[0.00, 3.00]$	$[0.00, 6.00]$
	$[0.00, \sqrt{6}]$	$[0.00, 6.00]$
7	$[0.00, 2.45]$	$[0.00, 6.00]$
8	$[0.00, 2.45]$	$[0.00, 4.90]$

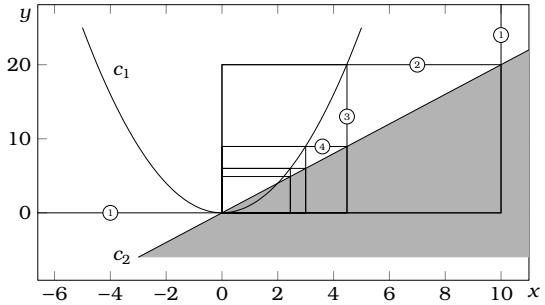


Figure 3.2: Example for a deduction chain with two constraints.

in which variables occur multiple times can lead to theoretically infinite (and practically very long) deduction chains. Initially (step 0), X and Y are given by their domains. Step 1 applies the first constraint c_1 to this box. Since $y = x^2$ and $y \in Y = [-100, 100]$, values for x that are smaller than -10 or greater than 10 cannot be solutions of this constraint and may hence safely be pruned. Also, negative values of y cannot be solutions, which is why Y can be restricted to its non-negative part. In step 2, we can prune the upper bound of Y by intersecting $2 \cdot [-10, 10] = [-20, 20]$ with the known interval $[0, 100]$. By dividing c_2 by 2, we can also deduce new bounds for x through the redirected constraint $y/2 \leq x$, i.e. $[0,20]/2 = [0, 10] \leq x$, which can only be satisfied by values of x that are greater or equal to the lower bound of this interval, i.e. must be non-negative. In step 3, we apply again c_1 and thereby gain the information that values for x which are greater than the square root of Y's upper bound, i.e. $\sqrt{20}$, cannot be solutions. However, since we are not using an infinite precision representation, we need to perform outward rounding here (assuming a fixed-point representation with two post-decimal digits for illustration purposes), hence get $x \leq \sqrt{20} \in 4.472 \pm 0.001 \leq 4.48$, which we apply as upper bound for X.

Using limited precision representations, we will obviously reach a fixed-point eventually, i.e. deduction will not provide any new bounds. However, were we using infinite precision, even this system of only two constraints and two variables would cause a problem for termination. In practice, floating-point numbers are finite precision, but may nonetheless cause deduction chains to be much longer than desirable. The simple solution to this problem is to introduce an acceptance threshold—in case of iSAT a user-defined parameter

called *absolute bound progress*—which defines a lower bound for the difference of a new bound with respect to the known bound.

The example also illustrates an important technical detail. Consider e.g. step 2. Even without knowing the current lower bound of X, we could safely deduce from X's upper bound 10, that no value for y greater than $2 \cdot 10$ could possibly satisfy c_2. This observation holds in many more situations and is essentially based on monotonicity properties of the arithmetic operators. In [Her11], a number of propagation rules are given, which do not propose the use of interval reasoning as applied in the example, but instead to consider individual bounds and thereby know more precisely the reasons why a new bound has been deduced. Besides potentially reducing the number of necessary computations, the major motivation is in fact to get a more confined set of reasons for each deduction, which is helpful in the context of conflict analysis—where smaller reason sets mean more general conflict clauses, which prune off larger parts of the search space and hence accelerate solving.

Relationship to Consistency Notions. Connecting this approach to the literature, it can be classified as *box consistency*, or an approximation thereof due to the termination threshold and floating-point representations, which are not always considered in the theoretical definitions. The definition of box consistency for a single constraint by Benhamou and Granvilliers [BG06, Eq. 16.2] is done via a fixed-point equation. Calling I_i the domain of variable x_i and having a constraint $f(x_1, \ldots, x_n) = 0$, they call I_k box consistent if

$$I_k = \text{hull}\{a_k \in I_k \mid 0 \in F(I_1, \ldots, I_{k-1}, \text{hull}\{a_k\}, I_{k+1}, \ldots, I_n)\},$$

with F being an interval extension of f. In [Her11], the notion is considered a variant of *hull consistency*, but with interval versions of the operators being used, which effectively makes the definition very similar to this one used by Benhamou and Granvilliers.

Lifting the above definition to conjunctions of constraints additionally requires us to intersect the box-consistent sets of the individual constraints and repeat the fixed-point iteration as shown in the chain of deductions that was shown in the example. Benhamou and Granvilliers point out that the standard way to obtain a box-consistent valuation is based on a bisection search to find the extreme points which constitute the end-points of the hull interval. Since the iSAT algorithm handles more than purely conjunctive formulae and therefore requires splitting on a layer outside of these interval propagations, we here only consider the case of having reached a (pre-)fixed point without branching.

Interpretation of Consistency and Pruning Results. Whichever name is used, it is important to note some fundamental characteristics that have a significant impact on the interpretation of results. Even if deductions could be done with perfect accuracy and be continued until the true fixed point is reached, not every point from the remaining box is a solution. Consider e.g. the point $(1, 0)$ in Figure 3.2. This point will remain inside the consistent box since it is "protected" by the solutions at $(0, 0)$ and $(2, 4)$, which—being solutions—cannot be pruned off by deductions. Obviously, the valuation $x = 1$ and $y = 0$ does not satisfy the constraint c_1, hence is not a solution of the conjunction of both constraints.

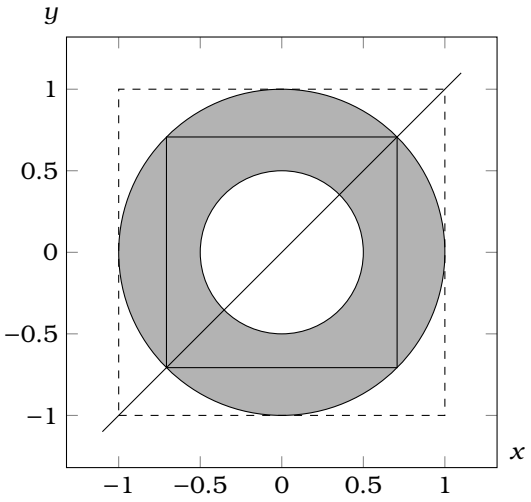


Figure 3.3: Box consistent valuation for $0.5 \leq x^2 + y^2 \leq 1 \wedge x = y$ with discontinuous solution sets leading to non-existence of solutions for subranges of each dimension of the valuation. Note that without splitting, deduction rules in iSAT only suffice to deduce the dashed box $[-1, 1]^2$ instead of the tightest possible box $[{}^{-1}/_{\sqrt{2}}, {}^{1}/_{\sqrt{2}}]^2$ that encloses all solution points.

Worse, not even for every point chosen in one dimension, there exists a point in the remaining dimensions such that together they form a solution. As an example, consider Figure 3.3 and e.g. $x = 0$ or $y = 0$ with their large surrounding region in which the constraint system $0.5 \leq x^2 + y^2 \leq 1 \wedge x = y$ is not satisfiable. The reason can be seen in the existence of non-convex solution sets which have been enclosed by the hull operator.

Finally, a valuation may be box-consistent, but not contain a single solution at all. Figure 3.4 illustrates this problem by showing a constraint system consisting of two oscillating curves that do not intersect for $x \in [-10, 10]$. The only deductions that can safely be made are to exclude the ranges $y \in (1, 1.1]$, which does not satisfy $y = \sin(x)$, and $y \in [-1, 0.9)$, which does not satisfy the other constraint $y = \sin(1.001 \cdot x) + 0.1$.

It is noteworthy, that stopping deduction chains when they do not yield sufficient progress can cause boxes to be considered consistent, even though further propagations would show that they do not contain a solution at all.

Slow Convergence and Higher Approximation Orders. As a side note, in iSAT, we do not employ methods with higher approximation orders that are known to converge more quickly than the comparatively simplistic method above. Benhamou and Granvilliers [BG06] point out: "it is clear that the search converges slowly. The univariate interval Newton method [. . .] can be used to accelerate the convergence [. . .]." And Neumaier provides even more of a warning that an approach with as low a convergence as presented above may very well have to explore an exponential number of boxes when used inside a branch and bound scheme:

"Indeed, branch and bound methods for minimizing a function in

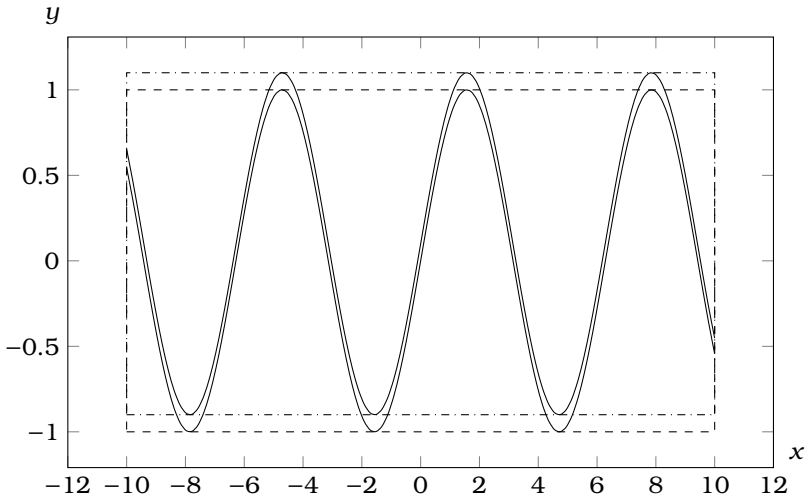


Figure 3.4: For the constraints $-10 \le x \le 10 \wedge y = \sin(x) \wedge y = \sin(1.001 \cdot x) + 0.1$, without splitting, only the box $[-10, 10] \times [-0.9, 1]$ can be deduced, which is the intersection of the two bounding boxes that can be wrapped around the two sine curves. Obviously, there is not a single intersection of the two curves, which means that the box does not contain a solution at all.

> a box (or a more complex region) frequently have the difficulty that subboxes containing no solution cannot be easily eliminated if there is a nearby good local minimum. This has the effect that near each local minimum, many small boxes are created by repeated splitting, whose processing may dominate the total work spent on the global search.
>
> This so-called cluster effect was explained and analyzed by Kearfott and Du [. . .]. They showed that it is a necessary consequence of range enclosures with less than cubic approximation order, which leave an exponential number of boxes near a minimizer uneliminated. If the order is < 2, the number of boxes grows exponentially with an exponent that increases as the box size decreases; if the order is 2, the number of boxes is roughly independent of the box size but is exponential in the dimension. For sufficiently ill-conditioned minimizers, the cluster effect occurs even with methods of cubic approximation order. [. . .]
>
> For finding all zeros of systems of equations by branch and bound methods, there is also a cluster effect. An analogous analysis by Neumaier and Schichl [. . .] shows that one order less is sufficient for comparable results. Thus first order methods (interval evaluation and simple constraint propagation) lead to an exponential cluster effect, but already second order methods based on centered forms eliminate it, at least near well-conditioned zeros. For singular zeros, the cluster effect persists with second order methods; for ill-conditioned zeros, the behavior is almost like that for singular zeros since the neighborhood where the asymptotic result applies

becomes tiny." [Neu03, p. 49]

This warning has to be taken seriously and it can be taken as a hint to why we may observe long solving times in some of our benchmarks. Our excuse for not taking more elaborate evaluation schemes—besides the initial ignorance of these matters—must be seen in the necessity of keeping track of reasons for each deduction and the desire to keep the reason sets as small as possible, hence even using the kind of bound arithmetic that is implemented in the arithmetic deduction rules of iSAT. The current implementation also includes a rewriting step into constraints that consist only of one arithmetic operator each and explicit auxiliary variables to allow composition of the larger expressions. This decomposition, which is not atypical, may hinder the easy adaption of methods of higher approximation order. Leaving these practical considerations aside, however, a next-generation solver should probably include methods with higher approximation order (yet probably not rely on them exclusively as they may behave worse when initially domains are still large) to circumnavigate these issues pointed out in the literature.

3.1.2 Input Language and Preprocessing

Having discussed ICP as a method to handle (conjunctions of) arithmetic constraints over variables with interval valuations, we can now refocus on the original problem which iSAT addresses: satisfiability modulo non-linear arithmetic over bounded continuous and discrete variables including transcendental functions.

Syntax of the Input Language. Examples for the input language of iSAT-ODE have already been shown in Figures 2.4 and 2.5. Since iSAT-ODE just extends the iSAT language by ODE- and flow-invariant-constraints, iSAT's input language is very similar, just lacking these ODE-related constructs. Models consist of four parts: a declaration of variables and constants, followed by the initial, transition and target predicates needed for encoding BMC problems.

⟨*model*⟩ ::= DECL { ⟨*constant declaration*⟩ | ⟨*variable declaration*⟩ }
INIT- ⟨*formula*⟩ -TRANS- ⟨*formula*⟩ -TARGET- ⟨*formula*⟩

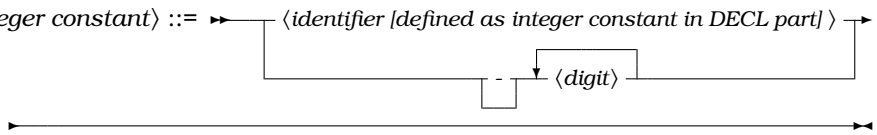
Within the declaration part, identifiers for symbolic constants and variables with their type and range information need to be introduced before they can be used within the remaining sections. Subsequently, we will call a (symbolic or literal) constant integer, if it has an integer value and will call all constants rational (including integer constants).

⟨*constant declaration*⟩ ::= define- ⟨*identifier*⟩ - = - ⟨*rational constant*⟩ - ;

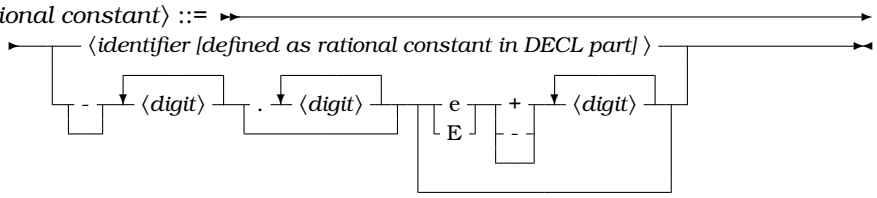
⟨*variable declaration*⟩ ::=
boole ⟨*identifier*⟩ ;
int[- ⟨*integer constant*⟩ - , - ⟨*integer constant*⟩ -]
float[- ⟨*rational constant*⟩ - , - ⟨*rational constant*⟩ -]
float[0 , - ⟨*rational constant*⟩ -] time
float[0 , - ⟨*rational constant*⟩ -] delta_time

Here, we already indicate the additional declarations for the global time and the duration used by iSAT-ODE in gray. These are not part of the original iSAT input language.

⟨*integer constant*⟩ ::=



⟨*rational constant*⟩ ::=

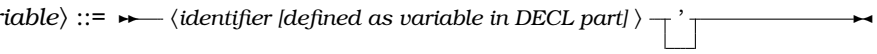


With identifiers following a usual naming convention for variable names.

⟨*identifier*⟩ ::=

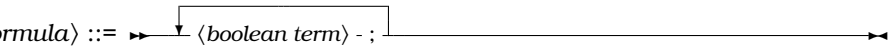
Subsequently, we refer to declared symbolic constants and variables by their introduced identifiers. As motivated in our Definition 2 of hybrid automata, within the transition relation we need to be able to refer to primed (e.g. x') and unprimed (x) instances of these variables to describe their post- and pre-states. Within the initial and target predicates, the primed instances of variables are not allowed—they may thus only occur within the transition predicate. They also cannot occur within ODE constraints or flow invariants in iSAT-ODE.

⟨*variable*⟩ ::=

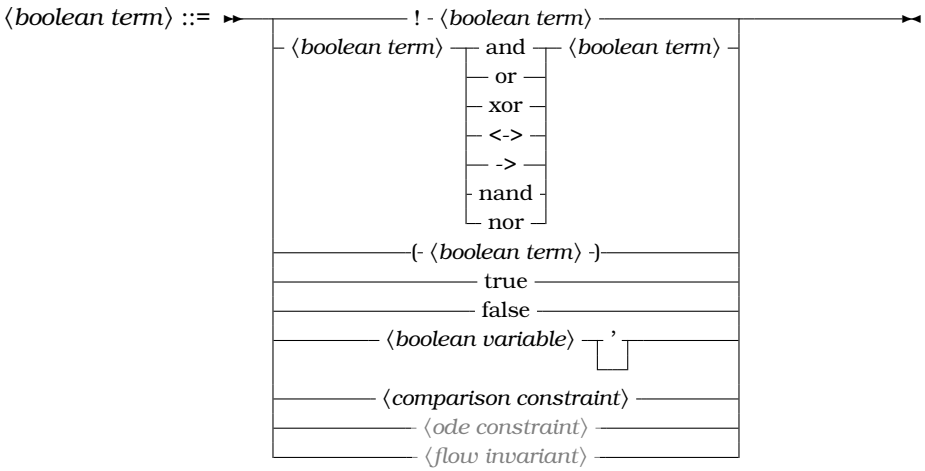


The initial, transition, and target predicates are then formulae consisting of a conjunction of arbitrarily many boolean terms, conveniently segmented and terminated by semicolons, which—when used between two boolean terms—just form a low-precedence alternative to the normal boolean connective “and”.

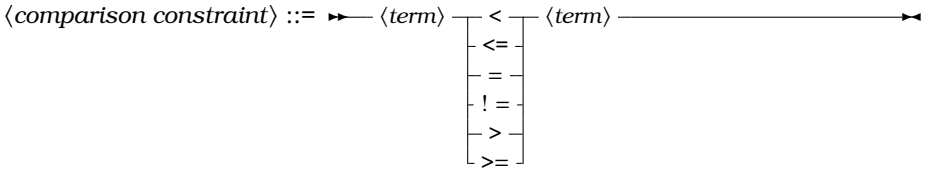
⟨*formula*⟩ ::=



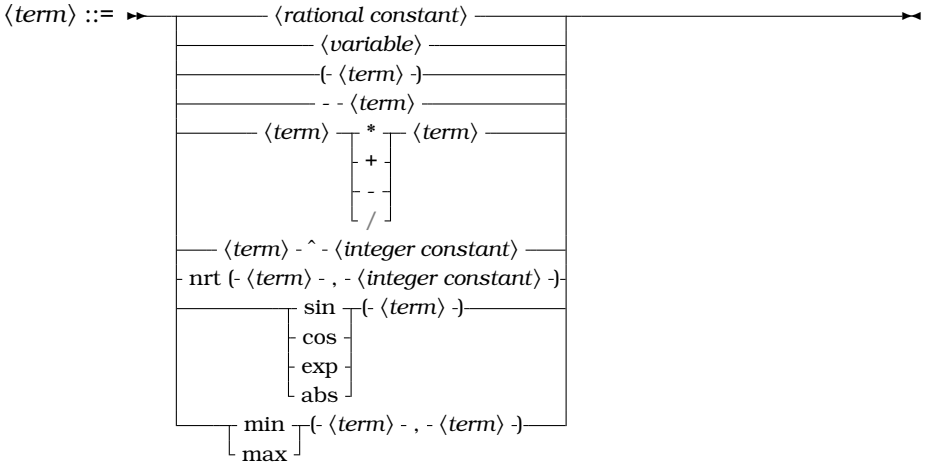
Boolean connectives are then used between atoms, which can be boolean constants or variables—used to model discrete aspects—together with constraints over discrete and continuous variables. This is also the level, where ODE constraints and flow invariants occur in our extension iSAT-ODE—shown here to avoid repeating the entire syntax definition, later.



Comparisons between arithmetic terms over discrete and continuous variables and constants evaluate to boolean values and therefore form valid boolean terms.



Arithmetic terms, finally, consist of (nested) arithmetic operators over the declared variables.



It is important to note that the core iSAT solver only accepts total operators, i.e. does not accept the division operator at all and will only accept the n-th root (nrt) for positive argument if n is even. Divisions can easily be replaced by manually introducing an auxiliary variable and rewriting the expression into a multiplication constraint, making explicit the choice of bounded range for the auxiliary variable, which might otherwise be unbounded and hence unsupported

by the iSAT algorithm. Within iSAT-ODE, we accept these operators in the right-hand sides of ODE constraints, albeit also there with the semantic restriction that the right-hand-side expression needs to be defined over at least the region admissible under the specified flow invariants.

⟨*ode constraint*⟩ ::= ►► (d.- ⟨*real variable*⟩ - / d.time = - ⟨*term*⟩ -) ►◄

⟨*flow invariant*⟩ ::= ►► ⟨*real variable*⟩ - (time) - <= / >= - ⟨*rational constant*⟩ ►◄

These ODE constraints (from iSAT-ODE and here only for completeness of the input language), finally, can only use continuous variables, i.e. using the "float" type in their declaration. This restriction refers to the left-hand-side variables in ODE constraints and flow invariants, as well as to the arithmetic expressions on the right-hand side of the ODE constraint.

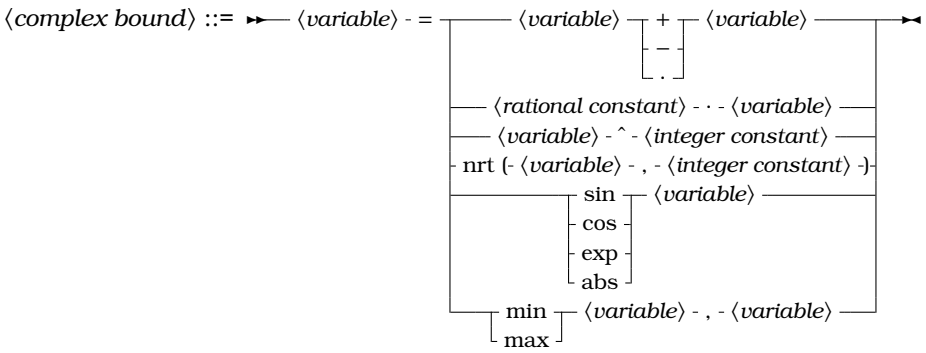
Rewriting into the Internal Format. While this rich input language is convenient for modeling, directly supporting the arbitrary nesting of arithmetic operators and the potentially complex boolean structure would necessitate an equally rich set of deduction rules in the solver. As indicated before, iSAT uses deduction rules based on a simpler representation. After the creation of the concrete BMC instance (i.e. the k-fold instantiation of variables and unwinding of the transition predicate), the solver's front end therefore performs a satisfiability-preserving rewriting into a simplified normal form [Her11]. This "definitional translation into conjunctive form" essentially splits nested arithmetic expressions into simpler constraints containing only a single arithmetic operator and introduces fresh auxiliary variables (with automatically computed finite ranges). The boolean structure is equally simplified by the introduction of boolean variables, finally leading to a format that resembles a *conjunctive normal form* (CNF). Admissible atoms, however, are not only boolean variables, but also the decomposed constraints consisting of a left-hand-side variable, a comparison operator, and a right-hand-side expression over at most two variables and an arithmetic operator, or simple bounds consisting of a left-hand-side variable compared with a right-hand-side constant. After rewriting, the internal format, which can be seen as the input to the actual solving algorithm, is structured as follows.

⟨*formula*⟩ ::= ►► ⟨*clause*⟩ (∧) ►◄

⟨*clause*⟩ ::= ►► (⟨*atom*⟩ (∨)) ►◄

⟨*atom*⟩ ::= ►► ¬ ⟨*boolean variable*⟩ / ⟨*simple bound*⟩ / ⟨*complex bound*⟩ ►◄

⟨*simple bound*⟩ ::= ►► ⟨*variable*⟩ - ≤ / < / > / ≥ - ⟨*rational constant*⟩ ►◄



The ODE constraints and flow invariants are not shown here. Their handling will be described in Section 3.3.

3.1.3 Unit Propagation

With ICP, we have introduced a pruning mechanism for conjunctions of arithmetic constraints. With the input formula and its rewriting, we have concretized the solver's task: to find a solution for a *conjunction of disjunctions of atoms*. In order to avoid having to bridge the gap between the problem and the ICP-based pruning only via branching, which we will discuss shortly, one further ingredient is helpful. This building block of iSAT is directly taken from propositional SAT solving: *unit propagation*.

The central underlying observation is simple. If the formula is to be satisfied, in each clause there must be at least one atom that is satisfied. Otherwise, there would be at least one clause in which all atoms evaluate to false, hence also their disjunction and consequently the entire formula, which is a conjunction of all clauses. Unit propagation is applied to all clauses in which all constraints but one are inconsistent with the current valuation. In SAT solving, these unit atoms are simply satisfied by choosing the value of the boolean variable such that the literal (a or $\neg a$) evaluates to true. In iSAT, the atoms are more general and hence unit propagation is actually a trigger for interval constraint propagation: if a constraint becomes unit, it is used to deduce further bounds via ICP.

Example 4 (unit propagation)**.** Given e.g. an excerpt of a formula

$$(a \vee x \leq 5) \wedge (\neg a) \wedge \ldots,$$

we can immediately identify the clause $(\neg a)$ to only have one atom left—in this case because there simply is only one atom. If we were to choose a to be true, the clause $(\neg a)$ would be false and hence the formula not satisfied by our valuation. Consequently, if there are any valuations at all that satisfy this formula, they will be valuations in which a is false. Like in ICP, pruning of a's interval range only discards definite non-solutions, leaving a tightened search space.

In the first clause, $(a \vee x \leq 5)$, the first atom, a, has become false due to our (forced) assignment of a to false. Again, only one atom is left: $x \leq 5$. Any valuation that satisfies this formula will therefore be one, in which $x \leq 5$ holds. Should x's interval valuation contain values greater 5, these could thus safely be pruned off and the new upper bound of x be propagated through other constraints in which x occurs.

Watched Literal Scheme. Unit propagation motivates another mechanism from propositional SAT solving that can be directly incorporated into iSAT: a *watched literal scheme* used to perform *lazy clause evaluation* [Her11]. Since atoms will not participate in ICP unless they have become unit, it suffices to observe two atoms in each clause that are still consistent under the current valuation. These can be considered witnesses for the clause not yet being unit. As soon as a bound is deduced for a variable that occurs in one of the two watched atoms of a clause, its consistency is reevaluated—using ICP without storing the resulting bounds. Still consistent, the atom stays watched. Having become inconsistent, the watch needs to be replaced by another atom in the clause that is still consistent under the current valuation. If no replacement can be found, the number of consistent atoms has dropped from two to one and therefore the clause has become unit.

The downside of having only two watches per clause is that the solver does not necessarily detect when a clause contains an atom that is satisfied by the entire current valuation. If this were discovered, the clause could be considered satisfied and its atoms would no longer have to be visited for the current valuation and any subset thereof. The upside of not having to check after each deduction the satisfaction of all atoms in which a variable occurs is, however, considered to outweigh this potential loss of an earlier detection of satisfied atoms in the case of iSAT. A comparison of an earlier version of the iSAT algorithm with a version of itself where all atoms are visited each time when a new bound is deduced for the variables that occur in them, was done in [THF+07]—however not taking into account the potential benefit from detecting when a clause becomes satisfied and therefore needs not to be visited anymore. Within this restriction, the use of two watched atoms is shown to accelerate the solver 2–18-fold on a set of random benchmarks, being most beneficial on formulae with large clause sizes (in the order of 200 atoms).

3.1.4 Branching by Splitting Intervals

We have seen that ICP alone will often not suffice to reach a single point solution even for purely conjunctive systems. Since the input of the solver actually consists of a conjunction of disjunctive clauses, we have a second reason to introduce some kind of branching. The classical scheme in SAT modulo theories would be to handle these two kinds of branching separately: a SAT solver selects atoms from the clauses and hands over the resulting conjunctive system to a theory solver—which in our case would be an ICP solver that would require an internal branching by splitting intervals after pruning. The approach used in iSAT is different: at all times, all variables—including the boolean ones—have a bounded interval valuation. Considering all variables equal in this respect, iSAT's only branching mechanism is *splitting intervals*. If a boolean variable's domain is $[0, 1] \subseteq \mathbb{IZ}$, thus in fact only $\{0, 1\}$, and splitting of this interval occurs, the new value immediately becomes one of the end-points, i.e. from being undecided, the variable is made either true or false by splitting—depending on whether the lower or upper part of the "interval" is selected. Continuous intervals are split, normally at their midpoint, and either the lower or upper part becomes the new valuation. We call this splitting of a variable's domain and selection of one of the possible halves a *decision*, since it is not a consequence that arises out of the formula like a deduction, but instead is a deliberate choice

to reduce the search space temporarily. Since decisions involve a degree of freedom, they can be undone and alternatives may have to be explored.

It is in fact the combination of splitting and unit propagation that allows iSAT to select a set of arithmetic constraints for ICP without explicitly choosing any atoms from the formula that should be satisfied, but instead by indirectly forcing them into ICP by making all other atoms of the clause false. This indirection has the advantage that boolean disjunctions and the more subtle disjunctions encoded in arithmetics (e.g. those caused by min or max constraints or those that arise as a result of periodic functions, which can be understood as offering multiple different regions where their value exceeds a certain threshold) are considered equally important and can be handled by the same branching mechanism. It is, however, still not clear, whether the loss of the ability to directly select atoms from the boolean structure is really outweighed by this advantage. A recent reimplementation of the iSAT algorithm[2] explores this question by also allowing the solver to explicitly select atoms. In general, the presented branching by splitting intervals is closer to approaches used in the constraint satisfaction domain, while branching by selecting atoms (and potentially splitting intervals within an isolated ICP-based theory solver) is closer to classical SAT solving as is done in SMT solvers.

Splitting Heuristics. The selection of variables to be split can be considered a research topic on its own. In the simplest static *splitting heuristics*, variables are selected in a round-robin fashion, often after sorting them e.g. by their type and / or their BMC depth (either forward or backward). Preferring boolean variables, for example, aims at satisfying that part of the formula that consists mainly of boolean atoms before trying to find a solution for the continuous part. Dynamic splitting heuristics are based e.g. on always splitting the widest interval or on splitting variables that have been found "important" e.g. in the sense that they were often involved in conflicts. For most of these splitting heuristics, a compelling story can be told why they intuitively not only make sense, but should outperform many inferior strategies. However, in practice, relative performances of heuristics vary significantly over benchmark sets and it is rather the exception when a particular heuristic leads to better computation times for a vast majority of cases. The hardness of the problem is quite obvious. Given a perfect splitting heuristic, if even only for the propositional fragment of the formula, we could find solutions to satisfiable SAT problems by always choosing the right variable and right valuation with no more decisions than variables occurring in the formula. Since the existence of such a "perfect" heuristic is as likely or unlikely as P=NP, iSAT settles for imperfect heuristics with compelling stories and some support in benchmarking. For iSAT-ODE we will later consider some reasons why splitting discrete variables first is a bit more compelling than not doing it (and measure the impact on actual benchmark instances).

3.1.5 Implication Graph and Conflict Analysis

We have discussed during our description of ICP that an important difference between deduction rules in iSAT and off-the-shelf libraries for interval computations is the identification of *reasons* for each deduction. We have also seen that

[2] https://projects.avacs.org/projects/isat3

the concentration on individual bounds instead of entire intervals can help to reduce the number of these reasons. The last classical ingredient of SAT solving that plays a significant role in iSAT is based on the recording of these reasons: *conflict analysis* and the learning of *conflict clauses*.

Whenever iSAT deduces a new bound for a variable, this new bound is stored in an *implication graph* as a node. Incoming edges are added to this new node from the bounds that are reasons for its deduction, including those bounds that caused the constraint to become unit. The implication graph is layered into *decision levels*. Each new decision is entered without reasons into the graph, opening a new decision level which keeps all subsequent deductions until a new decision is made and therefore a new decision level is opened.

If a variable's interval valuation is found to have become empty while deducing a new bound, the reasons for this deduction and those for the bound in the implication graph which conflicts it become the starting points of conflict analysis. Traversing backwards through the implication graph, the reasons for the conflict are collected until a termination criterion is reached. As often done in SAT solving, iSAT stops going backwards through the graph when only one of the reasons is still on the current decision level, i.e. when all reasons but one are not caused directly nor indirectly by the latest decision (called *unique implication point*). Again, [Her11] contains additional details on this procedure.

Since real-valued variables are often assigned more than one bound (in contrast to propositional variables in SAT), the number of bounds in the implication graph is not directly limited by the number of variables. This is another reason for limiting the size of deduction chains (like shown in Figure 3.2) by stopping at a minimum progress threshold. It should be noted that one of the additional techniques, which have been discussed for iSAT, but not been integrated, is to compact the graph by removing intermediate reasons and add their transitive reason set when adding new bounds—thereby effectively doing some parts of conflict analysis during the recording of the graph. While reducing the size of the graph, it would eagerly do some computations from conflict analysis, which would normally not be done at all if the corresponding bounds are not involved in a conflict. Even more aggressive reductions in size might be achieved by reducing the accuracy of reason sets by removing parts of the graph and replacing them with a more compact hull (containing at least some unnecessary reasons). In practice, the memory footprint of iSAT has been observed to grow significantly especially on problems with long deduction chains—indicating that some of the proposed techniques may be considered necessary in the future.

Once a set of reasons is obtained from backwards traversal through the implication graph, this set is used to generate a *conflict clause*. Since each deduction in iSAT and each split can be represented by a simple bound, the implication graph and hence the reason set consists of simple bounds, too. Conflict analysis therefore identifies a box within the search space in which the formula is not satisfiable. The idea of learning a conflict clause is to force the solver to avoid this region of the search space that has been identified to cause the current conflict (and hence is known not to contain any solution). This is achieved by negating the reasons and learning the disjunction of these negated atoms as a clause. Having e.g. reasons $(x \geq 5.4 \wedge y < 2.3)$, which together with the upper bound of x and lower bound of y from the given domain form a box, the conflict clause would be $(x < 5.4 \vee y \geq 2.3)$. A learned conflict clause will become unit whenever all but one of the reasons are satisfied, enforcing that

the opposite of the remaining reason is deduced by unit propagation. Learning a conflict clause thus amounts to excluding the box spanned by the identified reasons from the search space. Unit propagation ensures that it will never again be examined during the remainder of the search.

We already stressed that it is beneficial to keep the number of reasons for each deduction low. The motivation is much clearer now: each unnecessary bound that is collected during backwards traversal of the implication graph potentially reduces the volume of the box that can be excluded from the search space by the learned conflict clause. Similarly, the motivation for going backwards through the graph can be stressed now. Since deductions always prune valuations, bounds that come earlier in the implication graph are coarser than the ones they caused. However, going backward, we may find constraints on variables that so far did not occur in the reason set. These additional bounds will make the resulting conflict clause less general. The unique implication point technique is therefore only one of many potential techniques that may offer different trade-offs between the number of dimensions constrained in a conflict clause and the volume that it excludes.

3.1.6 The iSAT Solver

Finally, the iSAT solver can now be described by using the introduced elements and adding only a few more technical details. Figure 3.5 provides an algorithmic illustration of this description. Given variable domains and a preprocessed formula, the solver first adds the initial domains to an *implication queue*, which essentially is a list of bounds that need to be processed by deductions. The main loop then consists of deductions, conflict resolution (if necessary), and decisions by splitting intervals.

Bounds that are deduced either by unit propagation or ICP are added to the implication queue (unless they lack progress) and, together with their reasons, they are stored in the implication graph. The new upper and lower bounds then replace the current interval valuation in which the solver searches for a solution. This allows an *early conflict detection* by using the valuation in further deductions even before all consequences of the respective bound have been explored systematically. This systematic exploration happens when the bound is taken from the implication queue and the corresponding watched atoms from the formula are visited. These deductions continue until the implication queue becomes empty, i.e. the last element has been visited, but no new bounds have been entered because their progress (if any at all) does not exceed the user-defined absolute bound progress threshold.

When a deduction leads to an empty box, the implication graph is analyzed, reasons are identified, a conflict clause is learned, and just as many decisions are taken back as necessary to make this newly-learned conflict clause unit. If the conflict cannot be resolved, i.e. all of the reasons are on the zeroth decision level and hence are not based on any decision that could be taken back, the solver has proven that there is an inherent contradiction and that there is no satisfying valuation.

If no conflict was encountered, a fixed point of the deduction (with respect to the given absolute bound progress threshold parameter) has been reached. A decision is therefore done in this situation unless all variable ranges have widths below the user-defined *minimum splitting width*, which is the threshold

```
solve() {
    enqueue_domain_bounds();                    // Store domains in implication queue.
    while(true) {
        bool found_conflict = !deduce();                    // Deduce until no progress.
        if (found_conflict) {
            bool conflict_resolved = analyze_conflict();
            if (!conflict_resolved) {
                return unsat;            // There were no decisions that could be undone.
            } else {
                backjump();                    // Undo decisions. Make learned clause unit.
            }                          // No decision here. Will unit-propagate conflict clause.
        } else {
            if (all_clauses_satisfied()) {
                return satisfiable;                                 // Entire box is solution.
            }
            bool decision_done = decide();
            if (!decision_done) {                // Box size below minimum splitting width:
                return unknown;                                     // candidate solution found.
}   }   }   }
```

Figure 3.5: Abstract representation of the iSAT algorithm.

for stopping further splits. There also is a possibility after deduction that the formula is actually satisfied by each point from the remaining search space. In this case, the formula can be reported as satisfiable. Due to the issues with box consistency that we discussed earlier, it is more likely to reach a small box as a deduction fixed point without being able to prove that it contains only solutions. In this case, iSAT reports this box as a candidate solution box and cautiously calls the result "unknown".

Correctness and Termination. A detailed formal proof of correctness and termination is given in [Her11], which relieves us of the burden of fully analyzing these issues in detail, here. However, an abstract analysis of these issues is interesting and possible without any additional formalism.

Correctness is to be understood in that way that the solver does not give a wrong answer, i.e. it does not report the formula to be satisfiable when it is not satisfiable, nor does it claim unsatisfiability when in fact there is a solution. The answer "unknown", however, may be issued in both cases and even reporting a candidate solution box along with this answer is not to be understood as a claim that the formula is satisfiable.

We have discussed that ICP removes only definite non-solutions from the valuation that the solver currently explores. Similarly, we have seen that unit propagation enforces a constraint only when all other atoms in the same clause have become false. When deduction therefore leads to a conflict, a box which is described by the reasons for the conflict (no matter how far backwards traversal through the implication graph is performed) definitely does not contain any solution. The only point where parts of the search space are pruned without knowing whether they possibly contain solutions are interval splits. These, however, are stored as reasons in the implication graph and therefore become part of the learned conflict clauses if they (indirectly or directly) cause the

conflict. Since the solver only stops conflict resolution when it cannot undo any decisions (i.e. when all reasons are on the zeroth decision level), and since unit propagation will cause the negation of one of the atoms to be chosen, we can be certain that all alternatives to a decision will be explored eventually, before the solver reports a formula unsatisfiable. This result is therefore trustworthy. Satisfiability on the other hand is only reported under the (rare) circumstance that in each clause there is one atom that is satisfied by each point from the entire current valuation box. The remaining cases are those where the solver reports "unknown" upon finding a sufficiently small box which can not be pruned any further and hence for which no conflict has been detected. There is no guarantee that it contains a solution, but the solver does not claim there to be a solution in there, either.

The argument for *termination* is essentially based on the observation that the remaining search space volume decreases whenever a conflict clause is learned and that conflict clauses will always be learned eventually until either the entire search space is covered or a candidate solution (or actual solution) is encountered. Initially, the search space is bounded since all intervals are required to be closed with finite limits. From the construction of conflict clauses, we know that they describe boxes within the search space. The thresholds for deduction and splitting guarantee that bounds in the implication graph have a progress that does not converge to zero (unless the threshold parameters are chosen to be zero). Since conflict clauses are constructed from the bounds in the implication graph, also these clauses have sufficient progress, i.e. do not exclude only infinitesimally small regions. From backjumping and unit propagation, we know that after encountering a conflict, one of the bounds from the newly learned clause immediately becomes unit and the solver thus searches in a different region. A finite number of conflict clauses (covering non-negligible and differing volumes) will therefore suffice to cover the entire search space (if no solution or candidate solution is encountered before). Neither deduction chains nor sequences of splits can be infinite, since both are stopped when a minimum width has been reached—again under the assumption that the thresholds are chosen to be non-zero. Since only a finite number of conflict clauses is learned, the formula size does not grow infinitely either. Additional clauses allow additional deduction steps, but since only a finite number of conflict clauses will be generated and between the generation of two conflict clauses only finitely many deductions and decisions can occur for non-zero thresholds, the solver will need only finite time to enumerate all these conflict clauses and will therefore terminate eventually.

Example 5 (iSAT algorithm)**.** Figure 3.6 shows an iSAT constraint system rewritten into the internal format and an illustration of the corresponding search space. The first two lines describe the domain of the variables. Afterwards, the constraint system asks for solution points to lie on both, the sine and the cosine curve. The following constraints can be understood as implications of the form $a \rightarrow y \geq 0.25$. Variable a can be understood as a trigger for the constraint on the right-hand side of the implication: if a is set to true, the simple bound $y \geq 0.25$ must be satisfied as well. The last line requires at least one of the two boolean variables to satisfied. Solutions are therefore those intersections of the sine and cosine graphs, which lie in one of the dark gray regions—marking those parts which are described by the triggered simple bounds—but not in the

Variables:
$x, y \in [-10, 10]$
$a, b \in \{0, 1\}$
Formula:
$(y = \sin(x))$
$\wedge\ (y = \cos(x))$
$\wedge\ (\neg a \vee y \geq\ 0.25)$
$\wedge\ (a \vee y \leq -0.25)$
$\wedge\ (\neg b \vee x \geq -4)$
$\wedge\ (\neg b \vee x \leq -2)$
$\wedge\ (b \vee x \geq 2)$
$\wedge\ (b \vee x \leq 4)$
$\wedge\ (a \vee b)$

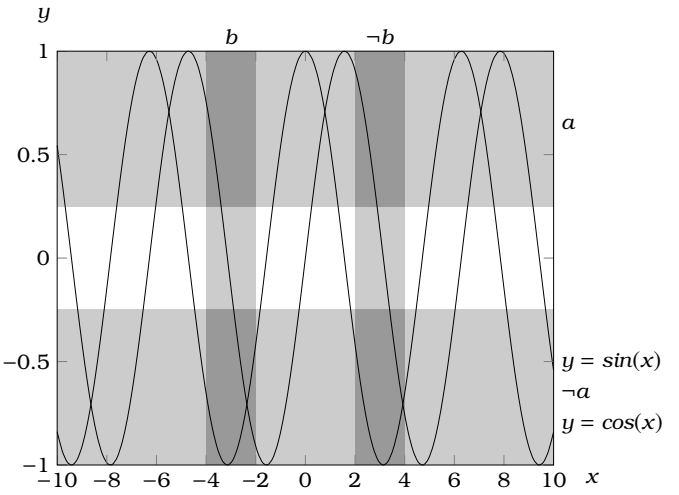


Figure 3.6: Core iSAT formula after rewriting into internal format and corresponding search space.

lower-right one of them since there, neither a nor b can be true. Inspecting the search space visually, one can easily see a single solution in the lower-left dark gray region.

In Figures 3.7–3.12, the first steps done by the iSAT algorithm are illustrated. The solver starts on decision level 0, deducing the initial domains and thereby pruning the range of y from $[-10, 10]$ to $[-1, 1]$—a consequence of $y = \sin(x)$ and $y = \cos(x)$ being unit. After these deductions, no further prunings are possible and iSAT decides to split the range of variable b. Since b is a boolean variable, this split leads to a point valuation of 0, signifying b being false. The clauses $(b \vee x \geq 2)$ and $(b \vee x \leq 4)$ become unit, constraining the range of x. The last clause of the constraint system $(a \vee b)$ requires a to become true, which in turn makes $(\neg a \vee y \geq 0.25)$ unit. Since y has already been found to have a range well in the negative numbers, a conflict is detected. In this case, backwards traversal through the implication graph leads to (b) as a conflict clause: the solver has learned that choosing $\neg b$ as a valuation never leads to a solution and therefore excludes this choice permanently from the solution space by learning this conflict clause. Only one decision needs to be undone (in general, the *backjump distance* may be larger) to cause the new clause to become unit. In the second step, again on decision level 0, the consequences are deduced. This restricts x to $[-4, -2]$ and y to a corresponding range covering the intersection of $\sin([-4, -2])$ and $\cos([-4, -2])$. In step 3, x has been split and the lower half has been selected for further analysis. Deductions quickly expose a conflict, since valuations for $\sin([-4, -3])$ lie outside the range already deduced for y. Again, a conflict clause is learned, and its consequences applied as interval prunings in step 4. We only show one more step in Figure 3.12, where again a split occurs and its consequences are explored.

Stronger pruning mechanisms might very well be able to detect at this point the existence of a single valuation within the identified bounds. The deduction rules for sine and cosine functions in iSAT are, however, not very strong since they lack a number of possible inverse propagations. Therefore, iSAT needs

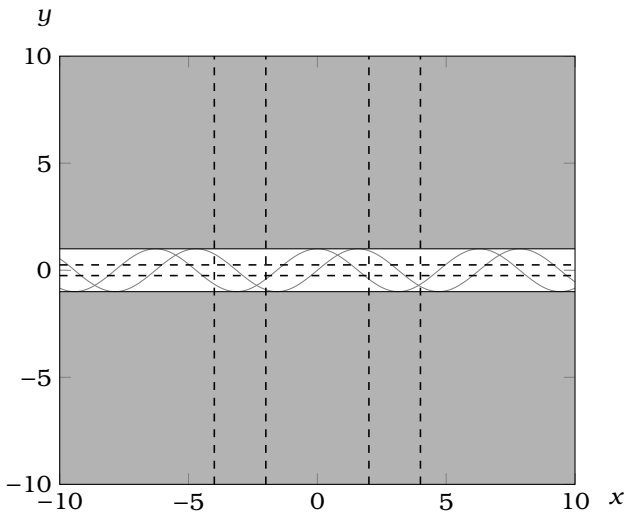


Figure 3.7: iSAT example, step 0, decision level 0: deductions caused by unit clauses and domain bounds.

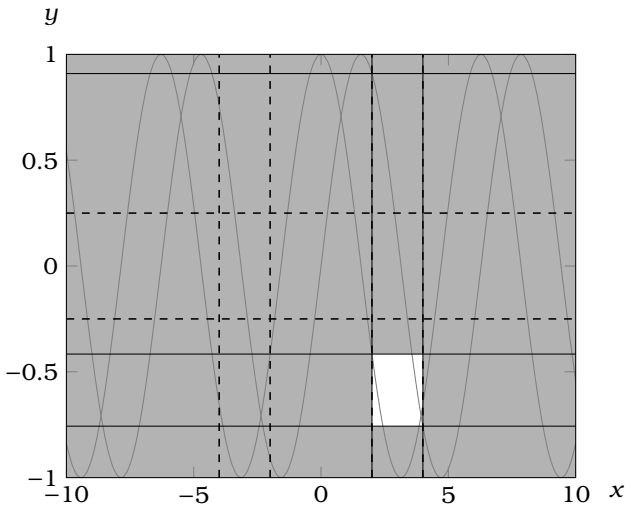
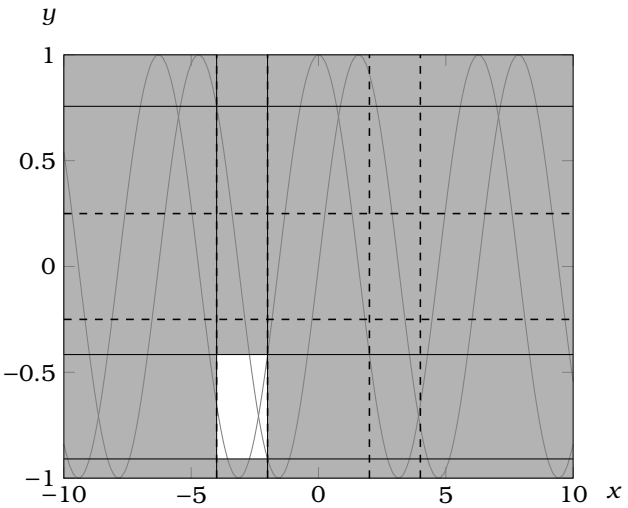
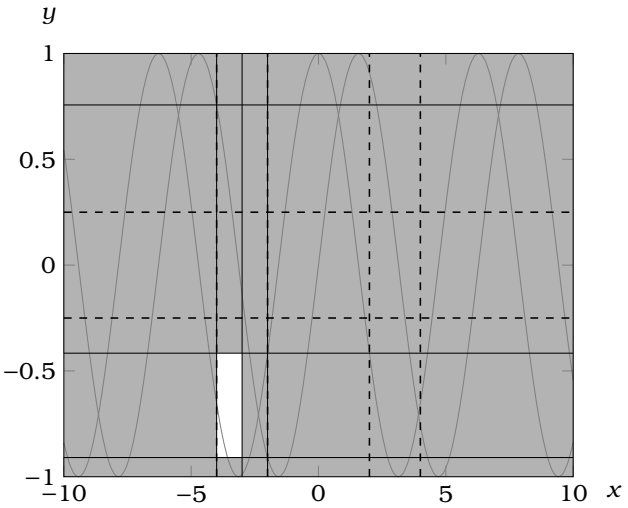


Figure 3.8: iSAT example, step 1, decision level 1, the decision to assign b to false causes a deduction chain that leads to a becoming true, causing conflicting bounds on y. The solver learns the conflict clause $(\neg b)$. To resolve the conflict, a backjump to decision level 0 is performed. Deductions that lack progress are written in parentheses.



Decision level: 0
Decision:
(none)
Deductions:
$b \geq 1$ (due to conflict clause)
$x \geq -4$
$x \leq -2$
$(x \leq -2)$
$y \geq -0.90929743$
$y \leq 0.7568025$
$y \geq -0.41614684$
$(x \geq -4)$
$a \leq 0$

Figure 3.9: iSAT example, step 2, decision level 0, deducing the consequences of the learned conflict clause.



Decision level: 1
Decision:
$x \leq -3$
Deduction:
$(x \geq -4)$
Conflict:
$y = \sin(x)$ has no solution inside box

Figure 3.10: iSAT example, step 3, decision level 1: splitting of range of x leads to a conflict during deduction. Learned conflict clause is $(x > -3 \vee y > -0.416146837 \vee x < -4)$. Backjump to decision level 0.

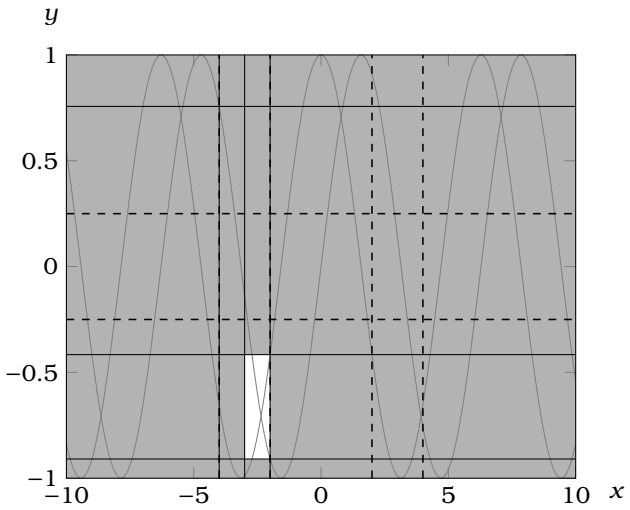


Figure 3.11: iSAT example, step 4, decision level 0: deduction after backjump. The newly learned conflict clause leads to an exploration of the previous decision's alternative.

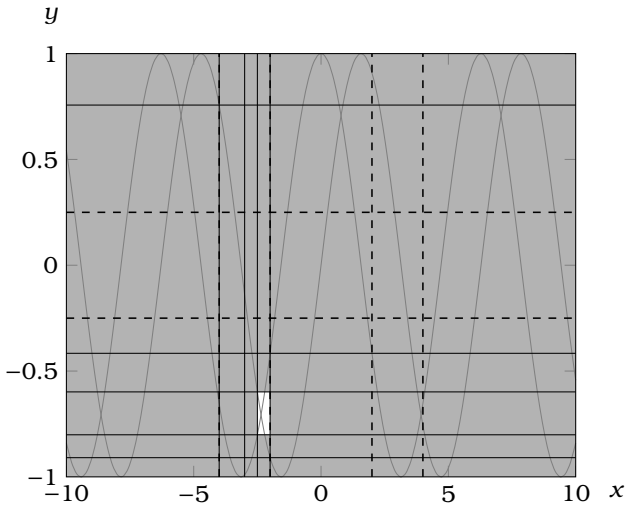


Figure 3.12: iSAT example, step 5, decision level 1: splitting domain of x and subsequent deductions.

a total of 11 decisions and 8 conflicts to find a candidate solution box for a minimum splitting width of 0.01 and an absolute bound progress of 0.001. The chosen splitting heuristic in this case is to always split the widest interval.

3.2 Enclosing Solution Sets of Ordinary Differential Equations

In Section 2.4, a brief overview of enclosure methods for ODEs has been given. For a sufficiently well-founded argument, why these methods can be used to handle ODE constraints and flow invariants within a SAT modulo ODE solver, we first need to supply some additional mathematical background. Thereafter, we will show its application within the VNODE-LP library for computing enclosures of set-valued initial value problems of ordinary differential equations—laying out the second main ingredient of iSAT-ODE.

Example 6 (approximative numerical integration)**.** As an introductory example to illuminate the mathematical approach, consider the harmonic oscillator system of ODEs, which we have already used as an example in Figure 2.3.

$$\begin{aligned}\frac{dx}{dt}(t) &= -y(t)\\ \frac{dy}{dt}(t) &= \quad x(t)\end{aligned}$$

Many simple types of numerical integration methods for ODEs are based on a *Taylor series* expansion of the (unknown) solution function.

$$\begin{aligned}x(t_0+h) &= \frac{h^0}{0!}\frac{d^0x}{dt^0}(t_0) + \frac{h^1}{1!}\frac{d^1x}{dt^1}(t_0) + \frac{h^2}{2!}\frac{d^2x}{dt^2}(t_0) + \cdots\\ y(t_0+h) &= \frac{h^0}{0!}\frac{d^0y}{dt^0}(t_0) + \frac{h^1}{1!}\frac{d^1y}{dt^1}(t_0) + \frac{h^2}{2!}\frac{d^2y}{dt^2}(t_0) + \cdots\end{aligned}$$

Luckily, by using the right-hand sides of the ODEs, these derivatives can be computed without knowing the solution function itself. For x, the first derivatives are thus:

$$\begin{aligned}\frac{d^0x}{dt^0}(t_0) &= x(t_0)\\ \frac{d^1x}{dt^1}(t_0) &= \frac{dx}{dt}(t_0)\\ &= -y(t_0) \quad \text{(using the ODE's right-hand side)}\\ \frac{d^2x}{dt^2}(t_0) &= \frac{d(\frac{dx}{dt})}{dt}(t_0)\\ &= \left(\frac{\partial(\frac{dx}{dt})}{\partial x}\cdot\frac{dx}{dt} + \frac{\partial(\frac{dx}{dt})}{\partial y}\cdot\frac{dy}{dt}\right)(t_0)\\ &= \left(\frac{\partial(-y)}{\partial x}\cdot\frac{dx}{dt} + \frac{\partial(-y)}{\partial y}\cdot\frac{dy}{dt}\right)(t_0)\\ &= (0\cdot(-y) + (-1)\cdot(x))\,(t_0)\\ &= -x(t_0)\end{aligned}$$

$$\begin{aligned}
\frac{d^3x}{dt^3}(t_0) &= \frac{d(\frac{d^2x}{dt^2})}{dt}(t_0) \\
&= \left(\frac{\partial(\frac{d^2x}{dt^2})}{\partial x}\cdot\frac{dx}{dt} + \frac{\partial(\frac{d^2x}{dt^2})}{\partial y}\cdot\frac{dy}{dt}\right)(t_0) \\
&= \left(\frac{\partial(-x)}{\partial x}\cdot\frac{dx}{dt} + \frac{\partial(-x)}{\partial y}\cdot\frac{dy}{dt}\right)(t_0) \\
&= (-1\cdot(-y) + 0\cdot(x))\,(t_0) \\
&= y(t_0)
\end{aligned}$$

Similarly, the first derivatives of y can be computed.

$$\begin{aligned}
\frac{d^0y}{dt^0}(t_0) &= y(t_0) \\
\frac{d^1y}{dt^1}(t_0) &= \frac{dy}{dt}(t_0) \\
&= x(t_0) \quad \text{(using the ODE's right-hand side)} \\
\frac{d^2y}{dt^2}(t_0) &= \frac{d(\frac{dy}{dt})}{dt}(t_0) \\
&= \left(\frac{\partial(\frac{dy}{dt})}{\partial x}\cdot\frac{dx}{dt} + \frac{\partial(\frac{dy}{dt})}{\partial y}\cdot\frac{dy}{dt}\right)(t_0) \\
&= \left(\frac{\partial(x)}{\partial x}\cdot\frac{dx}{dt} + \frac{\partial(x)}{\partial y}\cdot\frac{dy}{dt}\right)(t_0) \\
&= (1\cdot(-y) + 0\cdot(x))\,(t_0) \\
&= -y(t_0) \\
\frac{d^3y}{dt^3}(t_0) &= \frac{d(\frac{d^2y}{dt^2})}{dt}(t_0) \\
&= \left(\frac{\partial(\frac{d^2y}{dt^2})}{\partial x}\cdot\frac{dx}{dt} + \frac{\partial(\frac{d^2y}{dt^2})}{\partial y}\cdot\frac{dy}{dt}\right)(t_0) \\
&= \left(\frac{\partial(-y)}{\partial x}\cdot\frac{dx}{dt} + \frac{\partial(-y)}{\partial y}\cdot\frac{dy}{dt}\right)(t_0) \\
&= (0\cdot(-y) + (-1)\cdot(x))\,(t_0) \\
&= -x(t_0)
\end{aligned}$$

This only works when the right-hand side of the ODE is sufficiently often differentiable, i.e. when there are at least as many derivatives as the order to which the series is to be developed. Since the ODE system in this example has analytic right-hand sides, there is no restriction in this case.
Truncating the Taylor series at order zero would give a constant evolution:

$$\begin{aligned}
x(t_0 + h) &= \frac{h^0}{0!}x(t_0) = x(t_0) \\
y(t_0 + h) &= \frac{h^0}{0!}y(t_0) = y(t_0).
\end{aligned}$$

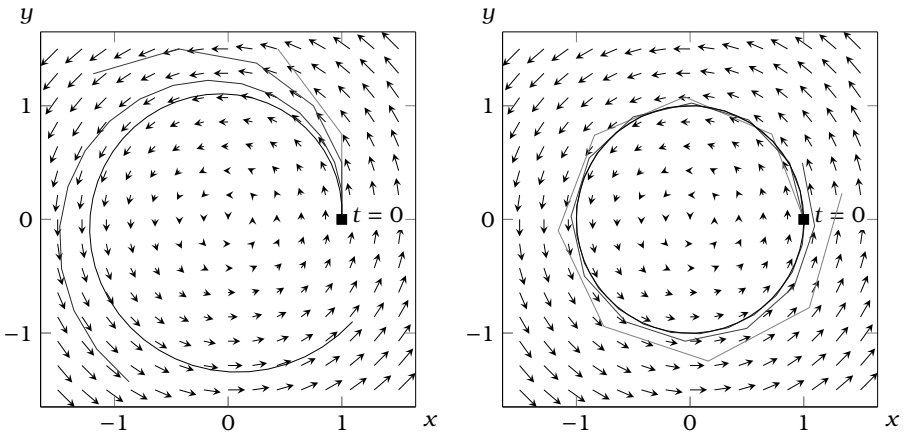


Figure 3.13: Approximative numerical integration. Left: Euler's method applied to the harmonic oscillator with step sizes of $h \in \{0.125, 0.25, 0.5, 0.75\}$. Graphs for smaller step sizes have been drawn in darker shade. Right: Second order Taylor expansion with the same step sizes.

This is obviously quite useless. However, increasing the order by one, we get

$$x(t_0 + h) = \frac{h^0}{0!}x(t_0) + \frac{h^1}{1!}\frac{dx}{dt}(t_0) = x(t_0) + h \cdot (-y(t_0))$$
$$y(t_0 + h) = \frac{h^0}{0!}y(t_0) + \frac{h^1}{1!}\frac{dy}{dt}(t_0) = y(t_0) + h \cdot (x(t_0)),$$

which yields a linear approximation, commonly referred to as *Euler's method*. It is the most intuitive numerical approximation to the solution: starting from the current point $(x(t_0), y(t_0))$, we compute the slope in each dimension and apply this slope as if it stayed constant for the entire duration h of the integration step. In the left part of Figure 3.13, we plot the approximated trajectories resulting from using this linear approximation when starting with the initial point $(1, 0)$.

In most cases, higher approximation orders yield better approximations. We illustrate this by also computing the second-order approximation.

$$x(t_0 + h) = \frac{h^0}{0!}x(t_0) + \frac{h^1}{1!}\frac{dx}{dt}(t_0) + \frac{h^2}{2!}\frac{d^2x}{dt^2}(t_0) = x(t_0) + h \cdot (-y(t_0)) + \frac{h^2}{2} \cdot (-x(t_0))$$
$$y(t_0 + h) = \frac{h^0}{0!}y(t_0) + \frac{h^1}{1!}\frac{dy}{dt}(t_0) + \frac{h^2}{2!}\frac{d^2y}{dt^2}(t_0) = y(t_0) + h \cdot (x(t_0)) + \frac{h^2}{2} \cdot (-y(t_0))$$

Looking at the graphs obtained from using this second-order approximation in the right part of Figure 3.13, one can easily see that they are much closer to the actual solution function for the same step sizes. It should not be left unsaid that in practical methods of higher approximation order, like e.g. a fourth-order Runge-Kutta method, the direct computation of higher order Taylor coefficients is replaced by a linear combination of substeps with pre-computed weight coefficients—saving a significant number of computation steps.

3.2.1 From Approximations to Enclosures

A fundamental problem arises when thinking about the application of numerical methods within a framework which forbids us to prune solutions from the search space. How do we know, which approximation order and which step size is "good enough"? How far away is the approximation from the actual solution, which must not be pruned away?

In "traditional" numerics, these questions are answered by an analysis of the convergence rates and estimates of the size of the error, often given in the form of an O-notation in the step size. In "validated" numerics, the answer to these questions leads directly to interval methods and guaranteed error bounds. A thorough discussion of the underlying principles of the validated approach can already be found in [Moo66]. Here, we try to follow the spotlights insofar as they illuminate our subsequent discussion of VNODE-LP.

Truncation Error of Taylor Series. The choice of a Taylor series expansion in the above example was not incidental. While more sophisticated methods exist for approximating ODE solution trajectories, Taylor series have the distinct advantage of offering a simple description of their truncation error. Given a system of first-order ODEs

$$\begin{pmatrix} \dot{x_1} \\ \vdots \\ \dot{x_n} \end{pmatrix} = \begin{pmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{pmatrix} \tag{3.1}$$

the Taylor series expansion up to order κ with the corresponding *Lagrange remainder term* of its solution is given by

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}(t_0 + h) = \begin{pmatrix} \left(\sum_{i=0}^{\kappa} \frac{h^i}{i!} \cdot \frac{d^i x_1}{dt^i}(t_0)\right) + \frac{h^{\kappa+1}}{(\kappa+1)!} \cdot \frac{d^{\kappa+1} x_1}{dt^{\kappa+1}}(t_0 + \vartheta_1 \cdot h) \\ \vdots \\ \left(\sum_{i=0}^{\kappa} \frac{h^i}{i!} \cdot \frac{d^i x_n}{dt^i}(t_0)\right) + \frac{h^{\kappa+1}}{(\kappa+1)!} \cdot \frac{d^{\kappa+1} x_n}{dt^{\kappa+1}}(t_0 + \vartheta_n \cdot h) \end{pmatrix}$$

with unknown $\vartheta_i \in (0, 1)$. Just like in the example, the derivatives of the unknown solution are given by the right-hand sides of the ODEs and their derivatives—again assuming their existence up to at least order $\kappa + 1$.

$$\begin{pmatrix} x_1 \\ \\ \vdots \\ x_n \\ \\ \end{pmatrix}(t_0 + h) = \begin{pmatrix} x_1(t_0) + \left(\sum_{i=1}^{\kappa} \frac{h^i}{i!} \cdot \frac{d^{i-1} f_1}{dt^{i-1}}(x_1(t_0), \ldots, x_n(t_0))\right) \\ \quad + \frac{h^{\kappa+1}}{(\kappa+1)!} \cdot \frac{d^{\kappa} f_1}{dt^{\kappa}}(x_1(t_0 + \vartheta_1 \cdot h), \ldots, x_n(t_0 + \vartheta_1 \cdot h)) \\ \vdots \\ x_n(t_0) + \left(\sum_{i=1}^{\kappa} \frac{h^i}{i!} \cdot \frac{d^{i-1} f_n}{dt^{i-1}}(x_1(t_0), \ldots, x_n(t_0))\right) \\ \quad + \frac{h^{\kappa+1}}{(\kappa+1)!} \cdot \frac{d^{\kappa} f_n}{dt^{\kappa}}(x_1(t_0 + \vartheta_n \cdot h), \ldots, x_n(t_0 + \vartheta_n \cdot h)) \end{pmatrix}$$

The zeroth order term for dimension j is simply the value $x_j(t_0)$, while the i-th-order terms are replaced by the $(i-1)$-th derivatives of the ODEs' right-hand sides with respect to t in the point $(x_1(t_0), \ldots, x_n(t_0))$. The remainder term is the most crucial aspect: since it must be evaluated at time $t_0 + \vartheta_j \cdot h$, in this rewriting, f_j must be evaluated at a future point of the unknown solution trajectory. To know the truncation error exactly, we would need to know this point—in which case we would have no need for an approximation in the first place. However,

an evaluation of the remainder term over a box that contains at least all possible trajectories will yield an overapproximation of the truncation error and hence an enclosure of the trajectory. In order to compute valid error bounds of the truncated Taylor series, we hence first need a valid enclosure of the trajectory to evaluate the remainder term on. While this sounds not very helpful, as a matter of fact, it is. Since the remainder term has $h^{\kappa+1}/(\kappa+1)!$ as a coefficient, even a quite conservative enclosure suffices to get a relatively small error term and hence a much tighter enclosure than the one that was used to compute the error bound. Moore utilized this to build a two step process, which modern algorithms are essentially still based upon—albeit using different methods within the two steps. First, an *a-priori enclosure* and a corresponding step size for which it contains all solution trajectories are computed. This coarse enclosure is then used in the second step to compute the remainder term of a higher-order series expansion.

Computing A-Priori Enclosures. The approach introduced by Moore [Moo66] and refined later by Lohner [Loh88] and Stauning [Sta97] to obtain a coarse first enclosure with which the truncation error can be overestimated is at its core based on the *Picard-Lindelöf existence and uniqueness theorem* for the solutions of initial value problems, which makes use of *Lipschitz continuity* and contraction properties on *Banach spaces*—mathematical background topics whose detailed discussion would lead us too far away from the subject of this thesis, in which we will mostly rely on this theory as an invisible bedrock.

In the following paragraphs, we will therefore only summarize the approach from [Loh88, pp. 27ff.] and [Moo66, pp. 92ff.]. Given the system of ODEs from Equation (3.1), we require $\vec{f}$ to be at least p-times differentiable on the open subset $\vec{D} \subseteq \mathbb{R}^n$ and the initial value to be $\vec{x}(0) = \vec{x}_0 \in \vec{D}$.

We introduce interval extensions $\vec{F} = (F_1, \ldots, F_n)$ of the right-hand sides $f_1, \ldots, f_n$ and require the interval extension $\vec{F}$ to be defined and continuous on D and to be *inclusion monotonic*, i.e. $\vec{F}(\vec{Y}') \subseteq \vec{F}(\vec{Y})$ for $\vec{Y}' \subseteq \vec{Y}$. Using the interval extension from Definition 10, which we have introduced in the previous section, inclusion monotonicity follows from construction when tightest hulls are used. Then $\vec{F}(\vec{Y}') = \text{hull}(\{\vec{f}(\vec{y}') \mid \vec{y}' \in \vec{Y}'\})$ and $\vec{F}(\vec{Y}) = \text{hull}(\{\vec{f}(\vec{y}) \mid \vec{y} \in \vec{Y}\})$, and consequently $\vec{F}(\vec{Y})$ must be a superset of $\vec{F}(\vec{Y}')$ whenever $\vec{Y} \supseteq \vec{Y}'$.[3]

Most centrally, we need a property that can be understood as a lifting of *Lipschitz continuity* (a function f is Lipschitz continuous on $D \subseteq \mathbb{R}$ if $\exists L \in \mathbb{R}_{>0} \forall y_1, y_2 \in D : \|f(y_1) - f(y_2)\| \leq L \cdot \|y_1 - y_2\|$) to the interval representation. Let $w(I)$ be the width of an interval I, i.e. $w(I) = \sup(I) - \inf(I)$, then there shall be a real number $L > 0$ such that for all $\vec{Y} \subseteq \vec{D}$ the width of $\vec{Y}$'s image under $\vec{F}$ is bounded by the maximum component width of $\vec{Y}$ scaled by L, i.e.

$$\forall (Y_1, \ldots, Y_n) = \vec{Y} \subseteq \vec{D} : \quad w(\vec{F}(\vec{Y})) \;\leq\; L \;\cdot \max_{j \in \{1,\ldots,n\}} w(Y_j).$$

This property may in fact be restrictive, since it is not satisfied where $\vec{f}$ has unbounded slope, e.g. for an $f(x) = \sqrt{x}$ at $x = 0$. For the purpose of this thesis, however, we may safely assume the right-hand sides of ODEs to satisfy this

[3] A practical implementation with outward rounding of floating-point borders will satisfy this criterion as well, since rounding errors captured for $\vec{Y}'$ would also be made for the larger $\vec{Y}$. Allowing arbitrary overapproximations, inclusion monotonicity might be violated, when the enclosure for the subset's results contains (spurious) values that are not enclosed for the larger set's results. These additional values would however only worsen the tightness of the computed bounds, but not lead to incorrect pruning of actual solutions, despite violating the definition of inclusion monotonicity.

property on the entire range admissible by the flow invariants, by all functions either being analytic and all variables having bounded ranges or by the modeler taking care about this explicitly by introducing flow invariants that exclude regions with unbounded slope or undefinedness.

If, under these constraints, there is a box $\vec{U} \subseteq \vec{D}$ such that

$$\vec{B} := \vec{x}_0 + [0, h] \cdot \vec{F}(\vec{U}) \subseteq \vec{U},$$

the initial value problem consisting of the ODE from Equation 3.1 and $\vec{x}(0) = \vec{x}_0$ has a unique solution $\vec{x} : [0, h] \to \vec{B}$, i.e. $\vec{B}$ is a *bounding box* that contains the solution for all $t \in [0, h]$.

Fortunately, this result is even stronger, since the point initial condition can be replaced by an interval box, effectively forming a set of initial value problems, with the *initial set* being a box $\vec{X}_0$. This result is labeled as "Satz (1.2.3.3)" in [Loh88] and proven there. Thus, having

$$\vec{B} := \vec{X}_0 + [0, h] \cdot \vec{F}(\vec{U}) \subseteq \vec{U},$$

all the initial value problems for each $\vec{x}_0 \in \vec{X}_0$ have a unique solution on $t \in [0, h]$, which is bounded by $\vec{B}$.

Only this applicability of the result to interval initial conditions allows the use of this method iteratively, since we can in general not expect an enclosure to stay point-valued—and hence have to take the entire enclosure after one integration step as starting set for the next.

Example 7 (rudimentary enclosure)**.** We can now attempt a first enclosure of the solutions of a set of a initial value problems. Again using the harmonic oscillator with $\dot{x} = -y$ and $\dot{y} = x$, we want to enclose the solutions originating from $X_0 = [0.9, 1.1]$ and $Y_0 = [-0.1, 0.1]$ at time $t = 0$. The right-hand sides can be summarized into a function $\vec{f}(x, y) = (-y, x)$, with $\vec{F}$ being its interval extension. First, we then compute an a-priori enclosure. Evidently, the initial set must be contained, i.e. we could choose

$$\vec{U}_0 := \begin{pmatrix} [0.9, 1.1] \\ [-0.1, 0.1] \end{pmatrix}$$

and compute

$$\begin{aligned} & \begin{pmatrix} X_0 \\ Y_0 \end{pmatrix} + [0, h] \cdot \vec{F}(\vec{U}_0) \\ = & \begin{pmatrix} [0.9, 1.1] \\ [-0.1, 0.1] \end{pmatrix} + [0, h] \cdot \begin{pmatrix} [-0.1, 0.1] \\ [0.9, 1.1] \end{pmatrix} \\ = & \begin{pmatrix} [0.9, 1.1] + [-0.1h, 0.1h] \\ [-0.1, 0.1] + [0, 1.1h] \end{pmatrix} \\ = & \begin{pmatrix} [0.9 - 0.1h, 1.1 + 0.1h] \\ [-0.1, 0.1 + 1.1h] \end{pmatrix} \\ \overset{?}{\subseteq} & \vec{U}_0 \end{aligned}$$

The subset relationship only holds for $h = 0$. Quite obviously, the choice of $\vec{U}_0$ as a candidate bounding box which covers just the initial values, only allows a

step of length zero, a result that can very well be expected—to move along the dynamics for any non-trivial time, we will most likely leave the initial region. We therefore increase the size of $\vec{U}_0$

$$\vec{U}_0' := \begin{pmatrix} [0.5, 1.5] \\ [-0.5, 0.5] \end{pmatrix}$$

and repeat the computation:

$$\begin{aligned} & \begin{pmatrix} X_0 \\ Y_0 \end{pmatrix} + [0, h] \cdot \vec{F}(\vec{U}_0') \\ = & \begin{pmatrix} [0.9, 1.1] \\ [-0.1, 0.1] \end{pmatrix} + [0, h] \cdot \begin{pmatrix} [-0.5, 0.5] \\ [0.5, 1.5] \end{pmatrix} \\ = & \begin{pmatrix} [0.9, 1.1] + [-0.5h, 0.5h] \\ [-0.1, 0.1] + [0, 1.5h] \end{pmatrix} \\ = & \begin{pmatrix} [0.9 - 0.5h, 1.1 + 0.5h] \\ [-0.1, 0.1 + 1.5h] \end{pmatrix} \\ \overset{?}{\subseteq} & \vec{U}_0' \end{aligned}$$

Now, h can be chosen larger than zero, more precisely, the following conditions must be satisfied for the subset relationship to hold:

$$\begin{array}{rcrcl} 0.9 - 0.5h \geq 0.5 & \leftrightarrow & -0.5h \geq -0.4 & \leftrightarrow & h \leq 0.8 \\ 1.1 + 0.5h \leq 1.5 & \leftrightarrow & 0.5h \leq 0.4 & \leftrightarrow & h \leq 0.8 \\ -0.1 \geq -0.5 & \leftrightarrow & \mathit{true} & & \\ 0.1 + 1.5h \leq 0.5 & \leftrightarrow & 1.5h \leq 0.4 & \leftrightarrow & h \leq 0.2\bar{6} \end{array}$$

We can now choose $h = 0.2$, and get

$$\vec{B}_0 = \begin{pmatrix} [0.9 - 0.5 \cdot 0.2, 1.1 + 0.5 \cdot 0.2] \\ [-0.1, 0.1 + 1.5 \cdot 0.2] \end{pmatrix} = \begin{pmatrix} [0.8, 1.2] \\ [-0.1, 0.4] \end{pmatrix} \subseteq \begin{pmatrix} [0.5, 1.5] \\ [-0.5, 0.5] \end{pmatrix} = \vec{U}_0{}'$$

With this $\vec{B}_0$, we have a bounding box for the first step and with $h_0 = 0.2$, we have a suitable step size for which the theoretical results guarantee that this a-priori enclosure contains all solutions which originate from the initial box.

Recalling the already computed terms of the truncated Taylor series

$$\begin{aligned} x(t_0 + h) &= x(t_0) + h \cdot (-y(t_0)) + \frac{h^2}{2} \cdot (-x(t_0)) \\ y(t_0 + h) &= y(t_0) + h \cdot (x(t_0)) + \frac{h^2}{2} \cdot (-y(t_0)), \end{aligned}$$

for an enclosure, we add the error term

$$\begin{aligned} X(t_0 + h) &= X(t_0) + h \cdot (-Y(t_0)) + \frac{h^2}{2} \cdot (-X(t_0)) + \frac{h^3}{3!} \cdot [-0.1, 0.4] \\ Y(t_0 + h) &= Y(t_0) + h \cdot (X(t_0)) + \frac{h^2}{2} \cdot (-Y(t_0)) + \frac{h^3}{3!} \cdot [-1.2, -0.8] \end{aligned}$$

since $\frac{d^3x}{dt^3}(\vartheta) = y(\vartheta)$ and $Y(\vartheta) = [-0.1, 0.4]$, and since $\frac{d^3y}{dt^3}(\vartheta) = -x(\vartheta)$ and $X(\vartheta) = [0.8, 1.2]$. We can now compute an enclosure at time $h = 0.2$.

$$X(0.2) = [0.9, 1.1] + 0.2 \cdot [-0.1, 0.1] + 0.02 \cdot [-1.1, -0.9] + [0.001^3_4] \cdot [-0.1, 0.4]$$

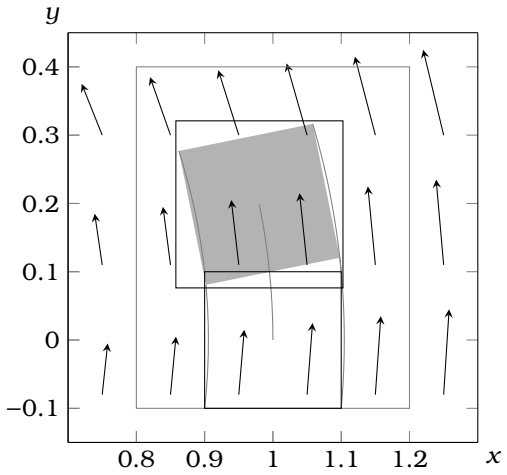


Figure 3.14: First step of a rudimentary ODE enclosure, showing the initial box $[0.9, 1.1] \times [-0.1, 0.1]$, the a-priori enclosure $[0.8, 1.2] \times [-0.1, 0.4]$, the enclosure at time $t = 0.2$, $[0.85786, 1.10256] \times [0.07632, 0.32096]$, and the exact solution set (gray, filled).

$$
\begin{aligned}
&= [0.9, 1.1] + [-0.02, 0.02] + [-0.022, -0.018] + [-0.00014, 0.00056] \\
&= [0.85786, 1.10256] \\
Y(0.2) &= [-0.1, 0.1] + 0.2 \cdot [0.9, 1.1] + 0.02 \cdot [-0.1, 0.1] + [0.001_4^3] \cdot [-1.2, -0.8] \\
&= [-0.1, 0.1] + [0.18, 0.22] + [-0.002, 0.002] + [-0.00168, -0.00104] \\
&= [0.07632, 0.32096]
\end{aligned}
$$

In Figure 3.14, we illustrate this first enclosure step. The filled gray area shows the exact solution set. The comparison with the computed enclosure for $t = 0.2$ illustrates two kinds of overapproximation. Dominating is the effect of using an axis-parallel box to enclose a solution set that is not aligned with these coordinate axes. This *wrapping effect* will be the subject of the next subsection. Not fighting it would mean that the next enclosure step would have to start with an already enlarged initial set—and further rotation would increase the volume of the enclosure in each successive step significantly. Far less pronounced—but still non-negligible for this low order Taylor expansion—is the effect of bounding the truncation error, which can also be seen numerically in the widths of the computed remainder terms. In the illustration, the computed enclosure is marginally larger than an ideal axis-aligned box around the exact solution set.

Using the computed box for $t = 0.2$, a naive enclosure algorithm could iteratively compute a successor by using this enclosure as initial set, computing a new a-priori enclosure and step size and thereby bounding the remainder term of this second step. It is however obvious from this first step alone that more has to be done to arrive at a tighter enclosure.

3.2.2 Fighting the Wrapping Effect

Figure 3.14 illustrates the inadequacy of using axis-parallel enclosures on the example of a harmonic oscillator. Moore [Moo66] (despite then seemingly

not yet having given the name to the phenomenon) used the same example to quantitatively analyze the impact of this wrapping effect, showing that the width of the enclosure grows by a factor of approximately 500 per revolution in the limit $h \to 0$. His approach to mitigating the wrapping effect is based on a simple observation: if the solution set rotates, so should the coordinate system with respect to which the enclosure is represented. Consequently, a coordinate transformation should be performed, i.e. a transformation matrix be computed that yields a rotated (and potentially sheared) coordinate system in every step with respect to which the enclosure is given. While this approach, a predecessor of the *parallelepiped method* in later algorithms, suffices for pure rotations like the harmonic oscillator system in our example, Moore also notes that the transformation matrix may become singular (and the method may thus break down) after some time (cf. [Moo66, p.136]). This problem and the wrapping effect in general is analyzed in greater detail by Lohner [Loh88] and he crucially extends the approach by adding an *orthogonalization* step, using the Q matrix resulting from applying *QR-factorization* on the transformation matrix. To minimize the amount of overapproximation incurred by no longer exactly following the changed orientation, the dimensions of the transformation matrix are sorted prior to orthogonalization by the widths of the enclosure in that dimension. Thereby, the dimension with the largest interval width retains its optimal orientation.

The direct interval evaluation used so far has an additional weakness that needs to be addressed at the same time. Consider e.g. just a truncated first-order Taylor expansion $X(t_0+h) = X(t_0) + h \cdot F(X(t_0))$, which is part of the rudimentary enclosure generation shown above. Since this computation consists of a sum of intervals, the successor enclosure $X(t_0+h)$ will be at least as wide as $X(t_0)$, even if the solution set actually decreases in width.[4] If a symbolic representation of the derivative were used like one would do in manually computed examples or as done in the prototypical work in [Egg06], this overapproximation could be partially avoided by first simplifying the Taylor terms before interval evaluation. The canonical path, however, is to choose a *midpoint representation*, which consists of a point solution and one or more interval vectors which hold the accumulated enclosure widths.

Looking at the fully general problem with coordinate transformations and changed enclosure representations leads to a rather convoluted formalism that does not lend itself very well to illustrating the method. Luckily, Nedialkov and Jackson [NJ01] have summarized the approach to a degree that is still relatively digestible by constraining the ODEs to the case of linear systems with constant coefficients. In the following paragraphs, we report their presentation of the approach and point out that the temporary limitation of generality is done for illustration purposes only.

Let an initial value problem of a linear system of ODEs with constant coefficients be defined as

$$\dot{\vec{x}} = \mathcal{B} \cdot \vec{x} \quad \text{and} \quad \vec{x}(t_0) \in \vec{X}_0$$

with $\mathcal{B} \in \mathbb{R}^{n \times n}$ an $n \times n$-matrix. The initial transformation matrix is $\mathcal{A}_0 = \mathcal{I}$, the identity matrix. Additionally, we need $\hat{\vec{x}}_0 = m(\vec{X}_0)$ and $\vec{s}_j = m(\vec{Z}_j)$, with $\vec{Z}_j$ denoting the error bounds computed using the a-priori enclosure in step j

[4] $w([a+c, b+d]) = (b+d) - (a+c) = (b-a) + (d-c) = w([a,b]) + w([c,d]) \geq w([a,b])$

and $m(\vec{Z}_j)$ denoting the midpoint of this box. In practice, these midpoints may not be representable, which can e.g. be overcome most easily by using a tight interval instead. The initial value for the box $\vec{R}_0$ is set to $\vec{0}$ for point-valued initial conditions and needs to be set to $\vec{X}_0 - \hat{\vec{x}}_0$ for an initial box $\vec{X}_0$. Due to the linearity of the ODE and under the assumption of having a constant step size h (again only for illustration purposes), the Taylor series expansion can be simplified to a matrix product with a constant matrix $\mathcal{T}$:

$$\mathcal{T} := \mathcal{T}_{k-1}(h\mathcal{B}) := \sum_{i=0}^{k-1} \frac{(h\mathcal{B})^i}{i!},$$

such that

$$\mathcal{T} \cdot \vec{x}(t_j) = \sum_{i=0}^{k-1} \frac{h^i}{i!}\mathcal{B}^i\vec{x}(t_j) = \sum_{i=0}^{k-1} \frac{h^i}{i!}\frac{d^i\vec{x}}{dt^i}(t_j),$$

since

$$\begin{aligned}
\frac{d^0\vec{x}}{dt^0}(t_j) &= \mathcal{B}^0 \cdot \vec{x}(t_j) \\
\frac{d^1\vec{x}}{dt^1}(t_j) &= \mathcal{B}^1 \cdot \vec{x}(t_j) \\
\frac{d^2\vec{x}}{dt^2}(t_j) &= \frac{d\mathcal{B}\vec{x}}{dt}(t_j) = \mathcal{B}\frac{d\vec{x}}{dt} = \mathcal{B}\left(\mathcal{B}\vec{x}\right)(t_j) = \mathcal{B}^2\vec{x}(t_j) \\
&\vdots
\end{aligned}$$

With these prerequisites, the solution enclosure $\vec{X}_j$ at t_j can be represented as

$$\vec{X}_j = \mathcal{T}\hat{\vec{x}}_{j-1} + (\mathcal{T}\mathcal{A}_{j-1})\vec{R}_{j-1} + \vec{Z}_j,$$

with a transformation matrix $\mathcal{A}_j$ and

$$\vec{R}_j = (\mathcal{A}_j^{-1}\mathcal{T}\mathcal{A}_{j-1})\vec{R}_{j-1} + \mathcal{A}_j^{-1}(\vec{Z}_j - \vec{s}_j).$$

The midpoint trajectory $\hat{\vec{x}}_j$ occurring in this computation is iterated by

$$\hat{\vec{x}}_j = \mathcal{T}\hat{\vec{x}}_{j-1} + \vec{s}_j,$$

i.e. it is adjusted slightly by the midpoint $\vec{s}_j$ of the local error term $\vec{Z}_j$, which is absorbed into $\vec{R}_j$ in the next step to avoid wrapping.

In this linear case, the computation of the transformation matrix in the parallelepiped method is then

$$\mathcal{A}_j = \mathcal{T}\mathcal{A}_{j-1},$$

whereas in the more general case it is the solution of the variational equation along the solution function $\vec{x}(t)$

$$\frac{d\mathcal{A}}{dt}(t) = \frac{\partial f}{\partial \vec{x}}\mathcal{A}(t) = \mathcal{J}\mathcal{A}(t),$$

with $\mathcal{J}$ being the Jacobian matrix consisting of the partial derivatives of $\vec{f}$ with respect to all $\vec{x}$. In Lohner's algorithm, this solution is approximated by using a truncated Taylor series and computed as an interval enclosure.

Example 8 (parallelepiped method)**.** Before looking at the orthogonalization, we compute an enclosure with the parallelepiped method for the harmonic oscillator example. The system can easily be represented in the linear format.

$$\dot{\vec{x}} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \mathcal{B}\vec{x}$$

In order for the coordinate transformation to be effective not only on the accumulated error, but also on the entire initial box, the set of initial values $[0.9, 1.1] \times [-0.1, 0.1]$ needs to be captured in the $\vec{R}_0$ box, which is given with respect to the initial transformation matrix $\mathcal{A}_0 = \mathcal{I}$. The midpoint of the original box is stored in $\hat{\vec{x}}_0$.

$$\hat{\vec{x}}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \vec{R}_0 = \begin{pmatrix} [-0.1, 0.1] \\ [-0.1, 0.1] \end{pmatrix}$$

Previously, we have already computed a valid a-priori enclosure $\vec{B}_0$ for the first time step from t_0 to $t_1 = t_0 + h = 0 + 0.2$, which we can now use again to compute $\vec{Z}_0$. Its midpoint is then stored in $\vec{s}_0$. We still want to truncate the Taylor series after the second-order term, hence set $k = 3$, which is the order of the remainder term.

$$\begin{aligned}
\mathcal{B}^2 &= \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}^2 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \\
\mathcal{B}^3 &= \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \\
\vec{Z}_0 &= \frac{(h\mathcal{B})^3}{3!} \vec{B}_0 = \frac{(h\mathcal{B})^3}{3!} \begin{pmatrix} [0.8, 1.2] \\ [-0.1, 0.4] \end{pmatrix} = \frac{0.008}{6} \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} [0.8, 1.2] \\ [-0.1, 0.4] \end{pmatrix} \\
&= \frac{0.008}{6} \begin{pmatrix} [-0.1, 0.4] \\ [-1.2, -0.8] \end{pmatrix} = \begin{pmatrix} [-0.000134, 0.000534] \\ [-0.0016, -0.001066] \end{pmatrix} \\
\vec{s}_0 &= \begin{pmatrix} 0.0002 \\ -0.00133 \end{pmatrix}
\end{aligned}$$

The first step can now be computed.

$$\begin{aligned}
\mathcal{T} &= \mathcal{T}_{\kappa-1}(h\mathcal{B}) = \frac{h^0\mathcal{B}^0}{0!} + \frac{h^1\mathcal{B}^1}{1!} + \frac{h^2\mathcal{B}^2}{2!} \\
&= \mathcal{I} + h\mathcal{B} + \frac{h^2}{2}\mathcal{B}^2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & -h \\ h & 0 \end{pmatrix} + \begin{pmatrix} -h^2/2 & 0 \\ 0 & -h^2/2 \end{pmatrix} \\
&= \begin{pmatrix} 1 - h^2/2 & -h \\ h & 1 - h^2/2 \end{pmatrix} = \begin{pmatrix} 0.98 & -0.2 \\ 0.2 & 0.98 \end{pmatrix} \\
\vec{X}_1 &= \mathcal{T}\hat{\vec{x}}_0 + (\mathcal{T}\mathcal{A}_0)\vec{R}_0 + \vec{Z}_0 \\
&= \begin{pmatrix} 0.98 & -0.2 \\ 0.2 & 0.98 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \left(\begin{pmatrix} 0.98 & -0.2 \\ 0.2 & 0.98 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right) \begin{pmatrix} [-0.1, 0.1] \\ [-0.1, 0.1] \end{pmatrix} \\
&\quad + \begin{pmatrix} [-0.000134, 0.000534] \\ [-0.0016, -0.001066] \end{pmatrix} \\
&= \begin{pmatrix} 0.98 \\ 0.2 \end{pmatrix} + \underbrace{\begin{pmatrix} 0.98 & -0.2 \\ 0.2 & 0.98 \end{pmatrix}}_{=\mathcal{A}_1} \begin{pmatrix} [-0.1, 0.1] \\ [-0.1, 0.1] \end{pmatrix} + \begin{pmatrix} [-0.000134, 0.000534] \\ [-0.0016, -0.001066] \end{pmatrix}
\end{aligned} \tag{3.2}$$

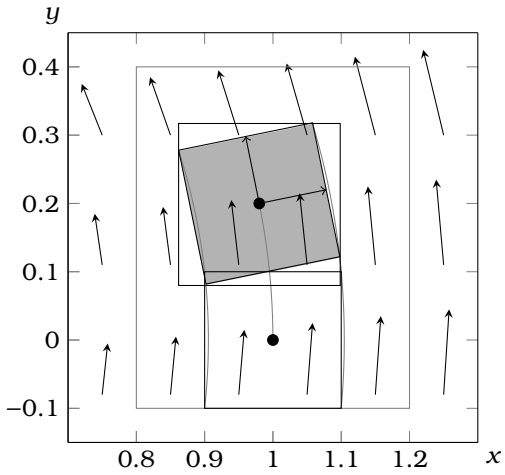


Figure 3.15: First step of the coordinate transformation, applying the parallelepiped method to the harmonic oscillator example.

$$
\begin{aligned}
&= \begin{pmatrix} 0.98 \\ 0.2 \end{pmatrix} + \begin{pmatrix} [-0.118, 0.118] \\ [-0.118, 0.118] \end{pmatrix} + \begin{pmatrix} [-0.000134, 0.000534] \\ [-0.0016, -0.001066] \end{pmatrix} \\
&= \begin{pmatrix} [0.861866, 1.098534] \\ [0.0804, 0.316934] \end{pmatrix}
\end{aligned}
$$

In Equation (3.2), the enclosure $\vec{X}_1$ is represented by a sum of a (mid-)point, a box with respect to a transformed coordinate system, and a small error term. We depict this representation in Figure 3.15. This decomposed representation allows the operations to work on points and point matrices and only in the end combines the results with interval computations. In a practical implementation, intervals will also occur during these point-operations to capture rounding errors, but this additional uncertainty can be moved into the interval entities.

The benefit of this representation then becomes manifest during iteration, since the successor enclosure $\vec{X}_j$ is not computed using the locally wrapped enclosure $\vec{X}_{j-1}$, but is instead freshly combined from the components.

It should also be noted, that the object described by Equation (3.2) is in fact a zonotope, consisting of a midpoint, two generator vectors (which form the basis $\mathcal{A}_j$) with corresponding intervals, and a (small) box, with which the object's boundary is finally expanded, when this small box is centered around zero, which can easily be achieved by moving the midpoint accordingly. Since the restriction to the linear case can be overcome when using the variational equation, this method is not as confined as the zonotope-based approaches discussed in Section 2.4. This relationship has been exploited in [MCRTM14] to extend zonotope-based computations of reachable state sets for hybrid systems to non-linear ODEs.

To prepare the next step, we need to compute $\vec{R}_1$ and the next midpoint $\hat{\vec{x}}_1$. To do this, we first need a new a-priori enclosure $\vec{B}_1$, which is then used to compute $\vec{Z}_1$. For this manual calculation, we simply generously expand the locally-wrapped enclosure $\vec{X}_1$ and check whether the expansion suffices to cover

the time step from $t_1 = 0.2$ to $t_2 = t_1 + 0.2$.

$$\begin{aligned}
\vec{B}_1 &= \begin{pmatrix}[0.861866, 1.098534]\\ [0.0804, 0.316934]\end{pmatrix} + [0, 0.2] \cdot \vec{F}\left(\begin{pmatrix}[0.4, 1.5]\\ [0.0, 0.8]\end{pmatrix}\right)\\
&= \begin{pmatrix}[0.861866, 1.098534]\\ [0.0804, 0.316934]\end{pmatrix} + [0, 0.2] \cdot \begin{pmatrix}[-0.8, 0.0]\\ [0.4, 1.5]\end{pmatrix}\\
&= \begin{pmatrix}[0.861866, 1.098534]\\ [0.0804, 0.316934]\end{pmatrix} + \begin{pmatrix}[-0.16, 0.0]\\ [0.0, 0.3]\end{pmatrix}\\
&= \begin{pmatrix}[0.701866, 1.098534]\\ [0.0804, 0.616934]\end{pmatrix} \subseteq \begin{pmatrix}[0.4, 1.5]\\ [0.0, 0.8]\end{pmatrix}\\
\vec{Z}_1 &= \frac{hB^3}{3!}\vec{B}_1 = \begin{pmatrix}[0.0001072, 0.000823]\\ [-0.001465, -0.000936]\end{pmatrix}\\
\vec{s}_1 &= \begin{pmatrix}0.000464\\ -0.001200\end{pmatrix}
\end{aligned}$$

We also need to invert the transformation matrix, which in practice amounts to computing an enclosure of this inverse since not all numbers are necessarily representable. Implementations can be based either on the kind of exact rational arithmetic used here in the manual case, on lifting the entirety of computations to intervals, or on first computing an approximative result and then enlarging it by a small ε-region until the product of $\mathcal{A}$ and the computed enclosure of the inverse contains the identity matrix.

$^{98}/_{100}$	$^{-1}/_{5}$	1	0
$^{1}/_{5}$	$^{98}/_{100}$	0	1
$^{2501}/_{2450}$	0	1	$^{20}/_{98}$
$^{4}/_{98}$	$^{1}/_{5}$	0	$^{20}/_{98}$
1	0	$^{2450}/_{2501}$	$^{500}/_{2501}$
$^{4}/_{98}$	$^{1}/_{5}$	0	$^{20}/_{98}$
1	0	$^{2450}/_{2501}$	$^{500}/_{2501}$
0	$^{1}/_{5}$	$^{-100}/_{2501}$	$^{490}/_{2501}$
1	0	$^{2450}/_{2501}$	$^{500}/_{2501}$
0	1	$^{-500}/_{2501}$	$^{2450}/_{2501}$
1	0	$[0.9796^{1}_{0}]$	$[0.1999^{3}_{2}]$
0	1	$[-0.1999^{2}_{3}]$	$[0.9796^{1}_{0}]$

$$\begin{aligned}
\mathcal{A}_1^{-1} &\in \begin{pmatrix}[0.97960, 0.97961] & [0.19992, 0.19993]\\ [-0.19993, -0.19992] & [0.97960, 0.97961]\end{pmatrix}\\
\vec{R}_1 &= (\mathcal{A}_1^{-1}\mathcal{T}\mathcal{A}_0)\vec{R}_0 + \mathcal{A}_1^{-1}(\vec{Z}_1 - \vec{s}_1)\\
&= \left(\begin{pmatrix}[0.97960, 0.97961] & [0.19992, 0.19993]\\ [-0.19993, -0.19992] & [0.97960, 0.97961]\end{pmatrix}\begin{pmatrix}0.98 & -0.2\\ 0.2 & 0.98\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\right)\\
&\quad \cdot \begin{pmatrix}[-0.1, 0.1]\\ [-0.1, 0.1]\end{pmatrix}\\
&\quad + \begin{pmatrix}[0.97960, 0.97961] & [0.19992, 0.19993]\\ [-0.19993, -0.19992] & [0.97960, 0.97961]\end{pmatrix}\\
&\quad \cdot \left(\begin{pmatrix}[0.0001072, 0.000823]\\ [-0.001465, -0.000936]\end{pmatrix} - \begin{pmatrix}0.000464\\ -0.001200\end{pmatrix}\right)
\end{aligned}$$

$$
\begin{aligned}
&= \begin{pmatrix} [0.999992, 1.0000038] & [-0.0000004, 0.0000114] \\ [-0.0000114, 0.0000004] & [0.999992, 1.0000038] \end{pmatrix} \begin{pmatrix} [-0.1, 0.1] \\ [-0.1, 0.1] \end{pmatrix} \\
&\quad + \begin{pmatrix} [0.97960, 0.97961] & [0.19992, 0.19993] \\ [-0.19993, -0.19992] & [0.97960, 0.97961] \end{pmatrix} \begin{pmatrix} [-0.0003568, 0.000359] \\ [-0.000265, 0.000264] \end{pmatrix} \\
&= \begin{pmatrix} [-0.10000152, 0.10000152] \\ [-0.10000152, 0.10000152] \end{pmatrix} + \left([-0.00041, 0.00041] [-0.000332, 0.00033] \right) \\
&= \begin{pmatrix} [-0.10042, 0.10042] \\ [-0.10034, 0.10034] \end{pmatrix}
\end{aligned}
$$

As can be seen, the size of $\vec{R}_1$ has grown only marginally and a significant part of this growth is due to the low precision with which we enclosed the inverse transformation matrix $\mathcal{A}_1^{-1}$ and of course the low order κ of the Taylor expansion. In practice, when post-decimal digits are a plus and do not eat valuable space on a page, these intervals can be made much tighter, reducing the growth of $\vec{R}_1$ even further.

Finally, we only need to compute a new midpoint enclosure to be able to iterate the method.

$$
\begin{aligned}
\hat{\vec{x}}_1 &= \mathcal{T}\hat{\vec{x}}_0 + \vec{s}_1 \\
&= \begin{pmatrix} 0.98 \\ 0.2 \end{pmatrix} + \begin{pmatrix} 0.000464 \\ -0.001200 \end{pmatrix} = \begin{pmatrix} 0.980464 \\ 0.1988 \end{pmatrix}
\end{aligned}
$$

QR-Factorization. As could be clearly seen in the example, in order to apply coordinate transformations of the presented kind, it is necessary to not only compute a suitable transformation matrix, but also to enclose its inverse. The existence of this inverse is paramount for the method to work—its non-existence, called *singularity* of the transformation matrix, must therefore be avoided. Even worse, the method already breaks down when $\mathcal{A}_j$ becomes *nearly singular*. Singularity can among others be measured by the linear independence of the column vectors, which can be graphically understood as the angle between the axes spanned by these vectors no longer being orthogonal, but more and more aligned. Nedialkov and Jackson [NJ01] analyze the case of linear ODE systems by looking at the eigenvalues of the matrices and derive conditions, which allow to predict whether singularity becomes a problem. For our purposes, it suffices to say that singular transformation matrices occur often enough to merit a countermeasure, Lohner's orthogonalization via QR-factorization.

Again, we follow closely the presentation from [NJ01], which summarizes Lohner's original approach [Loh88]. Having a transformation matrix $\mathcal{A}_{j-1}$ of the $j-1$-th step, its successor under $\mathcal{T}$ is computed and decomposed

$$\mathcal{T}\mathcal{A}_{j-1} = Q_j R_j$$

and then Q_j is chosen as transformation matrix for the j-th step:

$$\mathcal{A}_j = Q_j.$$

The decomposition itself is described in detail e.g. in [PTVF92]. Most importantly, Q_j is orthogonal, i.e. its column vectors are pairwise orthogonal, which is the actual goal in this context. Also, orthogonalization causes the inverse to coincide with the transposed matrix, i.e. $Q^{-1} = Q^T$.

By adding a permutation matrix $\mathcal{P}_j$, the columns of $\mathcal{TA}_{j-1}$ can be re-sorted prior to orthogonalization. This reordering is done in such a way that the longest edge of the parallelepiped is not changed through the QR-decomposition.

3.2.3 Automatic Differentiation

In our illustrating examples, we have computed the necessary derivatives in the traditional way using symbolic mathematics and term simplifications as necessary. While under the impression of today's strong computer algebra systems and large computational resources this approach might actually have become a viable option for an automatized method, the approach employed in Lohner's implementation as in VNODE-LP does not rely on symbolic mathematics with potentially huge simplification efforts, but instead makes use of *automatic differentiation*. The central idea behind automatic differentiation is to replace the normal arithmetic evaluation rules by ones that compute the value of the derivative instead of the value of the expression, leading to derivatives that can be computed with nearly the same computational effort as computing the expression itself.

Consider e.g. the expression $x^2 + cyx + a$. Automatic differentiation like normal evaluation is done on a per-atom basis, i.e. each arithmetic operation is handled individually by local application of derivation rules. In what is called the *forward mode*, the following evaluations would be added to compute the derivative of the above expression with respect to x.

$e_1 = x$	$e_1' = 1$ (seeded, want derivative w.r.t. x)
$e_2 = e_1^2$	$e_2' = 2e_1^{2-1}e_1' = 2x$
$e_3 = c$	$e_3' = 0$ (seeded, not derivative w.r.t. c)
$e_4 = y$	$e_4' = 0$ (seeded, not derivative w.r.t. y)
$e_5 = e_3e_4$	$e_5' = e_3'e_4 + e_3e_4' = 0$
$e_6 = e_5e_1$	$e_6' = e_5'e_1 + e_5e_1' = 0 + e_3e_4 = cy$
$e_7 = e_2 + e_6$	$e_7' = e_2' + e_6' = 2x + cy$
$e_8 = a$	$e_8' = 0$ (seeded, not derivative w.r.t. a)
$e_9 = e_7 + e_8$	$e_9' = e_7' + e_8' = \underline{2x + cy}$

While this type of extended evaluation could be achieved by implementing functions for the derivatives of basic operators and suitable nested calls, modern incarnations of automatic differentiation make heavy use of generic programming and operator overloading, thereby making the computation of a derivative nearly transparent. The FADBAD++ library [Sta97], which is used in VNODE-LP, offers so-called forward and backward modes and directly supports the generation of Taylor coefficients via automatic differentiation over nearly arbitrary numerical data types—including interval variables—as long as these data types offer the required overloaded arithmetic operators.

3.2.4 VNODE-LP

We have introduced all major building blocks needed to compute enclosures of sets of initial value problems. An a-priori enclosure is computed, obtaining conservative bounds over which the error term of a Taylor series can be

```
1  #include <iostream>
2  #include <fstream>
3  #include "vnode.h"
4
5  using namespace std;
6
7  template <typename var_type>
8  void f(int num_dims,
9         var_type* fx,          // result f(x)
10        const var_type* x,     // input x
11        var_type t,            // time variable
12        void* param) {         // parameters
13   fx[0] = -1.0 * x[1];        // dx_0/dt = -x_1
14   fx[1] =        x[0];        // dx_1/dt =  x_0
15 }
16 //------- (skipping output methods & calls)
17 int main() {
18   const int n = 2;            // two dimensions
19   iVector x(n);
20   x[0] = interval(9,11) / 10.0; // X_0 = [ 0.9, 1.1]
21   x[1] = interval(-1,1) / 10.0; // X_1 = [-0.1, 0.1]
22   interval t = 0;             // start time
23   interval t_end = 20;        // maximum time
24   vnodelp::AD* ad= new vnodelp::FADBAD_AD(n, f, f);
25   vnodelp::VNODE* solver= new vnodelp::VNODE(ad);
26   solver->setOneStep(vnodelp::on);
27   while(t != t_end) {
28     solver->integrate(t, x, t_end);
29     iVector a_priori(n);
30     interval Tj;
31     a_priori = solver->getAprioriEncl();
32     Tj = solver->getT();
33   }
34 }
```

Figure 3.16: Harmonic oscillator implemented in C++ program calling the VNODE-LP solver.

evaluated—leading to a tighter enclosure. Coordinate transformations are used to avoid the wrapping effect and with it an explosion of the enclosure's width. Orthogonalization is applied to avoid the coordinate transformation to fail. Structurally, VNODE-LP [Ned06] works exactly in this way, but in each of these steps it uses a better technological alternative. A-priori enclosures are computed using a higher order [NJP01] (instead of the first-order enclosure presented above). Taylor series results are refined by an Interval Hermite-Obreschkoff method [Ned99], which works in a predictor / corrector scheme (cf. [Ned06, Chapter 20] for precise details). Instead of choosing one type of coordinate transformation, VNODE-LP uses three complementary representations, one without coordinate transformation, one based on the parallelepiped method, and one based on the QR-factorization.

Since, apart from a few exceptions, we use VNODE-LP mostly as a black box and believe the fundamental approach to be well represented by the simpler technologies introduced above, we will not detail the more precise methods, but instead focus on matters of interfacing and computed results. These topics are most relevant for the question of how VNODE-LP can be used within a SAT modulo ODE solver—the topic of the next section.

Example 9 (computing enclosure with VNODE-LP)**.** In Figure 3.16, we encode the problem of computing an enclosure for the harmonic oscillator example using VNODE-LP. In order to make use of automatic differentiation, the right-hand side of the ODE needs to be given as a template function, here function *f*, whose variable type can be chosen by VNODE-LP—allowing the use of derivative and Taylor term computations over interval data types. The ODE is simply encoded by normal C++ arithmetic operators over the components of x and stored into the result argument *fx*. The function signature allows the passing of a parameter vector *param*, intended to allow setting constant parameters. Within the main function, we first initialize the state vector as an interval box with the same initial state as in the previous examples, $[0.9, 1.1] \times [-0.1, 0.1]$. Since these boundaries are not representable by floating-point numbers, we choose to initialize *x[0]* with $[9, 11]/10$, which is being handled as a safely-rounded interval operation by the interval library (in our case filib++). The initial time is

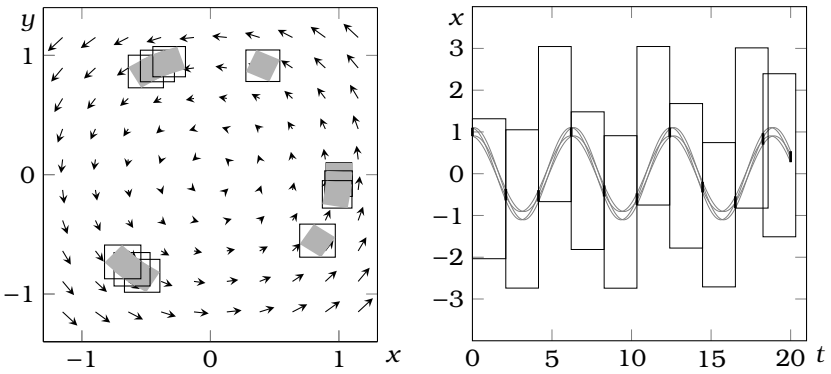


Figure 3.17: Output computed by VNODE-LP for the harmonic oscillator example. On the left, we show the (locally-wrapped) enclosures at the time-points computed by VNODE-LP and compare them with the exact solutions in gray. This phase-space plot is complemented on the right by a plot of the enclosures and a-priori enclosures in the x-dimension (0-th dimension in the program code) over the time. Thick lines at time points are tight enclosures, large thin boxes are the corresponding a-priori enclosures which cover the entire time steps (and sometimes are valid beyond the range finally used by VNODE-LP for the step). Gray lines follow the evolution of the set's four corner points.

set to $t = 0$ and an end time $t\_end = 20$ is given until which integration shall be performed. VNODE-LP is initialized by first handing the ODE's right-hand-side function to an automatic differentiation object using the FADBAD++ library and then handing this object to a *VNODE* solver object. Using the *setOneStep* method, we instruct VNODE-LP to stop after every integration step, such that we can extract the computed bounds at the end time of that step, the a-priori enclosure via *getAprioriEncl*, and the corresponding time span for which it is valid via the *getT* method. We have omitted the printing methods from this code, which we use to generate the graphs in Figure 3.17.

As can be seen from the example, VNODE-LP's intended use case is to compute a tight enclosure of a set of initial value problems over a hard-coded ODE for a given point of time. To get there, it computes enclosures for automatically determined intermediate time points and also gives access to the used step sizes and a-priori enclosures, which together capture the entire time span from the beginning to the final enclosure. These bounding boxes are, however, quite conservative. In Figure 3.17, on the right, the a-priori enclosures for the x-dimension are shown and can clearly be seen to exceed the actual state space covered by the exact solution trajectories by a large margin. On the other hand, when one is interested in computing an enclosure over intervals of time, the end time up to which integration shall be performed, may in fact be given as an interval. As long as this interval is not too wide, i.e. fits well into the step size that VNODE-LP computes, a tighter enclosure may be obtained for this interval of time. To cover the entire time span, however, the solver may have to be restarted many times, leading to significant computation times. In the next section, we will therefore investigate a less expensive evaluation scheme, which is needed in our context.

Solver Parametrization. The order of the series expansion can be set up to a maximum that needs to be specified at compile time. In [Ned06, p. 33] the amount of CPU time needed to compute an enclosure is measured for different orders, showing that, for that analyzed example, a local optimum lies between 15 and 23, depending on the desired accuracy (which influences the step size computation). For our purposes, we use the proposed default values of $\kappa = 20$ as order and absolute and relative tolerances of 10^{-12}. A solver parameter can be used to set a minimum step size, which indirectly controls the level of precision that can be reached. This parameter is set from the outside, as will be detailed in the next section. VNODE-LP supports different interval libraries—making use of FADBAD++'s flexibility of supported data types. We use the filib++ library [LTG+06] since it supports extended error handling for operations that lead to undefined results instead of aborting the computation, which happens in case of the PROFIL/BIAS library. This recovery is important since in our context, individual computations may very well fail, but later queries may still lead to useful results.

3.3 Embedding ODE Enclosures into iSAT

So far, this chapter has focused on the two fundamental building blocks that now need to be brought together to tackle the problem of solving SAT modulo ODE formulae. The major contribution of this thesis, which has been the basis of our scientific contributions, e.g. in [EFH08, ERNF12a], lies in combining constraint solving for boolean combinations of arithmetic constraints with enclosure computation for sets of initial value problems of ODEs. In this section, we therefore describe the fundamental idea of how to make use of ODE enclosures within iSAT, while the remaining sections of this chapter highlight important extensions that improve this interplay, namely the storing of learned enclosures and the computation of bracketing systems, which can lead to tighter results on non-linear ODE problems.

3.3.1 Structure of the iSAT-ODE Solver

To anchor the description of our approach, we first give an abstract overview of the iSAT-ODE solver structure, its data dependencies, and control flow in the diagram shown in Figure 3.18. We should expressly state that this diagram is neither complete nor perfectly accurate in all details, but should be understood more as a high-level view on the components and their interplay, whose refinement is the task of the remainder of this chapter.

As in the case of the core iSAT, also the extended solver's front end parses the input formula and generates a syntax tree. To the preprocessing routines from iSAT, which are applied to this representation of the input problem, a new one is added which filters out the ODE and flow invariant constraints and registers them with the ODE solver. In the graph, they are replaced by freshly-introduced trigger variables, and the correspondence between these triggers and the ODE and flow invariant constraints they represent is stored in the ODE solver. As a result of this extended preprocessing step, the iSAT front end again only has to deal with the syntax graph of a formula that no longer contains any ODE-related nodes and can therefore be processed in the accustomed manner.

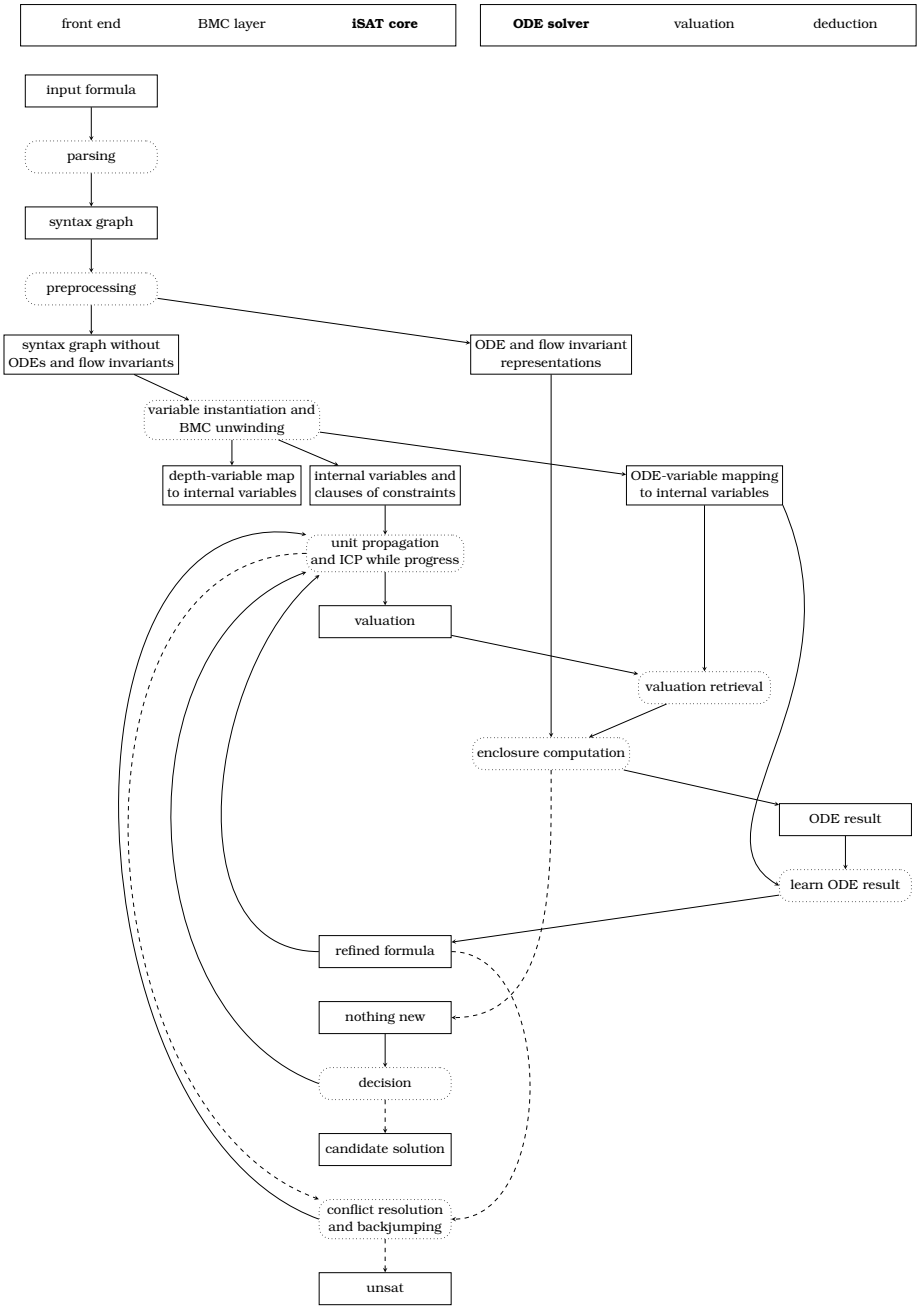


Figure 3.18: High-level structural and procedural view of iSAT-ODE. Ordering from top to bottom indicates temporal order except in cases of iterations and alternatives. Nodes in the graph with dotted outline indicate actions rather than results or data structures. Arrows have overloaded semantics: they indicate results from actions and usage of results in actions; additionally, when taking the vertical ordering of actions into account, they show the control flow. Dashed arrow lines indicate alternative control flow, e.g. ICP leading to a conflict instead of a usable valuation.

The unwinding of the formula in the bounded model checking layer of the solver leads to an instantiation of variables in the iSAT core and of a k-fold unwinding of the constraints in the transition system. At the same time, the BMC layer itself keeps a mapping of the symbolic variable names as used in the formula and their $k + 1$ instances in the solver core. These are used for reporting candidate solutions with respect to the original variable names and unwinding depths instead of their obscure internal identifiers. Here, an extension is necessary to also inform the ODE solver about the variable instances. This is done via a newly introduced valuation object in the ODE solver which captures a mapping of variables occurring in ODEs and of the trigger variables for ODE and flow invariant constraints to their respective instances in the solver core.

After BMC unwinding, the iSAT core applies deduction based on the initial domains using unit propagation and ICP as discussed extensively at the beginning of this chapter. During this step, the solver investigates an increasingly smaller interval valuation, pruning away definite non-solutions. If deduction leads to an empty valuation, a conflict is encountered (which can only be resolved if there are decisions that can be undone via backjumping—else, the formula is proven unsatisfiable and the solver terminates). More interestingly from the perspective of explaining the interplay with the ODE solver is the case when deductions cease to produce tighter bounds (cut-off by the discussed progress threshold).

Having found a valuation that is a consistent (near-)fixed point under unit propagation and ICP over the arithmetic constraints in the iSAT core, the ODE solver is asked to provide deductions for the ODE constraints. We will detail later the choices that are possible in this situation, but from the abstract perspective we currently take, the solver either detects that no new bounds can be deduced (leading iSAT to perform a decision and, with only a small interval width left, to report a candidate solution) or it computes an enclosure. Not producing a deduction happens especially when ICP has led to a valuation that has previously been observed and for which deduction results have already been learned.

To compute an enclosure, the stored mapping of symbolic variables occurring in ODE constraints to their iSAT counterparts is used together with a chosen BMC depth for which a deduction shall be performed. This information suffices to extract from the solver core the relevant bounds, a projection to the ODE-related variables, for which an enclosure is to be computed. The computation itself will require some explanation, too, but obviously it makes use of the approach, which we introduced in the previous section. As a result, a deduction object is generated, which can be understood as a set of implications, which have a part of the current valuation as premise and bounds derived from the computed enclosure as conclusion. Again using the stored mapping of symbolic variable names and depth information to core variable instances, the deduction can be applied by adding new clauses to the problem. The next section will focus on this aspect of making the deduction persistent. ODE-related deductions may also lead to conflicts, in which case learning these clauses may require conflict resolution and backjumping or even lead directly to the detection of unsatisfiability. If, on the other hand, the learned facts are consistent with the current valuation, ICP and unit propagation can be resumed to propagate the new bounds further through the arithmetic and boolean parts of the constraint system.

This embedding of ODE propagation into iSAT can very well be considered to be close to a standard theory integration in an SMT scheme, however with the

very important difference that the base solver is not a pure SAT solver, but is capable of handling arithmetic reasoning and will accept learned facts that go beyond merely propositional clauses.

3.3.2 Extended Solver Input

Starting with the solver's input, we can now discuss the details of the approach.

To avoid repeating the exposition of the entire input language, we have already added the ODE-related syntax to the description presented in Subsection 3.1.2. However, we have been quite superficial in describing some of these elements and have not yet discussed their rewriting into the internal format. In the declaration part of the formula, two special variables need to be introduced for any iSAT-ODE model. These are the *delta_time* and *time* variables, which both must have a range starting at zero. The *delta_time* variable represents the duration of a flow, i.e. takes the role of the δ variable introduced in our BMC encoding in Definition 5. Exploiting time invariance of the ODEs, the ODE solver later uses the interval valuation assigned to this duration variable instead of the *time* variable's valuation. This is particularly useful when the same dynamics can occur in different steps of a trace, since learning becomes stronger when the absolute time is not a condition of the learned deduction. The *time* variable on the other hand is used to aggregate the durations of transition steps and should therefore be incremented by the duration of flows and by zero in case of jumps. This constraint needs to be supplied explicitly by the modeler in the transition system. In most models also the constraint *time = 0* should typically be given in the initial predicate.

The actual ODE constraints occur only in the transition part since they describe the connection between the states before and after a (continuous) transition step. An important restriction following directly from Definition 8 on SAT modulo ODE formulae is that all variables occurring in the right-hand-side term of an ODE constraint must themselves be defined by an ODE constraint. Without this condition, it would be unclear what the evolution of a variable x shall be between its valuations at the beginning (given by x) and at the end (given by x') of the step. The flow invariants, describing admissible regions that must not be left by a trajectory during its continuous evolution, are distinguished from normal constraints by the *x(time) >= 5* syntax in contrast to *x >= 5* or *x' >= 5*, which are constraints that only need to hold on the actual primed or unprimed variable instances.

Most preprocessing on the syntax tree is inherited from iSAT and performs certain normalizations like term simplifications and detection of common subexpressions to later avoid them being introduced multiple times in the solver core. In iSAT-ODE, an additional step is necessary to extract the ODE constraints and flow invariants. When encountering these constraints while traversing the syntax tree, they are translated into the internal symbolic format of the ODE solver (again syntax trees built from linked objects representing constants, variables, and arithmetic operators) and stored there. For each ODE constraint or flow invariant, a new discrete variable over $\{0, 1\}$ is added to the set of declared variables with a unique name, stored at the former position of the constraint in the syntax graph, and the correspondence between this new trigger variable and the constraint is stored in the ODE solver's valuation object. The trigger works as can be expected: having a value of 1, it signals that the constraint it

represents shall be *active*, i.e. must be satisfied by the valuation (and hence can be used for pruning); having a value of 0, the corresponding constraint shall be considered *inactive*, i.e. it does not matter whether the current valuation satisfies it and therefore it cannot be used for pruning. Importantly, there is no way to say that the constraint shall be *not* satisfied by the valuation—saving us from the semantic trouble of thinking about the negation of an ODE constraint. Consequently, however, we must require that ODE constraints only occur under an even number of negations, which can easily be checked by keeping track of this number during the traversal of the syntax tree.

Under this rewriting, the activation of ODE constraints and flow invariants cannot change during a step. This is compatible with the understanding that flows are governed by a specific system of ODEs, however, it introduces some subtle semantics for disjunctions of flow invariants: whenever a different set of disjuncts shall become active, the valuation of the trigger variables must change and hence there must be a new step. Consider e.g. *(x(time) >= 4 or x(time) <= 5)*, which would in principle allow flows for x to take arbitrary values. Due to the encoding and unwinding, however, the flow must be interrupted somewhere between 4 and 5 such that the respectively other flow invariant constraint can be activated.

Consistency Constraints. The condition that ODE systems must be definitionally closed is so important that the ODE solver adds *consistency constraints* to the formula which enforce that on each BMC depth either no ODE constraint is active or that for each ODE-defined variable there is exactly one ODE constraint active on whose left-hand-side the variable occurs. Assuming there are ODE-defined variables x and y and there are two ODE constraints defining x, normally depending on a mode, which may be encoded by the following constraints

flow and *m1* −> (d.*x* / d.time = *y);*
flow and *m2* −> (d.*x* / d.time = −*y);*

inside the transition system. Then these are replaced by trigger variables, e.g. *ode_trigger_0* and *ode_trigger_1*. Assuming there is one ODE defining y, i.e. its evolution shall be the same in both modes, and the corresponding trigger is *ode_trigger_2*, then the ODE solver introduces variables *dimension_trigger_0* and *dimension_trigger_1*. By adding

dimension_trigger_0 <= 0 or *(ode_trigger_0 + ode_trigger_1 >= 1*
and ode_trigger_0 + ode_trigger_1 <= 1);
dimension_trigger_1 <= 0 or *(ode_trigger_2 >= 1*
and ode_trigger_2 <= 1);

to the transition system, it enforces that when a dimension trigger is active there must be exactly one active ODE constraint for that dimension. Furthermore, adding a variable *odes_active* and the constraints

odes_active >= 1 or *ode_trigger_0 + ode_trigger_1 + ode_trigger_2 <= 0;*
odes_active <= 0 or *dimension_trigger_0 >= 1;*
odes_active <= 0 or *dimension_trigger_1 >= 1;*

enforces that when there is one or more ODE triggers active (hence the sum of them not 0), the *odes_active* trigger is active as well and then all ODE-defined

variables must have active dimension triggers—leading to exactly one ODE constraint being activated per dimension. It is important to note that very little attempt is made to detect compatible ODE constraints that could be active simultaneously for the same variable since they always have the same value. Only if the detection of common subexpressions incidentally detects syntactic equivalence, can this case be handled. In general, it must be avoided by the modeler to enforce two (even non-conflicting) ODE constraints for the same dimension to be required to be active at the same time. We have considered this case pathological and therefore not invested much effort into handling it, but a more robust implementation should detect this kind of modeling error and report it.

BMC Unwinding. After this preprocessing step, the ODE solver holds syntactic representations of the ODE constraints and flow invariants and knows the relevant symbolic variable names. Further preprocessing is done in the iSAT front end, leading to a conversion of the formula into a conjunction of clauses, which are disjunctions of atoms. This CNF-like representation still consists of initial, transition, and target predicates. It is the solver's BMC layer's task to then instantiate the variables $k + 1$ times in the solver core, store a mapping from the symbolic names and unwinding depths to these instances, and finally perform the k-fold unwinding of the transition system and addition of the initial predicate to the zeroth depth instances and of the target predicate to those of the final depth. With all ODE constraints and flow invariants being represented by trigger variables, only these are instantiated, while the actual constraints remain in the ODE solver without any duplication. The valuation object of the ODE solver is triggered to retrieve the relevant parts of the mapping from the BMC layer and store itself a correspondence of symbolic variable names and BMC depths to the instances in the solver core. This is done for all ODE-defined variables, the duration variable, and the trigger variables.

With this knowledge, the ODE solver is aware of the BMC structure of the formula and can be asked to provide a deduction for a specific unwinding by retrieving the corresponding valuation for the instances of that depth and its successor. For arithmetic constraints in the solver core, which are instantiated many times, deductions are made individually for each instance. They are not learned as general results applicable also to the other instances. This is easily justified by the relative computational cheapness of obtaining these deductions—when they are needed, they can simply be computed. After the explanations of the previous section, not much speculation is needed, however, to assume that ODE deductions are much more expensive than those for arithmetic constraints. Making the ODE solver BMC-aware allows iSAT-ODE to learn a deduction not only for one particular instantiation, but immediately replicate it for all BMC depths, just like conflict clauses are replicated when a *constraints replication* scheme [Sht00] is used. This will become clearer, once we will have discussed the learning of deduction results, but essentially, it amounts to learning the same clauses multiple times for different instances of the variables.

A typical way of using bounded model checking is to start with an unwinding depth of zero and increase the number of unwindings until the formula can no longer be shown to be unsatisfiable. This *iterative mode*, which checks unwinding depths consecutively while keeping those learned conflict clauses which are

still valid, has the advantages that it finds the shortest candidate solution trace (having first proven that none exists for a lower number of unwindings than the current) and that it starts with the smallest problem instance, which can be hoped for to also be the easiest to solve. However, this assumption need not always hold, especially, when low unwinding depths lead to trivially unsatisfiable formulae and very large depths lead to ones that have so many solutions that the solver can easily pick one, while for some intermediate unwinding depth the solver has to search extensively to disprove satisfiability of a formula that is nearly satisfiable.

When the BMC depth is thus increased, the ODE layer's valuation object is updated with the newly introduced variable instances and it introduces all learned clauses also for this new depth—again, an approach imitating the behavior often applied on learned conflict clauses in SAT and SMT solvers for BMC.

Input formula:

```
DECL
    define MAX_GLOBAL_TIME = 10;
    define MAX_DURATION = 1;
    float [0, MAX_GLOBAL_TIME] time;
    float [0, MAX_DURATION] delta_time;
    float [−10, 10] x;
    float [−10, 10] y;
INIT
    time = 0;
    x >= 0.9;
    x <= 1.1;
    y >= −0.1;
    y <= 0.1;
TRANS
    time' = time + delta_time;
    (d.x / d.time = −y);
    (d.y / d.time = x);
TARGET
    x = −1;
```

Stored ODE constraints and their activation at first ODE deduction:

```
Stored ODEs:
 [0]     [ active ]     (d.ode_var_1['x'] / d.time = (ode_var_2['y'] * −1))
 [1]     [ active ]     (d.ode_var_2['y'] / d.time = ode_var_1['x'])
Stored flow invariants.
 (none)
```

Valuation object at first ODE deduction for $k = 1$:

```
BMC depth 0
    Variables
     0     VAR0_time_0              [0,0] (point interval)
     1     VAR1_x_0                 [0.89999...,1.10000...] (width: 0.20000...)
     2     VAR6_y_0                 [−0.10000...,0.10000...] (width: 0.20000...)
    Trigger variables (ODEs)
     0     VAR17_ode_trigger_0_0    [1,1] (point interval)
     1     VAR18_ode_trigger_1_0    [1,1] (point interval)
    Trigger variables (flow invariants)
    Delta time variable
     VAR14_delta_time_0            [0,1] (width: 1)
BMC depth 1
    Variables
     0     VAR13_time_1             [0,1] (width: 1)
     1     VAR29_x_1                [−1,−1] (point interval)
     2     VAR31_y_1                [−10,10] (width: 20)
    Trigger variables (ODEs)
    Trigger variables (flow invariants)
    Delta time variable
current BMC depth: 0
```

Figure 3.19: Top: harmonic oscillator as iSAT-ODE input, middle: state of ODE solver at beginning of first ODE deduction, bottom: state of valuation object.

Example 10 (ODE solver state and valuation)**.** Figure 3.19 leads us from this description of the preprocessing steps right into the deduction for ODE

constraints in the next subsection. It shows an iSAT-ODE encoding of the harmonic oscillator example and (with slight modifications for better readability) the state reported by the ODE solver when it is asked to provide its first deduction. From the formula it is obvious that for $k = 0$, the solver cannot find a solution since the initial predicate requires $x \in [0.9, 1.1]$ while the target predicate, which for $k = 0$ needs to be applied to the same instance of x, requires $x = -1$. The solver hence progresses to the next unwinding depth by removing the target from the zeroth variable instance, instantiating the variables once more, adding the first unwinding of the transition system over the zeroth and first variable instances, adding the target to the first variable instance, and finally again performing unit propagation and ICP. Since in this simple example both ODE constraints are part of the top-level conjunction of the transition predicate, it is clear that no solution can exist in which these triggers are inactive (for any depth). The trigger variables hence receive the point intervals $[1, 1]$ as shown in the valuation object in the middle of the figure. From these valuations of the triggers stems the "active" marker, which is set for both stored ODE constraints in the ODE solver state.

The valuation also contains the intervals of the other relevant variables. To support readability, we let the solver keep track of the symbolic variable names and unwinding depths, causing internal variables to get useful names like *VAR1_x_0* for the zeroth instance of variable x. As can be seen in the valuation of this variable, the constraints from the initial predicate have allowed pruning from the original domain $[-10, 10]$ to the tighter interval $[0.9, 1.1]$—with outward rounding since the borders cannot be represented exactly by floating-point numbers. For the illustration, we have removed some post-decimal digits. Also variable *VAR6_y_0* has received the range admissible under the initial predicate as current valuation. The target predicate has restricted *VAR29_x_1* to the point interval $[1, 1]$, while no tightening of *VAR31_y_1* has occurred. Similarly, the solver knows very little about the duration of the step, denoted by *VAR14_delta_time_0* having a valuation $[0, 1]$, which directly follows from its declaration. This valuation, consisting of a box for BMC depth 0 and another for depth 1 forms the input of the first ODE deduction.

3.3.3 Deduction for ODE Constraints

When unit propagation and ICP have reached a fixed point in the iSAT core, the ODE solver is asked to provide a deduction. Having added the consistency constraints to the formula, we know that for each BMC depth there either is no active ODE at all or that for each ODE-defined variable there is exactly one active ODE constraint. Unless an unreasonably large minimum splitting width is given, ICP and unit propagation guarantee that these discrete constraints are satisfied. Nonetheless, it can easily be checked whether this condition holds before a deduction attempt is made.

Selection of BMC depth. There are two major options on how to answer the core's request for a deduction. First, deductions can be computed for all BMC depths with activated ODEs and all these deductions then be returned together for learning. Second, the ODE solver can choose a subset of BMC depths, perform deductions only for these, return them for learning and further

arithmetic deductions to the core, and to a later query answer with deductions for those BMC depths that have not been handled at this time.

Deduction on all active depths at once has the advantage that the core learns about all possible conflicts as early as possible. However, it has the disadvantage that the deduction computed for one depth is not available during the deduction of another. If ODE deduction has pruned bounds on one depth, it is likely that this also tightens the bounds on neighboring depths and, often separated by a jump-transition, thereby also the valuations that are relevant for other ODE deductions. In this mode, however, the ODE solver regards the given valuation as static, computes results (even for very coarse valuations) individually, only to be confronted after another round of core propagations with tighter bounds. On the other hand, analyzing only one BMC depth at a time and potentially performing deductions for this depth repeatedly when the valuation changes, may hide a conflict arising from the ODE constraints of another BMC depth and hence cause significant work that would be avoided if all depths were handled simultaneously.

In iSAT-ODE we have implemented both options and allow the user to choose between them via a command-line parameter. The second option is guided by the relative width of the valuation, sorting BMC depths by how tight their valuation already is and trying to compute from tightest to widest while returning results for propagation in the solver core whenever a new deduction could be made. As shown in the abstract overview in Figure 3.18, as long as the ODE solver provides a new result, the iSAT core will always return to its own deduction loop and thereafter ask again for further ODE deductions before a decision is made (or potentially a candidate solution box is reported). The ODE solver can therefore always safely start with the deduction from the BMC depth with the tightest valuation, iterate, and return after having produced a result, knowing that it will get another chance to produce deductions for the remaining BMC depths. This approach leads to a scheme that first tries to find a fixed point of unit propagation, ICP, and a subset of all possible ODE deductions and subsequently extends this subset by adding more BMC depths to the deduction. The motivation to try this lies in the observation that in some models there are conditions leading to very tight valuations e.g. of the initial instance, while little is known for instances that are far away from this depth. Concentrating on the propagation of the evolution that starts in this tightly-defined depth avoids computations of enclosures for the wider valuations on the other depths, which might provide only very little to solve the problem.

Pruning of Duration, Prebox, and Postbox. When we established pruning rules for arithmetic constraints earlier in this chapter by introducing ICP, we pointed out that pruning must only remove definite non-solutions from the interval valuations of the involved variables. Deduction must ensure that, if the box currently under investigation contains a satisfying point valuation in the sense of Definition 9, the solver will not dismiss it erroneously as a non-solution. The same guarantee must now be given by deductions for definitionally-closed systems of ODE constraints and activated flow invariants.

Taking the ODE solver's valuation object and a selected BMC depth, a *forward propagation* uses the valuation for the ODE-defined variables of that depth as *prebox*. These form the set of initial values. It uses the valuation of

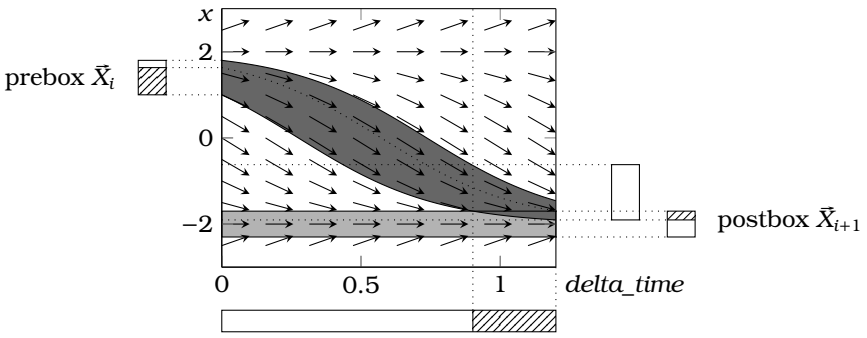


Figure 3.20: Deduction for the ODE constraint $\dot{x} = (x-2)(x+2)$ with prebox $x \in [1, 1.8]$, duration *delta_time* $\in [0, 1.2]$, and postbox $x' \in [-2.3, -1, 7]$. Possible deductions are indicated by hatched areas outside the graph. Within the graph, the dark-gray tube starting from the prebox is an exact enclosure of all possible trajectories emerging from there over the given temporal horizon. The light-gray area at the bottom illustrates the postbox over the entire range of possible durations. Deductions result from the intersection and its pre-image on the prebox.

the *delta_time* variable of that depth as duration, whose upper bound marks the *temporal horizon* up to which the enclosure must be computed. Finally, it uses the valuation of the ODE-defined variables of the successor depth as *postbox*, i.e. the set of states in which any "interesting" trajectory may end. Pruning can now remove those parts of the postbox which cannot be reached by any trajectory emerging from a point in the prebox for any length within the *delta_time* valuation. Similarly, it can prune the duration for which it can show that none of the trajectories emerging from the prebox can reach any point in the postbox. *Backward propagation* analogously may prune those points from the prebox which do not lead to any point in the postbox within the given duration.

Example 11 (ODE deduction)**.** In Figure 3.20, we illustrate ODE deductions and their effect on prebox, postbox, and duration using a simple example. Since the ODE system $\dot{x} = (x-2)(x+2)$ consists of only one dimension, so do the prebox and the postbox. The original prebox is given by $\vec{X}_i = ([1, 1.8])$. During a forward propagation, all trajectories emerging from this box need to be enclosed over the interval given for the *delta_time* variable, which is $[0, 1.2]$ in this case. The goals of a forward propagation are to prune the postbox and the duration. Therefore, the intersection between the enclosure and the postbox needs to be identified. This leads to a new lower bound of the *delta_time* interval valuation, *delta_time* ≥ 0.9018. The solver has thus successfully detected that no trajectory from the prebox can reach the postbox before this time. Just like with arithmetic deductions, pruning this part from the interval valuation removes only non-solutions. The intersection with the postbox also reveals that the lower part of the postbox cannot be reached within the temporal horizon (and as a human observer one could add that the part below $x = -2$ could not be reached for any larger temporal horizon either). With the given temporal horizon delta_time ≤ 1.2, this new lower bound is $x' \geq -1.9038$. The figure also illustrates clearly that the trajectories emerging from the upper part of the given

prebox will not reach the postbox before the horizon. Backward propagation is therefore allowed to deduce $x \leq 1.6319$, since points from above this threshold cannot be connected with points from the postbox via any solution trajectory of admissible length and hence these points constitute removable non-solutions.

Definition 11 (interval extension for ODE constraints and flow invariants)**.** Formalizing the intuition that can be gathered from our description of ODE deductions and its illustration in Figure 3.20, we define deductions for ODE constraints and flow invariants, based on our previous publications of this formalization in [EFH08, ERNF12a].

Given are an n-dimensional definitionally-closed, sufficiently-smooth, time-invariant ODE system $\dot{\vec{x}} = \vec{f}(\vec{x})$ and a (conjunctive) set of flow invariants (of the introduced form $x \sim c$ over variables x which are dimensions of $\vec{x}$, $\sim \in \{\leq, \geq\}$, and right-hand-side constants $c \in \mathbb{Q}$). Furthermore, given are interval valuations $\vec{X}_{pre}$ and $\vec{X}_{post}$ (in practice obtained from the valuations of the two successive BMC unwinding depths used within the current deduction) and a box $\vec{X}_{flow}$ as the subset of $\mathbb{IR}^n$ which is consistent with the flow invariants. Lastly, there is an interval $\Delta = [0, h]$ denoting the interval of possible durations. The combination of these elements may suitably be called an *ODE problem* $P = (\vec{f}, \vec{X}_{pre}, \vec{X}_{flow}, \Delta, \vec{X}_{post})$. We introduce under slight abuse of notational conventions

$$
\begin{aligned}
\vec{R}_\Delta = \{\vec{x}_{reach} \in \mathbb{R}^n \mid & && \text{a set of points, for which} \\
& \exists \vec{x} \in C^{\geq 1} : && \text{there exists a continuous function} \\
& \quad \vec{x}(0) \in \vec{X}_{pre} && \text{which starts in the prebox,} \\
& \wedge \exists \delta \in \Delta : && \text{which has admissible duration,} \\
& \quad \vec{x}(\delta) = \vec{x}_{reach} && \text{which reaches this point, and} \\
& \wedge \forall \tau \in [0, \delta] : && \text{which, for all time before,} \\
& \quad \vec{x}(\tau) \in \vec{X}_{flow} && \text{satisfies the flow invariants} \\
& \wedge \dot{\vec{x}}(\tau) = \vec{f}(\vec{x}(\tau)) && \text{and is an ODE solution,} \\
& \} &&
\end{aligned}
$$

and call $\vec{R}_\Delta$ the set of forward reachable states for which we call $\vec{E}_\Delta \supseteq \vec{R}_\Delta$ an enclosure of this set of forward reachable states. Similarly, we introduce

$$
\begin{aligned}
\vec{R}_0 = \{\vec{x}_0 \in \mathbb{R}^n \mid & && \text{a set of points, for which} \\
& \exists \vec{x} \in C^{\geq 1} : && \text{there exists a continuous function} \\
& \exists \delta \in \Delta : && \text{which has admissible duration} \\
& \quad \vec{x}(\delta) \in \vec{X}_{post} && \text{and ends in the postbox,} \\
& \wedge \vec{x}(0) = \vec{x}_0 && \text{which starts in this point, and} \\
& \wedge \forall \tau \in [0, \delta] : && \text{which, for all time before,} \\
& \quad \vec{x}(\tau) \in \vec{X}_{flow} && \text{satisfies the flow invariants} \\
& \wedge \dot{\vec{x}}(\tau) = \vec{f}(\vec{x}(\tau)) && \text{and is an ODE solution,} \\
& \} &&
\end{aligned}
$$

and call $\vec{R}_0$ the set of backward reachable states, calling $\vec{E}_0 \supseteq \vec{R}_0$ an enclosure of this set of backward reachable states. Finally, we call

$$
\begin{aligned}
\Lambda = \{\delta \in \Delta \mid & && \text{a set of durations, for which}
\end{aligned}
$$

$$
\begin{array}{rl}
\exists \vec{x} \in C^{\geq 1} : & \text{there exists a continuous function} \\
\vec{x}(0) \in \vec{X}_{pre} & \text{which starts in the prebox,} \\
\wedge\ \vec{x}(\delta) \in \vec{X}_{post} & \text{which ends in the postbox,} \\
\wedge\ \forall \tau \in [0, \delta] : & \text{which, for all time before,} \\
\vec{x}(\tau) \in \vec{X}_{flow} & \text{satisfies the flow invariants} \\
\wedge\ \dot{\vec{x}}(\tau) = \vec{f}(\vec{x}(\tau)) & \text{and which is an ODE solution,} \\
\} &
\end{array}
$$

the set of relevant trajectory lengths and $E_\Lambda \supseteq \Lambda$ an enclosure of these lengths.

Deduction for ODE constraints and flow invariants therefore amounts to computing these enclosures and subsequently the intersections $\vec{X}'_{pre} = \vec{X}_{pre} \cap \vec{E}_0$, $\vec{X}'_{post} = \vec{X}_{post} \cap \vec{E}_\Delta$, and $\Delta' = \Delta \cap E_\Lambda$.

3.3.4 Computation of Enclosures and Refinements

Bit by bit, we can now focus on how to actually perform the computations necessary for an ODE deduction—starting with the rather technical, yet fundamental question of how to use VNODE-LP when the ODE system is known only after having parsed and preprocessed the input file.

Run-Time Definition of ODE Constraints. In Figure 3.16, we showed the implementation of the right-hand side of an ODE as a C++ function that is passed to the automatic differentiation routines, which are used by VNODE-LP for the evaluation of the function and its derivatives. While this works well when the ODE system is known at compile time, it is unsuitable in our scenario, in which the ODE constraints are extracted from a parsed input formula at run time.

One could employ dynamic code generation and recompilation to generate functions on the fly, albeit at the cost of having to deal with several technical issues and potential errors. In our approach, we avoid these problems altogether by making the template function fully generic and dependent on the syntax tree of the ODE system. We recall the function signature of function *f* from Figure 3.16

```
template <typename var_type>
void f(int num_dims, var_type* fx,
       const var_type* x, var_type t, void* param),
```

which also needs to be the function signature used within our ODE layer. The central idea is to use the *param* argument to pass the ODE constraints as syntax trees and a mapping of ODE variable indices to their dimension within the ODE system and vectors *fx* and *x*. This mapping is needed since a variable in the input formula may have become e.g. the fifth variable in the ODE solver, *ode_var_5*, while it is not necessarily also the fifth dimension of the ODE system, e.g. simply due to a different ordering or due to an optimization which handles independent subsystems separately and therefore uses fewer dimensions than variable indices. Instead of evaluating a hard-coded expression for each dimension of the ODE system, the generic evaluation function iterates over the dimensions and for each expression tree it calls a second template

function. This second function performs the actual evaluation recursively by descending into the given syntax graph and using the supplied variable mapping. Its returned results are then stored in the respective dimensions of the (output) *fx* argument. Internally, it uses a case distinction to handle the different node types from the expression tree and recursively calls itself on the child nodes, returning e.g. their sum or product. These operations are transparently replaced by the overloaded automatic differentiation operations once the function template is instantiated. When a variable node—as a leaf of the expression tree—is reached, the corresponding dimension of the (input) *x* argument is returned. Reaching a constant in a leaf node, the function simply returns its interval representation.

Surprisingly, the overhead of this generic computation over a hard-coded function is not as large as it may seem, since FADBAD++ builds up an internal representation and therefore needs to use the actual evaluation function only during this initialization—which Ned Nedialkov pointed out during a discussion of this approach.

Enclosure Generation and Data Structures. Our application of VNODE-LP in Example 9 has illustrated how an enclosure over the entire duration interval from zero to the temporal horizon can be obtained from the sequence of a-priori enclosures. In principle, we could just intersect these a-priori enclosures with the postbox, discard the corresponding part from the *delta_time* valuation if that intersection is empty or discard those parts of the postbox, which are not in the intersection with any of the a-priori enclosures over the relevant time frame. In fact, this approach is a natural first candidate for a forward propagation algorithm since it requires little more than calling the VNODE-LP methods shown in the example and computing some intersections. However, we have already pointed out that the a-priori enclosures are quite coarse overapproximations that may contain a lot of spurious points, which could and should be excluded in order to provide a more precise pruning result.

For a tighter deduction result—as shown in Figure 3.20—we need to complement the computation of a coarse enclosure with refinements that reduce the amount of overapproximation while still keeping the guarantee of not losing any solutions. During the development of iSAT-ODE, we have explored different refinement schemes, starting with relying purely on the a-priori enclosures and finally arriving at a tight integration also of the bracketing system computation, which will be discussed in Section 3.5. Until then, we leave out the aspects referring to bracketing and concentrate on the enclosure computation based directly on VNODE-LP.

To support the generation of iterative refinements, the ODE solver within iSAT-ODE introduces a *solver_run* class that encapsulates the (either forward or backward) deduction computation for an ODE problem, i.e. a system of ODE constraints, flow invariants, pre- and postboxes, and the *delta_time* interval indicating the range of possible durations. A *solver_run* instance then offers the different phases of enclosure computation as methods. For now, they can be seen as a sequence: initialization, computation of an enclosure up to the given temporal horizon, refinement of enclosures to achieve different deduction goals, and generation of a deduction to be learned by the solver core. As can easily be guessed from the shown examples, during this process, a sequence of enclosures is computed, which needs to be stored for refinement and for the

extraction of relevant information that can be assembled into a deduction.

Central to the *solver_run* class is therefore a list of objects from an *enclosure* class. Primarily, an *enclosure* encapsulates a subrange of the duration and a box which encloses all trajectories over this interval. A temporally-ordered sequence of these objects—without any gaps in the subranges of the duration they cover—therefore represents an enclosure of the solution trajectories over the entire interval of durations. The *enclosure* objects, however, also contain all information that is needed to compute an enclosure for a subrange of their specific interval of the duration. Early evolutions of iSAT-ODE only stored the time and tight enclosure that VNODE-LP computed for the beginning of the range covered by the *enclosure* object. As hinted earlier, this enclosure together with an interval for what was called the *t_end* variable in our example can be used to compute an enclosure for that interval—however, at the significant cost of local wrapping and very frequent costly reinitializations of VNODE-LP. In [ERNF12a], we circumvented the VNODE-LP interface to extract more information that allows a much cheaper reevaluation of the enclosure for a temporal subrange and at the same time avoids additional wrapping by restoring the transformation matrices prior to the evaluation. We therefore added a *TaylorCoefficients* class to VNODE-LP, encapsulating this internal state and methods for its accelerated reevaluation on a subrange of time, and extended the *enclosure* class to hold an object of this type.

Enclosure up to the Temporal Horizon. After initializing the *solver_run* by passing an ODE system via FADBAD++ to a *VNODE* solver object, the first task is to compute an enclosure up to the temporal horizon. Ignoring aspects relating to bracketing systems for the moment, the approach amounts mostly to calling the stepwise integration method that we also employed to generate an enclosure in Example 9. However, instead of printing the enclosure and continuing with the next integration step, the *solver_run* extracts all the information that is necessary to generate a new *enclosure* object and appends it to the list of enclosures.

If VNODE-LP is unable to perform an integration step, the horizon cannot be reached and the enclosure is hence incomplete and not suitable for a deduction, since the evolution after the currently-reached point of time remains unknown. In this case, the *solver_run* is considered failed and error handling must occur on the level of the ODE solver. Depending on a command-line switch, this may simply be ignored, hoping for splitting and further deductions to yield a smaller initial set or duration interval for which an enclosure can be computed. Ignoring such incomplete ODE deductions, however, comes at the price of an increased risk to produce spurious candidate solutions whenever the ODE problem cannot be handled even after further splitting.

Flow invariants provide a second reason to stop before the horizon is reached. If it can be proven that all trajectories have left the region of states admissible under the active flow invariant constraints, the enclosure may be completed by adding a special *enclosure* object that indicates the empty set. This object is obviously not an enclosure of all trajectories emerging from the prebox, but it correctly describes the set of all trajectories that emerge from the prebox while never having left the admissible region, which we denoted by $\bar{X}_{flow}$ in Definition 11.

The algorithm for computing an enclosure up to the horizon under flow invariant constraints (implemented as *enclose_flowinv_aware* in the *solver_run* class), without many of the technical details that are better left to the source code and without bracketing enclosures, can be described as follows.

1. Compute $\vec{X}_{flow}$ and intersect the prebox with it.
2. Perform one enclosure step with VNODE-LP and generate an *enclosure* object, holding the relevant internal representation. Refine the a-priori enclosure by reevaluating the stored representation over the entire computed step.
3. Check whether during this step all trajectories have left $\vec{X}_{flow}$ and therefore the enclosure can be terminated before the temporal horizon has been reached. If so, add a special empty enclosure to the end of the sequence of stored enclosures.
4. Iterate from Step 2 until the horizon is reached.

The first step in detecting whether the enclosure leaves $\vec{X}_{flow}$ is to intersect the enclosure computed for the current subrange of the duration with the complement of $\vec{X}_{flow}$. Thereby, the method checks whether this region outside the admissible states may have been reached by any trajectory at all. If so, there is a chance that during this step, not only some, but in fact all trajectories have left the flow invariant. The *solver_run* therefore attempts to find a witness for not having left $\vec{X}_{flow}$, i.e. a sequence of enclosure points that cannot be removed by further temporal refinement (but which might still be spurious—and might be found so if the prebox were tighter).

There are several ways to achieve the goal of finding a trajectory that stays inside the admissible region. The most precise method would pick a single point from the tight enclosure at *delta_time* = 0 and compute an enclosure for this starting point. As long as this trajectory stays inside $\vec{X}_{flow}$, it is a true witness, and the enclosure must be continued. However, when the trajectory reaches the region outside $\vec{X}_{flow}$, there might be an alternative starting point for which a longer admissible trajectory exists. Picking the starting point farthest away from the flow invariant's border would help, but having not one, but n dimensions, there still would be multiple choices. Using the partial derivatives would give valuable hints in this respect—and in fact, the bracketing approach can be seen to actually support this to some degree—but the sign of the partial derivatives may change over the evolution of a trajectory, which may make the globally right choice for an initial point quite complicated.

The implemented method uses a far simpler and more local argument and does not aim to be precise, except when claiming that the flow invariant has definitely been violated. Its central part is illustrated in Figure 3.21 and implemented as *try_to_prove_that_some_trajectories_stay_inside_fi*. Having already established that a part of the enclosure for the entire step lies outside the flow invariant (but not the entire enclosure), the algorithm works essentially as follows.

1. Calling the current step's temporal interval $T = [t_j, t_{j+1}]$ and the enclosure for it $\vec{X}(T)$, compute the intersection $\vec{X}(T) \cap \vec{X}_{flow}$ and evaluate the ODE's right-hand side over this intersection. The resulting interval vector

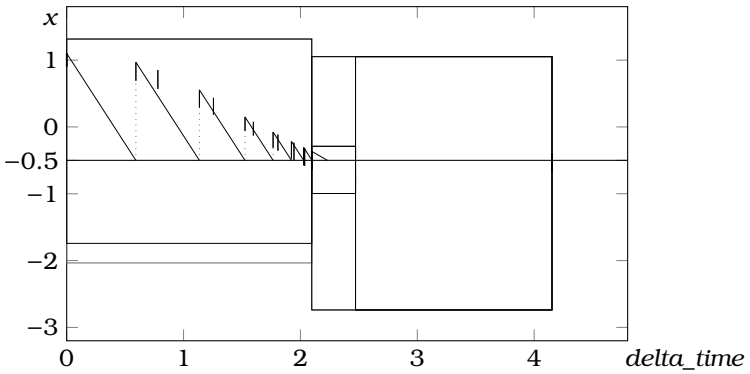


Figure 3.21: Computation of an enclosure up to the horizon for the ODE system $\dot{x} = -y$, $\dot{y} = x$ with $x_0 \in [0.9, 1.1]$, $y_0 \in [-0.1, 0.1]$. Enclosure leaves region admissible under flow invariant $x(t) \geq -0.5$, causing termination of enclosure computation before horizon. Additionally plotted are the linear evolutions from the point farthest away from the flow invariant border, the peeking enclosures for accelerated termination, and the splitting result of the final enclosure.

contains the most extreme slope that any trajectory starting from the enclosure at the beginning of this step, $\vec{X}(t_j)$, and staying inside the flow invariant may take over T. Evaluating over the intersection is in fact a heuristic choice, since we have not yet established the existence of a trajectory that actually stays inside the flow invariant. This choice has practical advantages since it decreases the range of possible slopes and is optimistic about finding a trajectory, but it does not guarantee to find the first point when the enclosure is left since it omits some of the slopes associated with trajectories that do not stay inside $\vec{X}_{flow}$.

2. For each dimension and direction (up or down), determine the largest distance between the points from the intersection of $\vec{X}_{flow}$ and the enclosure $\vec{X}_t$ at the current point of time t (initially $t = t_j$) and the lower / upper border of $\vec{X}_{flow}$. If there is no flow invariant constraint in this direction, e.g. no upper bound, or the slope is strictly negative (for upper bounds) or strictly positive (for lower bounds) this distance is infinite and the flow invariant cannot be violated by any trajectory in this dimension and direction. Otherwise, there is a finite distance and the quotient of distance and slope yields the earliest point time t', at which this flow invariant could be violated under the discussed restrictions.

3. Take the minimum point of time from the previous step as a new t. If $t \geq t_{j+1}$, the entire step T has been covered and this sequence is considered a witness (even though it may be a spurious one due to the locality of the argument and the wrapping and overapproximation entailed). Not having reached the end of this step, a point of time $t_p > t$ is chosen a little ahead, but inside T, and $\vec{X}(t_p)$, a *peeking enclosure*, is computed. If this enclosure lies entirely outside the flow invariant, report t_p as a known new temporal horizon, since this enclosure has proven that $\vec{X}_{flow}$ is indeed left at or before t_p and hence the flow invariant constraint is violated for at

least a point of time by all trajectories. The step over T is split into two parts at t_p and the part coming after t_p is safely discarded.

4. If neither the peeking enclosure allowed termination of the enclosure nor the end of the step has been reached, compute $\bar{X}(t)$ and iterate from Step 2 until one of the conditions is met or the progress $t' - t$ becomes smaller than a given threshold. In that case, the method reports that it could not refine the already known status, that the enclosure lies partially outside and partially inside the flow invariant's admissible region, and therefore the enclosure computation needs to be continued.

Clearly, this method has some drawbacks of its own, chiefly among them the imprecision and the problem that the progress between substeps decreases once the border of the flow invariant is approached, which is well illustrated in Figure 3.21. Future work might improve this algorithm significantly by combining a locally chosen extreme point with a high-order enclosure, which would allow large substeps while avoiding the problems of identifying an initial point that is followed from zero to the temporal horizon. Also, when available, the bracketing enclosures might sometimes be directly useable as witnesses when one of them stays inside the flow invariant for the entire time.

We should note here that our approach uses flow invariants solely for terminating the enclosure when all trajectories are known to have left $\bar{X}_{flow}$, but not for pruning of intermediate enclosures. Recalling the semantics of flow invariants, a trajectory only satisfies an active flow invariant constraint when it does not violate the invariant during its entire evolution between its starting point in the prebox and its end point in the postbox. Consequently, we are allowed to remove the tail of a trajectory at the point where it crosses the border of the invariant and need not enclose this particular trajectory beyond this point. However, we must also consider that intersecting an intermediate enclosure with $\bar{X}_{flow}$ involves difficulties very similar to those usually encountered in reachable state computations for hybrid systems. If the enclosures are wrapped to perform the intersection with $\bar{X}_{flow}$, this wrapping is not only local, but has to be propagated throughout the subsequent integration steps. If on the other hand an overapproximation of $\bar{X}_{flow}$ with respect to the coordinate systems of the enclosure is computed such that an intersection can be performed in the changed coordinates, the intersection result will not be very precise either. Since in iSAT-ODE there is an outer loop that will provide a tighter set (due to further deduction and splitting), we have decided not to try this kind of pruning of the trajectories during their evolution, but instead to only try to detect when all trajectories have left $\bar{X}_{flow}$ as shown above. Nonetheless, pruning intermediate enclosures might in many cases lead to tighter enclosures and could therefore accelerate the search since spurious trajectories could be discarded by pruning rather than having to wait for a suitable refinement made by the iSAT core.

Conservative Bounds over Entire Duration. At the beginning of this chapter, while discussing how conflict analysis within the iSAT core aims to find more general reasons for a conflict, we pointed out that learning more general facts increases the amount of pruning that can be applied to the search tree and hence accelerates the search. To support this strive for generality, deduction rules for arithmetic constraints try to limit the number of reasons they record for a deduction in the implication graph.

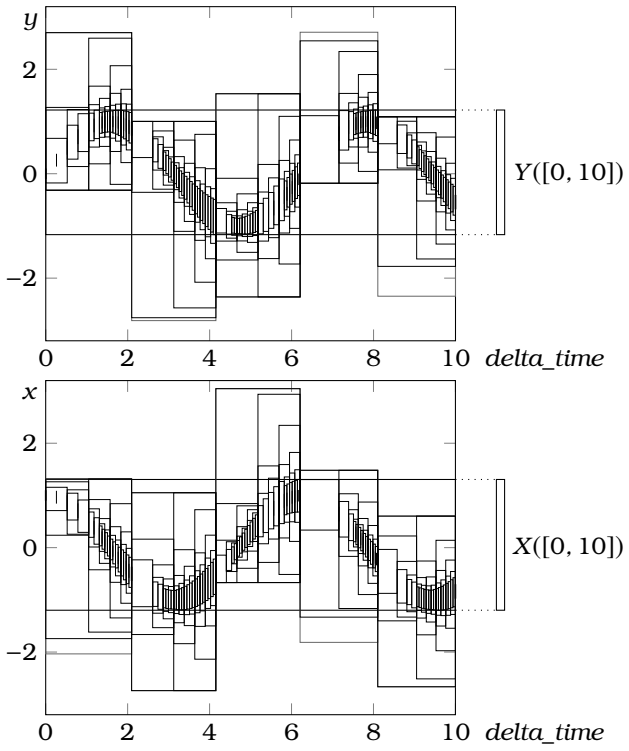


Figure 3.22: Computation of conservative bounds over entire duration for the harmonic oscillator example $\dot{x} = -y$, $\dot{y} = x$, with $x_0 \in [0.9, 1.1]$, $y_0 \in [-0.1, 0.1]$, and *delta_time* $\in [0, 10]$ with an unusually coarse resolution to reduce the number of boxes for illustration purposes. Each dimension requires boxes over different time spans to be refined, leading to seemingly superfluous refinements when looking at one graph in isolation.

Although our approach for ODE deductions does not allow us to detect when e.g. only a lower bound of a variable is truly relevant, we have the freedom to perform a deduction that is independent of the postbox and hence only depends on the prebox and the duration interval. In *get_safe_tight_enclosure_over_toi*, with *TOI* standing for *time of interest*, which in this case is synonymous with the valuation of *delta_time*, the *solver_run* computes such an enclosure by iteratively refining the sequence of enclosures via reevaluation of the stored representations over temporal subranges. This algorithm can be switched off by a parameter, but if left in place, it performs the following steps.

1. Select all elements from the list of *enclosure* objects that cover the given time of interest.

2. Since the outer elements may cover time outside the relevant time frame, split them temporally and reevaluate them for the refined subranges. Remove the (new) outer parts such that there is little to no coverage of any time outside the given time of interest.

3. Pick one dimension and decide to either refine its upper or lower bound.

This could be done by sophisticated heuristics, but our implementation simply iterates over all dimensions.

4. Sort the *enclosure* objects covering the time of interest by their bound in this dimension and direction, such that the boxes are ordered by their influence on the convex hull (i.e. the first box after ordering is most extreme and therefore defines the value that would be reported as a bound in this dimension and direction over the given time of interest).

5. If reasonable, refine the first element of the sorted list by splitting its temporal width and reevaluating the enclosure over the subranges. To get an estimate for the possible progress that can be expected from refinement, first compute an enclosure for the temporal midpoint of this enclosure and compare it with the bound for the entire subrange. Only if the progress exceeds a given spatial resolution threshold, perform the actual refinement. Also do not refine, if the computed bound for the temporal midpoint lies outside $\bar{X}_{flow}$, since in that case further refinement can only lead to a deduction that is still weaker than what follows directly from the flow invariant constraint. If refinement is done, insert the resulting elements in the sorted list (keeping it sorted by their relevance for the convex hull) and replace the to-be-refined element in the sequence of *enclosure* objects by the refined ones (keeping it temporally sorted and without gaps).

6. Continue refining the first element of the sorted list until the temporal width no longer exceeds a given minimum resolution threshold. Also stop the refinement if the element has a very tight spatial width in this dimension, which often occurs for dimensions with constant evolution, which will not become tighter under further refinement. The latter condition is somewhat redundant since the progress computation will quickly show that no pruning can be expected. The thresholds are specified by an *enclosure_resolution* object that is given as a parameter.

7. Continue with the next dimension and direction pair in Step 3.

The refinement and the resulting bounds, which are easily obtained by computing the union of the refined boxes, are illustrated in Figure 3.22.

Refutation of the Reachability of the Postbox. The most essential task of an ODE deduction is to remove parts from the valuation that do not satisfy the ODE problem as a whole, i.e. including the question of which parts of the postbox are reachable and for which durations. It makes, however, sense to first check whether the postbox is reachable at all. In the *try_to_refute_postbox* method, the *solver_run* searches for an enclosure that cannot be refined any further under the chosen parameters and has a non-empty intersection with the postbox for some duration from the *delta_time* valuation. If, on the other hand, all enclosures have empty intersections with the postbox, it can safely be considered unreachable and a conflict be learned. On finding a witness, the search can be aborted and instead an attempt be made to refine the postbox and set of durations.

Assuming the *solver_run* has computed a sequence of enclosures up to the temporal horizon, stored it, and has potentially done some refinements, but

kept the sequence ordered and without gaps, the algorithm for refuting the reachability of the postbox is as follows.

1. Iterate over the sequence of enclosures and skip those that cover a time before the *delta_time* valuation. Thereafter, skip all enclosures that have an empty intersection with the postbox.

2. If an enclosure within the *delta_time* range has a non-empty intersection with the postbox, this could either be caused by intervals that are too wide for a tight evaluation, or the enclosure could actually contain a non-refinable enclosure with non-empty intersection with the postbox. Therefore, reevaluate the stored VNODE-LP representation over the temporal midpoint of the current enclosure to get the tightest possible enclosure for that point under the current choice of parameters.

3. If this midpoint enclosure has a non-empty intersection with the postbox or the enclosure's temporal width does not exceed the given temporal resolution threshold, it is a witness. Otherwise, refine the current enclosure by splitting its temporal width in halves, reevaluating the stored terms, and replacing the stored enclosure with the newly computed ones.

4. Starting with the first of the newly added enclosures, skip again all elements of the list until one is found which has a non-empty intersection with the postbox and repeat this process from Step 2.

5. When reaching a time greater than the upper bound of the *delta_time* valuation without having found a witness, all boxes have been refined sufficiently to prove that they have an empty intersection with the postbox and hence that the postbox is unreachable from the prebox during the given *delta_time* valuation.

Tightening of Duration and Postbox. When a non-refinable enclosure is found during the attempt to refute the reachability of the postbox, the method *tighten_toi_and_postbox* is called. Its goal is to compute tighter bounds for the postbox and *delta_time*. The algorithm works on the sequence of enclosure objects that has been left behind by any preceding steps performed in the *solver_run*, so this sequence is known to reach the horizon and most likely some of the enclosures have already been refined, especially during the (failed) attempt to refute the reachability of the postbox and optionally during the computation of an enclosure for the entire duration independent of the postbox.

It is important to note that trajectories may reach the postbox, leave it, and later come back, i.e. there are not only multiple durations for which trajectories would end inside the postbox, but actually multiple *disjoint sets of durations*, for which the postbox is reached. Our discussion of *urgency* in Section 2.1 has already shed light on the use of flow invariants and their limitations to exclude such non-urgent solutions if necessary. As a preparation to be able to potentially learn these disjunctive subranges of the *delta_time* valuation for which the postbox may be reached, the *tighten_toi_and_postbox* method introduces a set of *times of interest (TOIs)*. In principle, these could be learned as individual enclosures for each subrange, some of which may be tighter than their convex hull. Since iSAT's deduction rules are not able to automatically propagate the

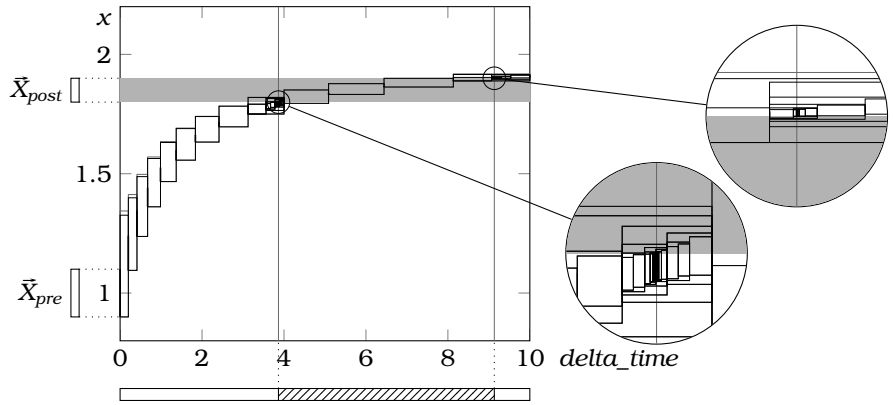


Figure 3.23: Refined enclosures after attempting to refute the reachability of the postbox and subsequent refinement of the postbox and *delta_time* valuations for the ODE $\dot{x} = (x-2)^2$, with $x_0 \in [0.9, 1.1]$, $x_1 \in [1.8, 1.9]$, and *delta_time* $\in [0, 10]$.

convex hull of the elements of a disjunction without first exploring the individual cases—e.g. they would not see that $x \geq 3$ if $(t \leq 3 \wedge x \geq 5) \vee (t > 3 \wedge x \geq 3)$ were supplied—we have decided not to learn these individual elements, but only the convex hull. It can either be considered an inconsistent design decision to first compute these individual enclosures and then to only learn their hull, or, as we should prefer to call it, a preparatory step for a possible extension which should be considered when the solver core makes more complete use of such disjunctions, sometimes called *disjunctive reasoning*.

First, a list of TOIs is initialized such that the *delta_time* interval is split into TOIs at those enclosures which do not reach the postbox.

1. Given the (time-ordered, gapless) sequence of enclosures, a *delta_time* valuation, and the postbox, start a fresh TOI, when the first enclosure is encountered in the sequence which covers some of *delta_time*'s valuation and has a non-empty intersection with the postbox.

2. Add all subsequent enclosures to this TOI if they have non-empty intersection with the postbox and *delta_time*.

3. End the current TOI when an enclosure with empty intersection with the postbox is encountered. Further enclosures with empty intersection are skipped, repeating from Step 1.

4. Stop, when reaching an enclosure that starts at a time whose lower bound is greater than *delta_time*'s upper bound, since this indicates that there can be no more TOIs beyond the current enclosure.

Subsequently, the enclosures over the identified TOIs are refined by an algorithm that is quite similar to one used by the *get_safe_tight_enclosure_over_toi* method. However, further refinement may uncover further enclosures which have an empty intersection with the postbox and hence may necessitate that the currently examined TOI be split further. Additionally, refinement must be performed from the beginning and from the end to get tight bounds also for the duration.

1. Iteratively pick the TOI and the sequence of enclosures covering it.
2. Iteratively pick a dimension and direction (up or down) for refinement and sort the enclosures over the currently to-be-refined TOI by their bound in this direction of this dimension.
3. If the most extreme enclosure has a temporal and spatial width above the respective thresholds, compute a midpoint enclosure. If this midpoint enclosure lies partially outside $\vec{X}_{flow}$, stop refinement for this direction since the flow invariant provides a stronger bound than can be expected from the enclosure. Compute a maximum possible progress by *not only* comparing the enclosure over the current subrange with the midpoint enclosure, *but also* comparing the postbox in this dimension with the midpoint enclosure. Only if this possible progress exceeds the spatial threshold, compute a refinement by performing reevaluations for temporal subranges and replacing the enclosure. This approach avoids refinement of enclosures whose midpoint enclosure is already known to cover the postbox.
4. If the enclosure has been refined, compute the intersection of the refined boxes with the postbox to check whether the TOI can be split. If the postbox is still partially covered by the resulting enclosures, repeat their refinement in Step 3 or continue with the next dimension of this same TOI in Step 2 when the resolution thresholds are no longer exceeded.
5. Otherwise, if a split of the TOI has occurred, enqueue both subranges that result from the split and repeat from Step 2 on the first element resulting from the split. If the current TOI has become obsolete, remove it from the list and continue with the next TOI in Step 1.
6. If the TOI has been refined successfully (without further splits), refine the duration for this TOI (which is implemented in *refine_toi_from_outside*). While the first element of the TOI's enclosures contains points that lie outside the postbox and has a temporal width which is larger than the given resolution, refine this element and remove from the TOI those resulting subranges whose enclosures have an empty intersection with the postbox. Analogously, refine the TOI's upper bound by refining the last element and removing subranges which have an empty intersection with the postbox.
7. Select the next TOI from the list in Step 1.

Figure 3.23 illustrates the refined boxes after application of this method (and the preceding steps) on an example in which the entire postbox is reachable, but the *delta_time* valuation can be tightened.

3.3.5 Backward Deduction

It has probably not gone unnoticed that we have been more than a bit vague about what needs to be done for backward deduction, i.e. for the pruning of the prebox via a computation of the set of initial points for which trajectories reach the postbox. The reason is that backward deduction can be performed by the methods we have presented so far, when only a few minor modifications are made to their input.

Reversing a Trajectory. Given are a sufficiently smooth time-invariant ODE $\dot{\vec{x}} = \vec{f}(\vec{x})$ and a solution function $\vec{x} : \mathbb{R} \to \mathbb{R}^n$, which connects two time instances 0 and $\delta \in \mathbb{R}_{\geq 0}$ and the corresponding points $\vec{x}(0) = \vec{x}_0$ and $\vec{x}(\delta) = \vec{x}_\delta$. Since $\vec{x}$ is a solution function, it has the right slope at all times and stays inside the box admissible by the flow invariants, i.e. $\forall t \in [0, \delta] : \frac{d\vec{x}}{dt}(t) = \vec{f}(\vec{x}(t)) \wedge \vec{x}(t) \in \vec{X}_{flow}$.

We are interested in finding a *reverse ODE system* $\dot{\vec{y}} = \vec{g}(\vec{y})$ with a solution function $\vec{y} : \mathbb{R} \to \mathbb{R}^n$. To be useful for backward propagation, this reverse solution function must do the "opposite" of the original solution function. Given the final point of the original solution, the reverse solution function must follow the original trajectory—only backwards—until it reaches the original trajectory's starting point, exactly after the same duration.

Considering a *backward time* function $\beta : \mathbb{R} \to \mathbb{R}$ with $\beta(t) = \delta - t$, we therefore want that $\vec{y}(\beta(t)) = \vec{x}(t)$, thus e.g. $\vec{y}(0) = \vec{x}(\delta)$. For the slope of $\vec{x}$, we know from the original ODE system that

$$\begin{aligned}
\frac{d\vec{x}}{dt}(t) &= \lim_{h \to 0} \frac{\vec{x}(t+h) - \vec{x}(t)}{h} && \text{(limit from the right side at } t\text{)} \\
&= \lim_{h \to 0} \frac{\vec{x}(t) - \vec{x}(t-h)}{h} && \text{(limit from the left side at } t\text{)} \\
&= \vec{f}(\vec{x}(t)).
\end{aligned}$$

The limits from the left and from the right must coincide since the derivative is known to exist. For the two points of time t and $t + h$, the backward times are $\beta(t) = \delta - t$ and $\beta(t+h) = \delta - (t+h)$. The corresponding values are $\vec{y}(\beta(t)) = \vec{x}(\delta - t)$ and $\vec{y}(\beta(t+h)) = \vec{x}(\delta - (t+h))$. Therefore

$$\begin{aligned}
\frac{d\vec{y}}{dt}(\beta(t)) &= \lim_{h \to 0} \frac{\vec{y}(\beta(t+h)) - \vec{y}(\beta(t))}{h} \\
&= \lim_{h \to 0} \frac{\vec{x}(\delta - (t+h)) - \vec{x}(\delta - t)}{h} \\
&= -\lim_{h \to 0} \frac{\vec{x}(\delta - t) - \vec{x}(\delta - (t+h))}{h} \\
&= -\lim_{h \to 0} \frac{\vec{x}(\delta - t) - \vec{x}(\delta - t - h)}{h} && \text{(limit from the left at } \delta - t\text{)} \\
&= -\lim_{h \to 0} \frac{\vec{x}(\delta - t + h) - \vec{x}(\delta - t)}{h} && \text{(limit from the right at } \delta - t\text{)} \\
&= -\vec{f}(\vec{x}(\delta - t)) \\
&= -\vec{f}(\vec{y}(\beta(t)))
\end{aligned}$$

gives us $\dot{\vec{y}}(\beta(t)) = -\vec{f}(\vec{y}(\beta(t)))$. Luckily, we have required $\vec{f}$ to be time-invariant, i.e. it does not depend on t (nor $\beta(t)$ in this case) directly. Therefore, we can write the reverse ODE system as $\dot{\vec{y}}(t) = -\vec{f}(\vec{y}(t))$.

A less formalistic argument is based on simple geometry. Picking a point on the forward trajectory, $\vec{f}$ yields the slope of the trajectory towards a larger point of time. Picking the same point and "looking" backwards in time, we therefore need to go with the negative of the forward trajectory's slope to reach this point on the trajectory for a smaller point of time. Whenever we go down with a slope by e.g. -1, to undo this, we need to go up with a slope of $+1$.

Flow invariants apply to the reverse trajectory in the exact same way as they apply to the forward trajectory. Since they do not vary with time, but simply

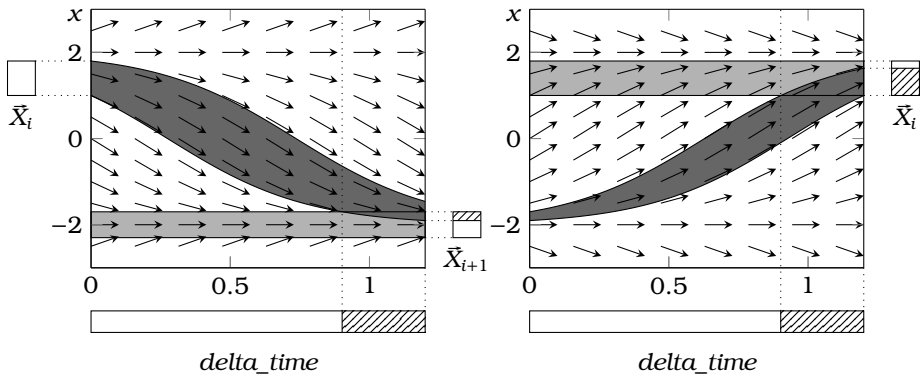


Figure 3.24: Forward and backward ODE deductions. Left: forward deduction as seen in Figure 3.20. Right: postbox from forward deduction taken as prebox for backward deduction with reverse ODE system $\dot{x} = -(x-2)(x+2)$ and original prebox taken as postbox over same *delta_time* interval. The backward deduction leads to the tightened prebox of the original problem.

constrain the space through which the solution function may pass, it does not matter when we reverse the time and start at δ instead of 0.

Lifting of Reverse Trajectories to Intervals. Time invariance is in fact very important when not only looking at individual trajectories, but at the backward propagation algorithm we are aiming at. In Figure 3.20, a backward deduction looked quite different from a forward deduction: instead of starting with an enclosure at one point of time ($t = 0$), the backward propagation needs to start with at box that has temporal and spatial extent, since it needs to consider all points starting in the postbox over time instances from the *delta_time* valuation. Obviously, this is not directly compatible with the kind of enclosure algorithm we have used so far, but fortunately, it is also not necessary.

Using the time invariance argument, we can consider the postbox as prebox at time $t = 0$ and compute the enclosures again over the *delta_time* interval. Time invariance guarantees that the absolute value of time does not matter as long as we do not change the duration of the trajectory. Therefore, instead of starting over an interval of time and ending at a point, we start again at a point of time $t = 0$, and enclose the reverse trajectories over all durations compatible with the *delta_time* valuation. Figure 3.24 illustrates this exchange of pre- and postbox for the reverse ODE problem with negated right-hand sides.

Computation of Backward Deductions. To actually compute backward deductions, the ODE solver first creates the reverse ODE system by simply manipulating the symbolically-represented ODE constraints, i.e. at the root of each right-hand-side syntax graph it inserts a multiplication node and constant -1 as factor. Knowing that it is doing a backward deduction, it exchanges the roles of pre- and postbox when creating the *solver_run*. The results are then known to require an additional swapping of these roles. Backward deduction is therefore already covered by the steps we have described throughout this subsection and requires only little attention in the remaining steps that follow now.

3.3.6 Utilizing Deduction Results in iSAT-ODE

After having discussed the details of propagation for ODE constraints, it is probably quite obvious that computing an ODE deduction is orders of magnitude more expensive than ICP for an arithmetic atom. In our earlier attempts to solve SAT modulo ODE problems [Egg06, EFH08] we tried to use ODE deductions like any other propagator, i.e. compute deductions as they are needed, store their reasons and consequences in the implication graph, and iterate until either no more progress can be achieved or a conflict is encountered. In such a naive integration scheme, every backjump that the iSAT core performs to resolve a conflict discards all propagations made on the decision levels that are undone. Sometimes the learned conflict clause may exclude so much of the search space that none of these discarded propagations has to be repeated ever again, but in general, it is much more likely that partial assignments made later in the search will coincide with those made within the excluded branch. Having discarded the corresponding deduction results, there is no choice then but to repeat these deductions. With the large amount of runtime invested in the ODE deductions, this approach was quickly discovered to be unsustainable.

In the iSAT-ODE solver as we present it here and in our later publications, e.g. [ERNF12a], we attempt to never repeat any ODE deduction. At the core of this attempt lies the learning of clauses that encode ODE deduction results.

Example 12 (learning of ODE deduction results)**.** In Figure 3.21, we illustrated, how the ODE solver detects that the flow invariant is left by all trajectories at a peeking enclosure computed for $t \approx 2.471$. We did not show that a constraint $x < -4$ was given for the next unwinding depth and allowed the solver to subsequently refute the reachability of the postbox. For this example, the ODE solver generates the following *deduction* object—reformatted slightly for better readability—which encapsulates all learned facts.

```
============= Deductions ========================
Implications.
   ((ode_var_2['x'](time) >= -1/2 is active))
=> (    ode_var_2['x'] >= -0.5
    AND ode_var_2['x]' >= -0.5)

        (((d.ode_var_1['y'] / d.time = ode_var_2['x']) is active)
    AND ((d.ode_var_2['x'] / d.time = (ode_var_1['y'] * -1)) is active)
    AND (ode_var_2['x'](time) >= -1/2 is active)
    AND ode_var_1['y'] >= -0.10000000000000000555111512312578270211815834045410156 25
    AND ode_var_1['y'] <= 0.1000000000000000055511151231257827021181583404541015625
    AND ode_var_2['x'] >= 0.899999999999999991182158029987476766109466552734375
    AND ode_var_2['x'] <= 1.10000000000000008881784197001252323389053344726562 5)
=> (delta_time < 2.4714277592842455177901683782692998647689819335937 5)

Conflicting boxes.
!(
        (ode_var_2['x'](time) >= -1/2 is active)
    AND ((d.ode_var_1['y'] / d.time = ode_var_2['x']) is active)
    AND ode_var_1['y'] >= -0.1000000000000000055511151231257827021181583404541015625
    AND ode_var_1['y'] <= 0.100000000000000005551115123125782702118158340454101562 5
    AND ode_var_1['y]' >= -100
    AND ode_var_1['y]' <= 100
    AND ((d.ode_var_2['x'] / d.time = (ode_var_1['y'] * -1)) is active)
    AND ode_var_2['x'] >= 0.899999999999999911821580299874767661094665527343 75
    AND ode_var_2['x'] <= 1.10000000000000008881784197001252323389053344726562 5
    AND ode_var_2['x]' >= -100
    AND ode_var_2['x]' <= -4
    AND delta_time >= 0
    AND delta_time <= 10
)
================================================
```

First, since this is the very first deduction in which the flow invariant occurs, it contains the implication that when the flow invariant *x(time)* $>= -0.5$ is active, the values of *x* and of *x'* shall satisfy this invariant, too. This has been derived syntactically and needs to be done only once.

The second implication is more complex. The first three conjuncts of its premise enumerate the active ODE and flow invariant constraints—a necessary condition for this deduction to be applicable. Similarly, the remainder of the premise describes the current valuation, here only the prebox since the postbox was not necessary to detect that the flow invariant is left. The conclusion is then the actual new knowledge: under the given circumstances, any value for *delta_time* can be discarded if it is larger than the identified time when the flow invariant is left.

The third element of this *deduction* object describes the observation that the postbox is not reachable from the prebox within the given valuation for *delta_time*. This is essentially the same as the box ruled out by a conflict clause, since it describes a region of the search space—spanned by the active triggers, the prebox, the postbox, and the duration valuation—that is known to contain no solutions.

Learning Deductions as Clauses. The *deduction* object that is generated by a *solver_run* is always structured into implications and conflict boxes just like those shown in the example. The implications use the current valuation or a subset thereof as premise and the deduced facts—like convex hulls computed from the relevant *enclosure* objects or tighter bounds for the *delta_time* variable—as conclusion. Having the full set of reasons for a deduction in the premise of an implication makes it valid universally, i.e. adding it to the formula allows the solver to propagate the conclusions whenever the premises hold. Similarly, the conflict boxes contain the relevant premises for the conflict to occur. They are therefore equally universal.

To instantiate the deduced facts for multiple BMC depths, the *valuation* object is used, which stores a mapping of symbolic variable names and BMC depths to their instances in the iSAT core. Again, we benefit from time invariance and from the use of the duration variable *delta_time* instead of the global *time*: since the deductions are premised by the context in which they are applicable, it is allowed to add them not only to the pair of BMC unwinding depths for which they were computed, but to all pairs of successive depths. This constraint replication [Sht00] avoids that the same deduction has to be repeated only because the same ODE problem occurs on another BMC depth. For a trajectory of a hybrid system that passes multiple times through the same region of the state space in the same mode, i.e. under the same dynamic laws, an ODE deduction made for one unwinding depth may therefore be used for propagation also at other steps as long as the durations of the respective flows are covered by the learned facts. More importantly, when the solver decides to explore e.g. an alternative switching sequence by assigning different values to the mode variables, the replicated constraints may become useful on a different unwinding depth than the one they were computed for.

The actual learning requires little more than transforming implications and conflict boxes into clause form and retrieving the correct variable instances. In the solver core, slight modifications are necessary to support learning of clauses

that are not necessarily conflicting with the current assignment. These changes are, however, rather technical (e.g. the search for suitable watched atoms) and do not change the algorithm in any fundamental way. The main difference is that after learning clauses encoding ODE deductions, the iSAT core needs to check whether a backjump is necessary and potentially *not* perform it—in contrast to conflict clauses, which always trigger such an undoing of decision levels. If no backjump occurs, at least some of the new clauses are already unit, since the premises of the implications from which they were constructed consist of the current valuation. This leads to their conclusions to be propagated as soon as the ODE solver returns the control flow to the solver core. The replicated clauses, which are learned for BMC depths for which they were not deduced, will most likely trigger neither conflict nor further deductions right away, but are simply added and may become unit at a later time.

Termination and Correctness. Were we using ODE deductions exactly like ICP, the proofs made in [FHR+07] would directly extend their termination and correctness guarantees to iSAT-ODE under the assumption that ODE deductions produce locally correct results. The correctness of this assumption is based on the use of validated numerics for the enclosure computation. These computations are also used in the refinement of the enclosures by reevaluation of the involved terms over temporal subranges. When terminating an enclosure prior to reaching the temporal horizon, we have found an enclosure that lies entirely outside the region admissible by the postbox. When claiming that no trajectory reaches the postbox from the prebox over the given interval of durations, we have refined all enclosures such that their intersection with the postbox has become empty. For pruning the postbox, we use the convex hull of all enclosures that have a non-empty intersection with it and the *delta_time* valuation. Similarly, for pruning the *delta_time* interval, we use all enclosures that have a non-empty intersection with the postbox.

Using ODE deductions via clause learning changes this argumentation only slightly. Since the clauses only make explicit the implications that would otherwise be inserted into the implication graph directly, this indirection of learning a clause and propagating its consequences does not change the correctness of the propagations. Due to the explicit encoding of each deduction's premise, neither does the constraint replication.

Sometimes not being able to compute an enclosure or failing to detect when trajectories leave the flow invariant or do not reach the postbox is detrimental to the precision of the solver, since the ODE constraints and flow invariants become practically ineffective in constraining the solution, but this is only more of the same overapproximation that the solver core has to struggle with even in case of arithmetic atoms. Since the solver does not claim a candidate solution box to actually contain a solution, spurious results are consistent with the chosen notion of correctness.

More questionable is the matter of termination. In the original algorithm infinite deduction chains can be stopped by checking, before inserting a deduction result in the implication graph, whether its progress exceeds the minimum progress threshold. Recalling the structural diagram of iSAT-ODE as shown in Figure 3.18, after learning deductions as clauses, the control flow returns to unit propagation and ICP. When their progress no longer exceeds the threshold,

the ODE solver is asked again to provide a deduction. Clearly, if no new bound has been generated in the solver core, this query will be exactly the same as the one that has been answered before. However, we have not yet discussed the one building block that is needed inside the ODE layer to detect that it has previously computed a deduction for an ODE problem and therefore should not compute the same deduction again. In fact, it is this "nothing new" answer that is necessary for termination. Since the topic is quite technical and otherwise independent of the computation of ODE deductions, we leave its detailed discussion for Section 3.4. Assuming, for the sake of argument, that prior to computing a deduction, the ODE solver checks whether the same ODE problem with the same valuation or a superset thereof that is only slightly larger (up to a similar threshold as used to terminate deduction chains) has been encountered before. If the current ODE problem is thereby identified as having been covered before, the ODE solver does not produce a new deduction. Termination then follows from the finite number of queries of sufficient progress that can possibly exist over the bounded search space and the consequently finite number of clauses that the ODE solver may produce.

3.3.7 Acceleration Techniques

The goal of learning ODE deductions is to accelerate the solver by making repetitions unnecessary. Similarly, storing VNODE-LP's internal state and using it for reevaluation instead of costly restarts of VNODE-LP aims at accelerated enclosure refinements. Finally, the option to perform ODE deductions starting from those unwinding depths and directions with the tightest valuations is motivated by our observation that this may avoid superfluous propagations. While these improvements have already been discussed, some additional acceleration techniques have not yet been mentioned.

Grouping of Independent Subsystems. Models of hybrid systems often consist of interacting components, and sometimes such parallel subsystems may interact not via their continuous evolutions, but merely by exchanging events. Between these points of interaction, the continuous evolutions of the components are independent of each other, just connected by the flow durations. On the level of a SAT modulo ODE formula, there are ODE constraints for a set of variables $\{x_1, \ldots, x_n\}$. For each of these, there are one or more ODE constraints of the form $\dot{x}_i = f(x_1, \ldots, x_n)$. If components do not interact, but evolve independently of each other for the same duration of a flow, the right-hand-side functions f of the ODE constraints do not depend on the set of all variables, but instead require only subsets of it. It is easy to identify such disjoint subsets syntactically and to thereby form independent groups of variables, ODE constraints, and flow invariants.

The advantage of identifying these independent subsystems is threefold. First, the algorithms to generate an enclosure involve superlinear computations, e.g. the matrix operations, needed for coordinate transformations. In [Ned99, pp. 64f] the complexity of the underlying algorithms as implemented in a predecessor of VNODE-LP are analyzed to some detail, the highest occurring order there is cubic in the number of dimensions. Our optimization loops that try to refine the boxes with the largest impact on the convex hull additionally need to repeat these refinements for each dimension and direction individually.

Decreasing the dimensionality can hence be expected to reduce the cost of the individual enclosures and the number of enclosures that need to be refined per system. The second advantage lies in the increased generality of the learned deductions. Since the deductions computed for an ODE problem depend only on a subset of the variables, the premise of the learned deduction may be satisfied earlier and therefore the propagation be more effective. Finally, the independent deductions also mean that a change to one group's valuation only requires a new computation of a deduction for that group, but not for the independent dimensions.

The cost of this optimization on the other hand is relatively low. Identifying the groups needs only be done once, initially, and when selecting an ODE problem for propagation, instead of picking a BMC depth and direction, the ODE solver merely needs to additionally pick a group. Both operations are rather inexpensive and likely to be offset by the advantage of more general constraints and accelerated enclosure computations. However, for models in which all dimensions are inter-dependent, the optimization obviously cannot pay off, but causes only very mild run-time penalties and can in fact be controlled by a command-line switch.

Splitting Heuristics. When we discussed the splitting heuristics used by the iSAT core, we pointed out that they can have a great influence on the performance of the solver, but that one should in general not expect to find an optimal heuristic, since it might lead straight towards also finding a polynomial-time SAT solver. We have nothing to take back from this position, but there is—as is often the case with heuristics—a compelling argument, why in the case of iSAT-ODE some variables should be split first. It is certainly not always true and we can only point to our experimental results in the next chapter for some empirical validation, but we think that it makes sense to split the discrete variables prior to splitting the continuous ones.

Since ODE deductions are expensive, they should not be performed for ODE problems that are irrelevant for the solution. Additionally, computing ODE enclosures for large initial sets over large time horizons can be quite costly, especially in the non-linear case when this may lead to decreasing step-sizes and a large number of enclosure computations before the solver finally gives up and returns without a deduction.

If the solver core is allowed to split continuous variables, the ODE solver may have to compute enclosures for the valuation tightened by this split—as long as an ODE problem has already become active for the involved variables. If these splits occur while there are still undecided discrete variables, the solver has not even established that a boolean abstraction of the solution is feasible. The ODE deductions made for some part of the trajectory may therefore later turn out to be superfluous because they belong to a trajectory that cannot be extended over the remaining steps.

Our *"discrete first"* heuristic enforces that the continuous variables are split only after all discrete variables have reached points as their valuations—be it by deduction or decision. As a consequence, the solver must first satisfy a boolean abstraction of the model before it may explore the continuous state space by splitting. Conflicts in the sequence of discrete modes are hence detected before the ODE solver could be forced to compute more and more propagations for some

part of a trace that cannot even be completed on the discrete level. Knowing the switching sequence, the constraints representing the transition guards and actions from the hybrid system are also more likely to have become unit and may thereby reduce the width of the valuation before an ODE deduction is attempted.

Enclosure Resolutions. In the context of the ODE enclosure computation, we have mentioned the influence of temporal and spatial thresholds to decide when to stop the refinement. Also VNODE-LP can be given a lower bound for the admissible step size, such that it stops the integration when its automatically-computed step size exceeds this threshold. Further parameters in VNODE-LP are the absolute and relative tolerances and the order of the Taylor series. Except for these tolerances and the order of the series expansion, we make use of this parameter set to influence the precision with which the enclosures are computed and refined.

Intuitively, the search method performed by the combined iSAT-ODE solver can be seen as an iterative refinement of a valuation until it becomes either a candidate solution, which can be reported, or exhibits a conflict, in which case an alternative must be explored. Starting with the entire domain from the declaration as initial box, the solver will not report a candidate solution until each variable has a valuation whose width is below the minimum splitting width. Again, there is no ultimate truth, but there are some strong arguments why it makes sense not to perform enclosures with the highest possible resolution when the search space is still very large, but instead to change the parameters that influence the precision and speed of ODE deduction dynamically.

We have implemented a two-step approach. The first step depends on the entirety of variables, not only on those known by the ODE solver. To this purpose, we have added a method *can_there_be_another_split* to the solver core that compares all interval widths with the minimum splitting width and reports whether there is still a variable that could be split. Doing this right before calling the ODE solver, an *enclosure_resolution* object can be initialized with a coarse resolution if another split is possible or with a fine resolution otherwise. These resolutions are derived (using fixed coefficients) from the minimum splitting width and absolute bound progress parameters, which are user-supplied. Deductions in the ODE solver can therefore be requested for exactly two different resolutions. When there is still at least one variable with a width above the minimum splitting width, the reported deduction will be made for the coarser resolution, which is likely to accelerate the computation inside the ODE solver, but may lead to imprecise results. To avoid that this imprecision influences the quality of a candidate solution box, the finer resolution is given when no further splits are possible under the current interval width. The ODE solver's detection of whether a particular deduction has been done before is extended in such a way that it differentiates this detection on the requested resolution, i.e. it will now repeat all deductions that have only been done for a coarser resolution. Thereby, the new deduction may unveil a conflict, which allows the solver core to discard the current valuation—avoiding a spurious counter example, which would otherwise be caused by the lower deduction precision. When the ODE solver is called with the coarse resolution and produces a deduction, it will after further ICP and unit propagation be called again, such that there is no risk that the last deduction

made by the ODE solver would ever be made with the coarse resolution setting.

This first step trades in the promise of never repeating an ODE deduction for being able to first computing it at higher speed with lower precision and hopefully only once, right before a candidate solution would be reported, having to repeat it with the otherwise requested finer default precision. On the level of the iSAT core, the lower precision weakens deduction as a whole and may therefore necessitate further splitting. Balancing the effort spent on deduction and the risk of an increased amount of branching is a highly heuristic matter.

The second step in manipulating the enclosure resolution is made optional and is controlled by a command-line switch since it may cause the introduction of even more repetitions and larger amounts of imprecision for earlier deductions than the first step. When activated, the ODE solver compares the valuation widths of each of the ODE problem's variables with their entire domain and with the minimum splitting width—leading to a relative measure, how far away from a solution the solver still is with respect to this subset of variables. This leads to a *dynamic resolution* choice on a continuous scale between the coarsest and finest admissible resolutions.

The potential advantage lies in the even coarser resolutions that may speed up especially the earliest deductions even further, when valuations are still very wide and exactness seems not really relevant yet. On the other hand, however, this assumption may be quite mistaken, since maybe a single precise early deduction may allow the solver to prune off large parts of the search space—the balance of decisions and deductions may be changed too strongly in this case. Even worse, at least the choice of a large minimal admissible step size in VNODE-LP can cause enclosure computations to fail for ODE problems that can very well be analyzed with finer resolution. This aspect might be improved by taking into account also different settings for orders and tolerances, but this has not yet been attempted in our approach. Additionally, having not merely two levels of precision, but a continuous scale, it may very well happen that a deduction needs to be repeated a number of times with increased precision.

3.3.8 Direction Deduction

Although we have not yet discussed how the detection of previously encountered queries works and have postponed the discussion of bracketing systems, the description of the iSAT-ODE algorithm is essentially complete. However, there is one particular type of spurious result that is quite astonishing and that cannot be ruled out by the deduction mechanism that we have discussed so far.

Example 13 (iSAT-ODE without direction deduction)**.** Consider the example in Figure 3.25. The model on the left encodes a hybrid system with two modes, one continuous variable, strict alternation of jumps and flows, and a counter that is incremented with each step. The transition system enforces that in both modes the slope is strictly positive. It also requires that the automaton alternates between the two modes and that each flow has a strictly positive duration. The mode invariants require that in mode zero *x(time) <= 0* and in mode one *x(time) >= 0*. Initially, x must take a value below -5.

From the strict alternation of the two modes combined with their flow invariants follows that jumps can only occur when $x = 0$. However, the dynamic laws together with the constraint *delta_time* > 0 cause x to always grow during

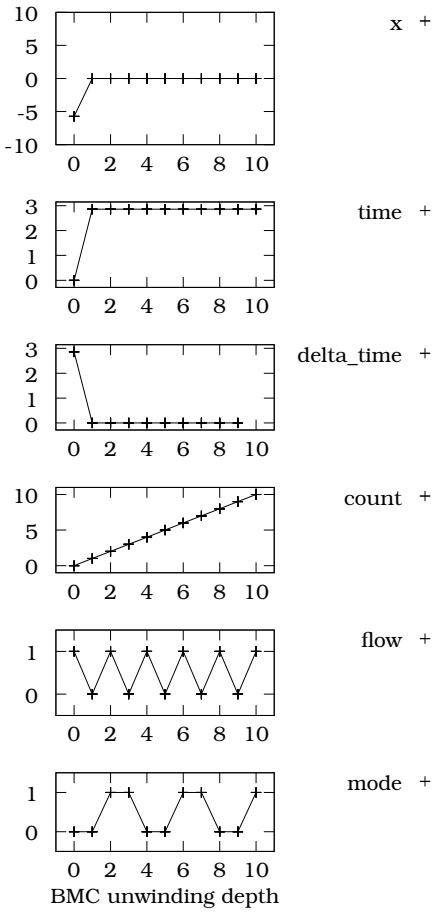
```
DECL
    float [−10,10] x; −− Continuous state space.
    int [0,1] mode;   −− Discrete state space.

    float [0,100] time;
    float [0,100] delta_time;

    boole flow;       −− !flow = jump.
    int [0,10] count;
INIT
    time = 0;
    x <= −5;  −− Initial continuous state;
    flow;     −− Start with flow.
    mode = 0; −− Initial mode.
    count = 0;
TRANS
    time' = time + delta_time; −− Progress of time.
    flow <−> !flow'; −− Alternating jumps & flows.
    count' = count + 1;

    −− Flows.
    −− ... stay in the same mode
    flow −> mode' = mode;
    −− ... shall really take time.
    flow −> delta_time > 0;
    −− ... in mode zero (below 0): slope 2.
    flow and mode = 0 −> (d.x / d.time = 2);
    flow and mode = 0 −> (x(time) <= 0);
    −− ... in mode one (above 0): slope 1.
    flow and mode = 1 −> (d.x / d.time = 1);
    flow and mode = 1 −> (x(time) >= 0);

    −− Jumps.
    −− ... do not take time and keep x constant.
    !flow −> delta_time = 0 and x' = x;
    −− ... alternate between both modes.
    !flow and mode = 0 −> mode' = 1;
    !flow and mode = 1 −> mode' = 0;
TARGET
    −− Find a 10−step trajectory ending in x=0.
    count = 10;
    x = 0;
```



Excerpt from the candidate solution box for the *delta_time* and *x* variables:

```
delta_time (float):
 @0:  [2.8573143,2.86155426)
          (width: 0.00423995653)
 @1:  [0,0] (point interval)
 @2:  (0,0.00012208) (width: 0.000122070312)
 @3:  [0,0] (point interval)
 @4:  (0,0.00012217) (width: 0.000122167969)
 @5:  [0,0] (point interval)
 @6:  (0,0.00012208) (width: 0.000122070312)
 @7:  [0,0] (point interval)
 @8:  (0,0.00012217) (width: 0.000122167969)
 @9:  [0,0] (point interval)
```

```
x (float):
 @0:  [−5.72213196,−5.71471904]
          (width: 0.00741291622)
 @1:  [−0,0] (point interval)
 @2:  [0,0] (point interval)
 @3:  [0,0] (point interval)
 @4:  [0,0] (point interval)
 @5:  [−0,0] (point interval)
 @6:  [0,0] (point interval)
 @7:  [0,0] (point interval)
 @8:  [0,0] (point interval)
 @9:  [−0,0] (point interval)
 @10:  [0,0] (point interval)
```

Figure 3.25: Lack of direction deduction leads to spurious solution traces.

flows. Combined, these constraints rule out any trajectory that comes back to 0 after the second flow. As a consequence, there cannot be a trajectory with more than one jump at all, since jumps only occur for $x = 0$. Surprisingly, the candidate solution box and its visualization on the right of Figure 3.25 show that the deduction mechanisms discussed so far are unable to prove this seemingly simple property.

The example has been crafted specifically to expose this weakness, but the underlying problem has actually become visible during the analysis of one of the benchmarks that we will discuss in the next chapter. In that example, we will analyze a time-bounded property, however, since the solver cannot prove that time really progresses in each step even when we encode it as explicitly as in the example here, from time-boundedness we cannot derive step-boundedness and therefore do not know up to which BMC unwinding depth we need to analyze the system.

The underlying reason for the behavior, however, is rather simple. If we take e.g. the boxes shown for the x-dimension in Figure 3.22, we can see that even though every exact solution trajectory starting in the prebox will decrease in value for any positive duration, the very first enclosure box must include this prebox entirely—no matter how much temporal refinement is performed. This observation is true not only in these examples, but holds in general: the first enclosure must contain the prebox, since it must cover the behavior for a duration of zero.

Fortunately, the problem can be mitigated, leading to what we have introduced as *direction deduction* in [ERNF11]. The idea is as simple as it is effective: for as long a prefix of the sequence of *enclosure* objects as possible, try to find a unique sign of the derivative. Separately for each dimension, the *solver_run* therefore evaluates over the first enclosure the right-hand side of this dimension's ODE constraint using the same syntax graph representation that we also pass to VNODE-LP. If the resulting interval does not contain zero, the sign of this right-hand-side function is either positive or negative and so the slope is determined at least for a duration equivalent to this *enclosure* object's temporal validity. The computation continues with the next enclosure until the sign is no longer unique. The previously computed maximum duration is then taken to generate a direction deduction for this variable.

In the example, the following direction deduction for x is computed.

```
      (((d.ode_var_1['x'] / d.time = 2) is active)
   AND (ode_var_1['x'](time) <= 0 is active)
   AND ode_var_1['x'] >= -10
   AND ode_var_1['x'] <= -5
   AND delta_time > 0
   AND delta_time <= 16.875099999999999766941982670687139034271240234375)
=> (ode_var_1['x]' > ode_var_1['x'])
```

The upper bound of *delta_time* in this case is determined by the detected leaving of the flow invariant, without any further refinement. The implication's premise holds for the current valuation such that the conclusion can be used by ICP once the deduction is learned. This conclusion encodes the direction by the constraint $x' > x$ and allows iSAT-ODE to rule out any trajectory in which the successor instance of x has the same value, i.e. it is a mitigation for the spurious trajectory that had x stuck at zero despite some positive time passage.

In fact, this direction deduction is a very light-weight form of a linear overapproximation. Instead of using only the sign, one could also learn conclusions like

$x' >= x + 3.2 * delta\_time$ when a lower bound for the slope of x has been computed as 3.2 for such a prefix or for the entire trajectory. Whether this additional information would pay off during search—on the one hand it strengthens ICP, on the other hand it increases the formula size with relatively weak information about the trajectory—is an open question. Approaches like the HSolver model checker [RS07] rely quite heavily on (unbounded) linear arguments.

3.4 Avoiding Repeated Enclosures

Storing ODE deductions in clauses needs to be complemented by the ability to detect when an ODE problem has already been encountered before. Since the clauses are stored anyway, the seemingly cleanest solution would be to detect when a deduction request is made for a valuation that follows directly from a previously learned deduction. Such a detection would have to be added to the solver core, since only there is it possible to access the implication graph, which can provide the reasons for the current valuation. This approach would let learning and avoidance of repeated deductions be based on the same storage: learned deductions in the formula.

However, there are a number of problems with this approach. Sometimes ODE deductions cannot be computed for a given ODE problem, e.g. when the valuation is still very wide and the enclosures computed by VNODE-LP diverge before the temporal horizon is reached. In these cases, there is knowledge about the ODE problem, namely that the ODE solver cannot produce a deduction, but this knowledge is not encoded as a clause in the formula. At the same time, these failed deduction attempts can be quite costly when they involve large numbers of small enclosure steps, converging to some time point before the horizon. Attempting to repeat a deduction for the same problem with the same resolution has no benefit, since it must fail again. Not being able to detect these cases would therefore waste valuable computation time on cases that do not provide any pruning at all.

Secondly, ODE deductions are resolution dependent. Unless the current resolution were somehow encoded in the learned clauses, it would be impossible to know which resolution the stored deduction was computed with. The whole point of having different resolutions is, however, to be able to first compute a coarse enclosure and later repeat the computation with finer resolution if necessary. Without knowledge about the resolution, the solver could not decide whether a repetition with the current resolution might yield better results.

Thirdly, adding to the solver core a potentially complex tracking of which part of the current valuation results directly from an ODE deduction would make it harder to transplant the ODE solving layer to future iSAT versions, since this functionality would have to be integrated tightly with iSAT's data structures.

For these reasons, we have decided to add the detection of previous queries to the ODE solver and accept the price of storing learned facts both within the formula and in these additional data structures. Fortunately, this approach has then allowed us to employ what was originally intended to be purely used for the detection of duplicate queries, also to caching of *solver_run* objects—allowing refinements for only slightly tighter valuations to be computed without having to recompute e.g. an enclosure up the horizon.

Finding Stored Boxes. To detect when the combination of current valuation, group, resolution, ODE constraints, and flow invariants corresponds to a previously encountered query, we add an *ode_cache* to the ODE solver. This structure encapsulates all previous queries under several caching layers. First, each group of independent ODE systems has its own set of caches. The second layer orders caches by their associated resolution. Underneath, caches are distinguished by the activated ODE constraints and flow invariants—represented by a bit vector that encodes the activation pattern. The innermost layer then stores the actual valuations.

While the outer layers can be implemented using standard containers, the inner cache needs to be able to answer the rather uncommon question whether a given box is covered (with limited margin) by one of the stored boxes. We have explored two different approaches to this problem. The first one could be called a projection-based approach. Iterating over all dimensions, it finds those boxes whose interval in the current dimension covers the interval of the to-be-searched box. Intersecting the results, it retrieves the set of covering boxes. During our initial experiments, especially when we had not yet implemented many of the acceleration techniques discussed in the previous section, this approach was sufficiently fast. However, with increased efficiency of the deduction routine and hence growing number of stored valuations, it has proven to be quite costly in terms of runtime for larger cache sizes, since the individual projections may become rather large sets.

Our second approach therefore attempts to avoid these early projections. It is strongly inspired by search trees, especially k-d trees [Ben75], and to some degree also by decision diagrams, which are successfully used in model checking, e.g. as binary decision diagrams, but also in interval contexts as interval decision diagrams (IDDs) [ST98]. While IDDs represent functions by case distinctions, which are encoded in the graph, our case is rather different, since we are only interested in finding a set of elements. We were unable to find prior work in the literature that is more directly related to our requirements, but find it possible that similar solutions have arisen in other applications e.g. to detect coverage in 3D graphics.

A necessary condition for any box $\vec{A}$ to cover the currently searched box $\vec{S}$ is that there is no point inside $\vec{S}$ that is not inside $\vec{A}$. This condition is very helpful in providing a branching rule that partitions the set of all stored boxes. For the leaf nodes in our search tree, we use a container, *box_store*, that can store boxes and for a small number of elements offers a sufficiently efficient projection-based check of containment. Whenever a new box is added to this leaf node and its capacity exceeded, we try to identify a point $\vec{p}$ that splits this set of stored boxes. For any two non-identical boxes, there is at least one point that lies in one of these boxes and not in the other one, obtainable by picking in the differing dimensions' intervals a point between the borders. In practice, since floating-point numbers are not dense, $\vec{p}$ may lie on the border of the outer box, but still partition the set. Each inner *tree_node* therefore holds the point $\vec{p}$ and pointers to two subtrees, one containing only elements that cover $\vec{p}$, the other containing those elements that do not cover it.

When given a box $\vec{S}$, the search simply needs to check at each encountered *tree_node* containment of its stored point $\vec{p}$. If $\vec{p}$ is inside $\vec{S}$, it can rule out the subtree of boxes that do not contain the point and recursively investigate the other subtree. Otherwise, $\vec{p}$ is not indicative and both subtrees can contain

covering boxes. However, if $\vec{p}$ is farther away from the border of $\vec{S}$ than the given threshold, boxes that cover $\vec{p}$ will be too large and therefore need not to be examined either. Finally arriving at *box_store* containers in the identified relevant leaf nodes, the candidate boxes can easily be checked for coverage and the size of the margin be compared to the threshold.

Again, there are several technical details that are better left to the source code, primarily the rebalancing of the search tree and the selection of split points to achieve an even distribution of boxes. In our experiments, imbalanced trees were far less problematic than originally thought. Although we implemented shifting of nodes from deep subtrees to shallower parts, after some experimentation with our benchmarks, the surprisingly simple setting of allowing only two nodes in the *box_store* leaf nodes and not using rebalancing at all lead to a sufficiently efficient detection of stored boxes, such that detection of previously encountered queries no longer accounts for relevant portions of the solver's runtime. Since this may cause search to become worst-case linear in the number of stored boxes, we believe that further improvements in the speed of the actual deduction routines will necessitate more active care about rebalancing and finding suitable leaf node capacities. The current situation, however, is one where deductions are so much more costly than lookup that further optimization of the caching layer does not pay off.

Use of Caches. Instantiating *ode_cache* objects for the four combinations of forward and backward direction with either just the prebox and *delta_time* valuation or with prebox, postbox, and *delta_time* valuation, the ODE solver gains the ability to perform lookups separately for these combinations.

A side-product of finding out that a valuation has been the subject of a previous query, is a pointer to the stored element. This pointer can be used as a unique key to access a cache of *solver_run* objects. In the *ode_cache_solver_runs* class, we implement such a second cache, which holds a limited number of solver runs. We have not extensively benchmarked different settings, but have chosen a value of 100 as the capacity of this cache, which could be subjected to further experimentation in the future. This cache contains the stored *solver_run* objects in a map that allows logarithmic lookup complexity and additionally keeps a sorted list of all stored objects, which records the age of the last access to an element. Replacement follows a *least recently used* strategy—as is standard in many other caching applications.

Since boxes are not only retrieved when they are equal to the queried one, but also when they extend to a slightly larger valuation covering it, this second cache can be seen as a scheme to continue a previous deduction for a valuation that has been refined only slightly by ICP in the solver core. If the valuation has changed only with respect to the *delta_time* valuation or to the postbox, this reuse causes no loss of precision at all. If the prebox has been tightened slightly, but the difference does not exceed the threshold for reuse, the new deduction will not be as tight as one that could be computed with the tighter prebox. This threshold is therefore part of the resolution, which is controlled by the current width of the valuation.

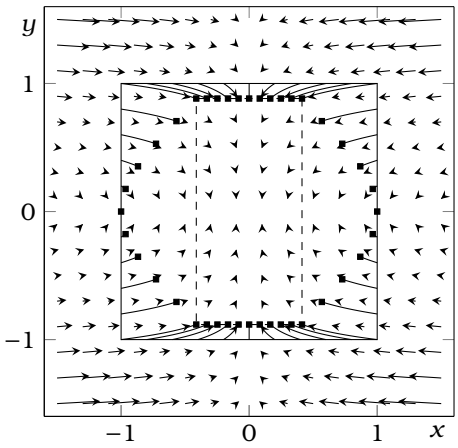


Figure 3.26: In general, following the corner trajectories does not produce an enclosure. ODE system: $\dot{x} = -x \cdot y^2$ and $\dot{y} = -0.125 \cdot y$, solution trajectories emerging from the border of the initial set with their final values after duration 1 marked.

3.5 Building Bracketing Systems Automatically

In the one-dimensional case, the solution functions of an ODE cannot cross each other. Two trajectories hence trap all others that start between them. In two dimensions, following only the trajectories emerging from the corner points, however, leaves open the possibility that a point on the edge of the initial box leads to a trajectory that escapes the convex hull spanned by the corner trajectories—see Figure 3.26 for an example. In general, enclosures of ODE problems are therefore computed using the entire prebox as initial set. Nonetheless, intuitively there seem to be cases when the trajectories that emerge from some of the corners of an initial box suffice to describe the solution set. Intuition even suffices to come to the conclusion that monotonicity must be a central criterion to detect these cases and to decide which points to choose.

Bracketing Systems. In [RMC09, RMC10] Ramdani et al. introduced the necessary theoretical foundation for enclosing solution sets by selecting two corners of the initial box and computing enclosures for a modified ODE system, the *bracketing system*, whose solution trajectories enclose all solutions of the original system. The major benefit of using the bracketing system instead of using VNODE-LP directly as presented above lies in the smaller initial set that results from choosing points instead of using the entire prebox. Especially in the case of non-linear ODEs, large initial sets may cause significant amounts of overestimation that render the computed enclosures useless for propagation. Having point-valued initial conditions, the amount of overestimation accumulating in the enclosure of the emerging solution trajectories often remains much smaller, leading to tight enclosures for longer durations. On the other hand, however, the set enclosed by the bracketing trajectories' enclosures may grow significantly. In some cases, especially those that benefit most from coordinate transformations, this growth is much stronger than that obtained from using VNODE-LP directly

on the original problem. Bracketing enclosures and the direct method need therefore be combined to exploit their complementarity.

One of the underlying observations for the theoretical work is that some systems are *monotone* in that their "flows preserve a suitable partial ordering on states, hence on *initial conditions*" [RMC10]. We will not repeat the theory, references to the underlying mathematical literature, and proofs presented in [RMC09, RMC10], but instead concentrate on the algorithmic aspects and the integration into ODE deductions, since the sole contribution of this thesis with respect to bracketing systems is their *automatic computation* (and their use in iSAT-ODE) in contrast to the primarily manual approach followed in the literature. We should point out, that our implementation of bracketing systems does not fulfill the entire potential opened up by this theory, in particular, it does not try to convert an ODE system into a "cooperative" [RMC10] one, though this might be an extension that could increase the applicability of the approach.

Signs of the Jacobian Matrix. The first step in determining the applicability of the bracketing approach and the construction of the bracketing system lies in the computation of the signs of the off-diagonal elements of the Jacobian matrix. Given as input is an ODE system $\dot{\vec{x}} = \vec{f}(\vec{x})$ represented by a vector of right-hand side syntax graphs. In the *bracketing_system* object a data structure *jacobian_matrix* for the Jacobian matrix

$$\mathcal{J} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ & \vdots & \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

is initialized via the *identify_relevant_partial_derivatives* method by marking the (i,j)-th element as *relevant* if x_j occurs in the expression tree of f_i, i.e. it is known that this entry is not trivially zero. The diagonal elements of the matrix are always marked as not relevant since they are not needed for the construction of the bracketing system.

Afterwards, and whenever the signs of the Jacobian need to be determined, the *evaluate_partial_derivatives* method is called, which takes as parameter a box $\vec{X}$. For each element of the *jacobian_matrix* that is marked relevant, the bounds of the partial derivatives over the given box must be computed. Internally, this method uses FADBAD++ and hence relies on the same automatic differentiation routines that are used in VNODE-LP. Since these evaluation results are sometimes too coarse to determine a unique sign, we have—after discussing these issues with Nacim Ramdani—experimented with an improved evaluation that uses a local optimization loop and the second-order partial derivatives to get a tighter interval evaluation of the Jacobian in case the direct evaluation contains positive and negative numbers and hence no unambiguous sign information. Once more, the right balance of precision and runtime is a matter of progress thresholds—in this case a fixed maximum recursion depth for the splitting of the evaluation intervals during the optimization loop, which we have fixed at 20. This maximum depth has not been subjected to intensive experimentation, and may in fact be too large. Our main results have been obtained using only the direct evaluation via FADBAD++, such that we consider this optimization an experimental extension.

Constructing the Bracketing System. The simplest form of the bracketing system approach, to which our implementation is confined, requires that all relevant off-diagonal entries of the Jacobian have a uniquely determined sign. Under this condition, we introduce a $2n$-dimensional vector of bracketing variables

$$\begin{pmatrix} \overline{x_1} \\ \underline{x_1} \\ \vdots \\ \overline{x_n} \\ \underline{x_n} \end{pmatrix},$$

in which $\overline{x_i}$ represents the upper bracketing variable for x_i and $\underline{x_i}$ its lower bracketing variable.

The *generate_bracketing_system* method then creates the actual ODE system over these new variables. For variable $\underline{x_i}$, it traverses the right-hand-side syntax tree representing f_i and copies each node, except when reaching a node representing a variable. If that variable is x_i itself, the bracketing ODE expression tree receives $\underline{x_i}$. If the variable is x_j for any $j \neq i$, then there is a valid entry in the Jacobian and its sign determines which of the bracketing variables to use. For $\partial f_i/\partial x_j > 0$, the lower bracketing variable is chosen, i.e. $\underline{x_j}$. For $\partial f_i/\partial x_j < 0$, it is the upper bracketing variable $\overline{x_j}$. When building the ODE for $\overline{x_i}$, the syntax tree of f_i is traversed again and copied, except that variable replacement is done differently, i.e. when encountering x_i in the syntax tree, it is replaced by $\overline{x_i}$, when encountering x_j for $j \neq i$ and $\partial f_i/\partial x_j > 0$, it is replaced by $\overline{x_j}$, and finally any x_j with $\partial f_i/\partial x_j < 0$ is replaced by $\underline{x_j}$. The rule is relatively simple: for the variable representing the bracketing dimension itself take the same bracketing variable as the one for which the right-hand-side is currently being built, for all other variables, if the sign of the Jacobian is positive, take the same bracketing direction, if the sign is negative, take the opposite bracketing variable.

Integration Step for the Bracketing System. So far, all we have is a bracketing system that is valid for the initial set, over which the Jacobian has been computed. Similar to the "dynamic hybridization" approach discussed as related work in Section 2.4, which first computes a step and then checks whether the assumptions made in the computation still hold for the region that the step covers, our bracketing system implementation uses an *a-posteriori check* of its validity.

To compute a step, the bracketing system is given to a separate *VNODE* solver object, and an integration step from time t to time $t + h$ is performed from the current enclosure. Initially, as was our goal, this enclosure consists of a point. More precisely, the upper bracketing variables take the supremum of the initial box and the lower bracketing variables its infinum, i.e. $\underline{x_i}(0) = \inf(X_i(0))$ and $\overline{x_i}(0) = \sup(X_i(0))$. The upper bracketing variables together therefore take the greatest point from the initial box, the lower bracketing variables the smallest—the two corners needed to span the initial box. As in the direct case, the integration yields a new enclosure at time $t + h$ and an a-priori enclosure over $[t, t + h]$. Since the bracketing system relies on the assumption that the signs of the Jacobian are known and fixed, it is now paramount to check whether this assumption holds on the entire a-priori

enclosure. The *validate_jacobian_matrix_for_box* method therefore takes this box, checks whether it is a subset of the region for which the *jacobian_matrix* is known to be valid—initially the prebox—and if not, repeats the computation of the relevant partial derivatives over this new box. If the signs do not change, the region of validity can be extended and this step's result can be accepted as a valid bracketing enclosure. If the extension of the box over which the partial derivatives are evaluated leads to a sign change, the step is discarded and the validity of the bracketing system ends before $t + h$. As long as the bracketing system is valid, the box between the two bracketing enclosures is an enclosure of the original ODE system.

Example 14 (bracketing systems)**.** Consider the example, which we also presented in [ERNF11],

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{p} \\ \dot{q} \\ \dot{r} \\ \dot{s} \end{pmatrix} = \begin{pmatrix} -sx - \frac{px}{1+qy} + ry + 0.1 \\ sx - ry \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

with the following initial values: $x_0 \in [1, 1.4]$, $y_0 \in [0.00001, 0.4]$, $p_0 \in [0.9, 1.1]$, $q_0 \in [1.1, 1.3]$, $r_0 \in [0.45, 0.55]$, and $s_0 \in [0.2, 0.3]$.

First, the relevant entries of the Jacobian are identified:

```
not relevant   not valid      not valid      not valid      not valid      not valid
not valid      not relevant   not relevant   not relevant   not valid      not valid
not relevant   not relevant   not relevant   not relevant   not relevant   not relevant
not relevant   not relevant   not relevant   not relevant   not relevant   not relevant
not relevant   not relevant   not relevant   not relevant   not relevant   not relevant
not relevant   not relevant   not relevant   not relevant   not relevant   not relevant
```

using the same ordering as above and indicating with "not valid" those entries that need to be computed. Using the given initial values as a box over which to perform this computation, the first relevant entry, $\partial f_x/\partial y$ is found to be contained in $[0.878, 2.552]$ and similarly all other relevant entries, leading to the following Jacobian for the initial box (after some reformatting and reduction of post-decimal digits for better readability):

```
not relevant   [0.878,2.552] [-1.400,-0.657] [3.9e-6,0.616] [9.9e-6,0.401]   [-1.401,-1]
[0.199,0.301] not relevant   not relevant     not relevant   [-0.401,-9.9e-6] [1,1.401]
not relevant   not relevant   not relevant     not relevant   not relevant     not relevant
not relevant   not relevant   not relevant     not relevant   not relevant     not relevant
not relevant   not relevant   not relevant     not relevant   not relevant     not relevant
not relevant   not relevant   not relevant     not relevant   not relevant     not relevant
```

Each relevant entry has a unique sign information. A bracketing system can therefore be generated. We examine just one of the original dimensions:

```
In the bracketing system, the original ODE
    (d.ode_var_1['x'] / d.time =
        ((ode_var_2['r'] * ode_var_3['y'])
        + (1/10 + ((ode_var_4['s'] * (ode_var_1['x'] * -1))
        - ((ode_var_5['p'] * ode_var_1['x']))
        / ((ode_var_6['q'] * ode_var_3['y']) + 1))))))
 will be replaced by the lower bracketing ODE (dim 0)
    (d.ode_var_0['x:lower'] / d.time =
        ((ode_var_8['r:lower'] * ode_var_2['y:lower']))
        + (1/10 + ((ode_var_11['s:upper'] * (ode_var_0['x:lower'] * -1))
        - ((ode_var_5['p:upper'] * ode_var_0['x:lower']))
        / ((ode_var_6['q:lower'] * ode_var_2['y:lower']) + 1))))))
```

```
and the upper bracketing ODE (dim 1)
   (d.ode_var_1['x:upper'] / d.time =
       ((ode_var_9['r:upper'] * ode_var_3['y:upper'])
       + (1/10 + ((ode_var_10['s:lower'] * (ode_var_1['x:upper'] * -1))
       - ((ode_var_4['p:lower'] * ode_var_1['x:upper'])
       / ((ode_var_7['q:upper'] * ode_var_3['y:upper']) + 1))))))
```

The dimension numbers of the bracketing system start afresh at 0, but their symbolic names have been printed as well, allowing better traceability. For creating the zeroth dimension of the bracketing system, *x:lower*, i.e. $\underline{x}$, the entries of the first row of the *jacobian_matrix* are relevant. For example, the sign of the partial derivative of f_x with respect to r is positive, hence r is replaced by $\underline{r}$, shown as *r:lower*. In the upper bracketing dimension $\overline{x}$, it is replaced by $\overline{r}$ for the same reason.

The first integration step with the bracketing system starts from the initial box obtained from the two most extreme corners of the original prebox (again reformatted and with fewer post-decimal digits):

```
bracketing system prebox:
    pre(ode_var_0['x:lower'])(0) = [1, 1]
    pre(ode_var_1['x:upper'])(1) = [1.400, 1.401]
    pre(ode_var_2['y:lower'])(2) = [9.9e-06, 1.0e-05]
    pre(ode_var_3['y:upper'])(3) = [0.400, 0.401]
    pre(ode_var_4['p:lower'])(4) = [0.899, 0.900]
    pre(ode_var_5['p:upper'])(5) = [1.100, 1.101]
    pre(ode_var_6['q:lower'])(6) = [1.099, 1.100]
    pre(ode_var_7['q:upper'])(7) = [1.300, 1.301]
    pre(ode_var_8['r:lower'])(8) = [0.449, 0.450]
    pre(ode_var_9['r:upper'])(9) = [0.550, 0.551]
    pre(ode_var_10['s:lower'])(10) = [0.199, 0.200]
    pre(ode_var_11['s:upper'])(11) = [0.300, 0.301]
```

The bracketing system's a-priori enclosure is then extracted and converted back to the original dimensions by setting each $X_i := [\underline{x_i}, \overline{x_i}]$. The following box is obtained:

```
Checking validity of bracketing system for box:
   (ode_var_1['x'])(0) = [0.634, 1.401]
   (ode_var_3['y'])(1) = [9.999e-06, 0.468]
   (ode_var_5['p'])(2) = [0.899, 1.101]
   (ode_var_6['q'])(3) = [1.099, 1.301]
   (ode_var_2['r'])(4) = [0.449, 0.551]
   (ode_var_4['s'])(5) = [0.199, 0.301]
```

Since it is not a subset of the box for which the bracketing system's validity is known (i.e. currently only the original prebox), the signs of the Jacobian need to be reevaluated.

In this case, the check is successful, as none of the signs change. Another step can be performed with this bracketing system. In Figure 3.27, we show the computed a-priori enclosures of the bracketing system and compare them with the diverging direct enclosures at the points of time chosen by VNODE-LP's step size control.

Combination of the Direct and Bracketing Methods. To couple the direct and bracketing methods, the algorithm for computing an enclosure up to the temporal horizon, the *enclosure* data structure, and the refinement algorithms are slightly extended with respect to the description from Section 3.3. If bracketing systems are not disabled via a command-line switch, the *enclose_flowinv_aware* method, which computes an enclosure up to the horizon, interleaves the integration steps of the direct method with those of the bracketing system. For this purpose, it initializes data structures for both approaches and computes the

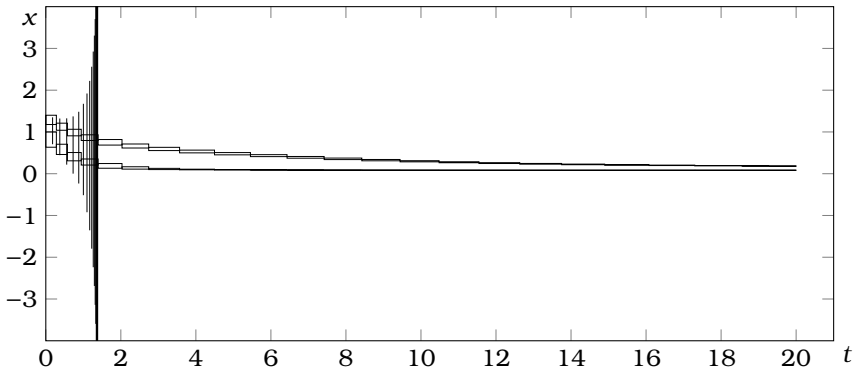


Figure 3.27: Comparison of bracketing system's enclosure for x in Example 14 with direct method. Direct enclosures diverge spatially and are aborted early, while bracketing enclosures lead to tight result for desired duration.

next step always with the method that "runs behind", i.e. if an enclosure with the direct method has been computed up to time t_d and with the bracketing system up to t_b, the next enclosure is computed with the direct method if $t_d \leq t_b$ and with the bracketing system otherwise. The sequence of *enclosure* objects is then extended to always cover $\min(t_d, t_b)$, i.e. the duration covered by both methods, unless one of them has failed and therefore had to be stopped. In the *enclosure* object, the internal data of both methods is stored to allow later refinement. The actual box stored in the *enclosure* object is obtained by intersecting the direct and bracketing results. During all later refinements, the reevaluation is done using the stored internal representation from both methods (or from the one that is valid in case the other one has failed) and intersecting the results. This combination therefore increases the deduction routine's robustness against one of the methods failing, as shown in Figure 3.27, in which the direct method alone would not have allowed an enclosure up to the horizon.

We have, again only as an experimental extension, also tried to automate the *hybrid bounding method* presented in [RMC09]. Theoretically, it can be understood as a preprocessing step on the automaton level that replaces a mode with submodes in which different bracketing systems for the same dynamic govern the continuous evolution—depending on the region of the state space and the signs of the Jacobian in them. In a central submode, the original dynamic is used instead of the bracketing systems, allowing to bridge gaps caused by regions in which the signs of the Jacobian are not uniquely determined. Our prototypical implementation does not try to apply this method on the automaton level, but instead integrates it in the coupling of direct and bracketing methods in the deduction routine. Lazily, it tries to compute an enclosure with the bracketing and direct methods as shown above until the bracketing system is no longer valid. Computing the next step with the direct enclosure alone, after this step, it tries again to *restart the bracketing system* by analyzing the Jacobian over the new enclosure. Such restarts are successful if the region in which one of the partial derivatives changes its sign have been successfully left behind. While we have employed this combination successfully on an example, we have seen less positive effects on our larger benchmarks than we anticipated—probably

because the regions in which bracketing systems were not valid are too large or in too relevant parts of the search domain. In principle however, this kind of combination may facilitate the analysis of systems that need the bracketing approach for tight enclosures yet cross regions of the state space where at least the simple criterion of the Jacobian having uniquely determined signs does not hold.

Chapter 4

Experimental Evaluation

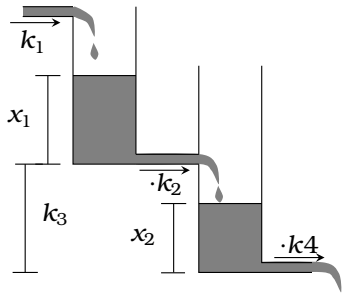
We have described the research problem, its context, and our algorithmic approach to its solution. As can be expected in any matter of undecidable nature that involves huge algorithmic complexities, heuristics, and problems with incompletenesses of the theoretical and practical kinds, "solution" is too big a word. Our solver copes well with some problems of our choosing, even with some taken from the literature, and it performs underwhelmingly on others. This chapter's task is to present these results, show applications, and thereby shed some light on these strengths and weaknesses. To some degree, we also provide comparisons to competitor tools to put these results into a better context.

As pointed out already in the introduction, the factual content of this thesis has been subject of our own prior publications—referenced on many occasions explicitly. In contrast to the other chapters, in which at least the text and illustrations describing this content were written to a large degree specifically for this thesis, this chapter draws most heavily from our prior publications not only in its content, but also in the text and figures. While minor changes have to be made for better presentation, notational consistency, layout, composition, or similar reasons, essentially, this chapter quotes our own publications.

4.1 Bounded Model Checking Case Studies

4.1.1 Two Tank System

The goal of this first benchmark model lies in the evaluation of the integrated tool and the influence of the different enclosure methods. A special focus lies on the effect of the algorithmic enhancements, chiefly the storing of VNODE-LP's internal state and its usage for accelerated reevaluations, the detection of trajectories leaving the region admissible under the activated flow invariants, and the improved tree-based caching algorithm. As a baseline for assessing the effect of these enhancements serves our previous implementation that we used to obtain results for the same benchmark in [ERNF11].



For $x_2 > k_3$:

$$\begin{pmatrix}\dot{x}_1\\ \dot{x}_2\end{pmatrix} = \begin{pmatrix} k_1 - k_2\sqrt{x_1 - x_2 + k_3} \\ k_2\sqrt{x_1 - x_2 + k_3} - k_4\sqrt{x_2}\end{pmatrix}$$

For $x_2 \leq k_3$:

$$\begin{pmatrix}\dot{x}_1\\ \dot{x}_2\end{pmatrix} = \begin{pmatrix} k_1 - k_2\sqrt{x_1} \\ k_2\sqrt{x_1} - k_4\sqrt{x_2}\end{pmatrix}$$

Figure 4.1: Structure and dynamics of the two tank hybrid system (from [SKHP97]).

We apply our solver to the two-tank model from [SKHP97], which has been frequently used as a case study for verification tools e.g. in [HHMWT00, RS07]. This system comprises two tanks connected by a tube. The first tank has an inflow of constantly $k_1 = 0.75$ volume units, and its base is $k_3 = 0.5$ length units above the base of the second tank. The connecting tube is characterized by a constant factor $k_2 = 1$, which also characterizes the outflow of the system as $k_4 = 1$.

Figure 4.1 illustrates this setting and formalizes the dynamic behavior of the liquid's height x_1 and x_2 in the two tanks. The system's behavior switches between two dynamics, when x_2 reaches the outlet from tank 1 and therefore exerts a counter pressure against the incoming flow. Note that the model is implicitly bounded to the case that $x_2 \leq x_1 + k_3$, since it does not provide the dynamics for the inverse direction. To understand better the dynamics of this system and the proof obligations we encoded, Figure 4.2 depicts simulated trajectories.

Similar to our earlier examples, we encode this model in the iSAT-ODE input language and use the ODEs from Figure 4.1 directly as constraints.[1]

Bounded Reachability. To validate the model, we first check bounded reachability properties. As can be assumed from Figure 4.2, there should not be any trajectory leading from region $\vec{D} = [0.70, 0.80] \times [0.45, 0.50]$ to $\vec{E} = [0.60, 0.80] \times [0.60, 0.65]$. This property has been verified by Henzinger et. al. using HyperTech [HHMWT00].

We restrict the global *time* ≤ 100 and each step duration *delta_time* ≤ 10. To avoid unnecessary non-determinism in the model, all steps are explicitly enforced in the transition relation to take the maximum possible duration. They may be shorter only if they reach the switching surface at $x_2 = k_3$, if the *time* $= 100$, or if (x_1, x_2) reaches $\vec{E}$.

In [ERNF11], we reported that our solver could prove unsatisfiability of this bounded property for up to 300 unwindings of the transition system within 3109.1 seconds on a 2.4 GHz AMD Opteron machine, which is also used for the runtime measurements presented here for this benchmark. As can be seen

[1] Original models and raw results from [ERNF11]:
http://www.avacs.org/fileadmin/Benchmarks/Open/iSAT_ODE_SEFM_2011_models.tar.gz
Updated models and raw results from [ERNF12a]:
http://www.avacs.org/fileadmin/Benchmarks/Open/iSAT_ODE_SoSyM_2012_models.tar.gz

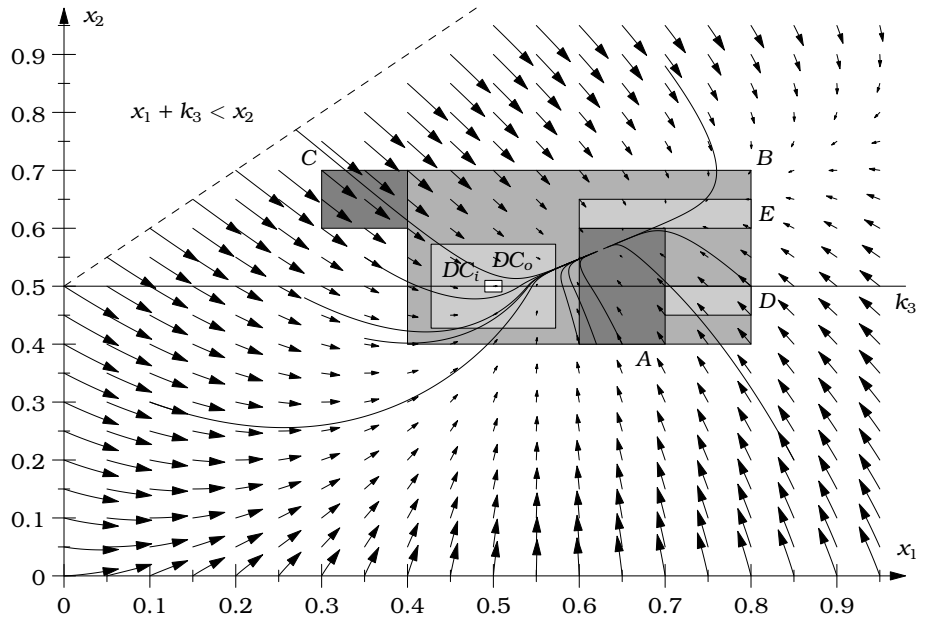


Figure 4.2: Simulated trajectories for the two tanks system, inner and outer bounds of the don't care mode, and regions $\vec{A}$ - $\vec{E}$ used in the different verification conditions.

from the runtime graphs in Figure 4.3, for the same model, this runtime can no longer be achieved by the current version of our tool, unless flow invariants provide additional pruning. For the version without flow invariants (graph "w/o flow invariants, all encl. methods"), the runtime has increased to 17098 seconds on the same machine, a 5.5-fold increase. Also, the previously reported flat development of the cumulative runtimes is no longer observable. A likely reason for this is that our previous implementation contained a subtle bug in the collection of reasons for direction deductions in that it did not add all active ODE constraints to the reason set and hence generated conflict clauses that sometimes were too general. Such clauses can potentially prune off too large parts of the search space (in the worst case including solutions) and may hence accelerate the search in an unjustified way.

Among the other possible explanations for the slow-down, we can safely rule out detrimental effects of our additional deduction mechanisms and optimized data structures. A negative impact from the caching behavior, which has been measured to take up only 13.2 seconds, and also of other changes in the ODE solver (e.g. the evaluation of stored Taylor coefficients for refinement) are unlikely, since our profiling indicates that only 2326.8 seconds are spent in total within functions of the ODE layer, indicating that the majority of the runtime was consumed by the iSAT core, i.e. for splitting, deductions, and conflict analysis.

We have also extended the benchmark to make use of the flow invariant feature that was not available in the version used in [ERNF11] by adding flow invariants for the variable domains and for capturing the mode invariants of $x_2(time) \leq k_3$ or $x_2(time) \geq k_3$. For this more elaborate benchmark version, iSAT-ODE is able to prove unsatisfiability for 300 unwinding depths within just

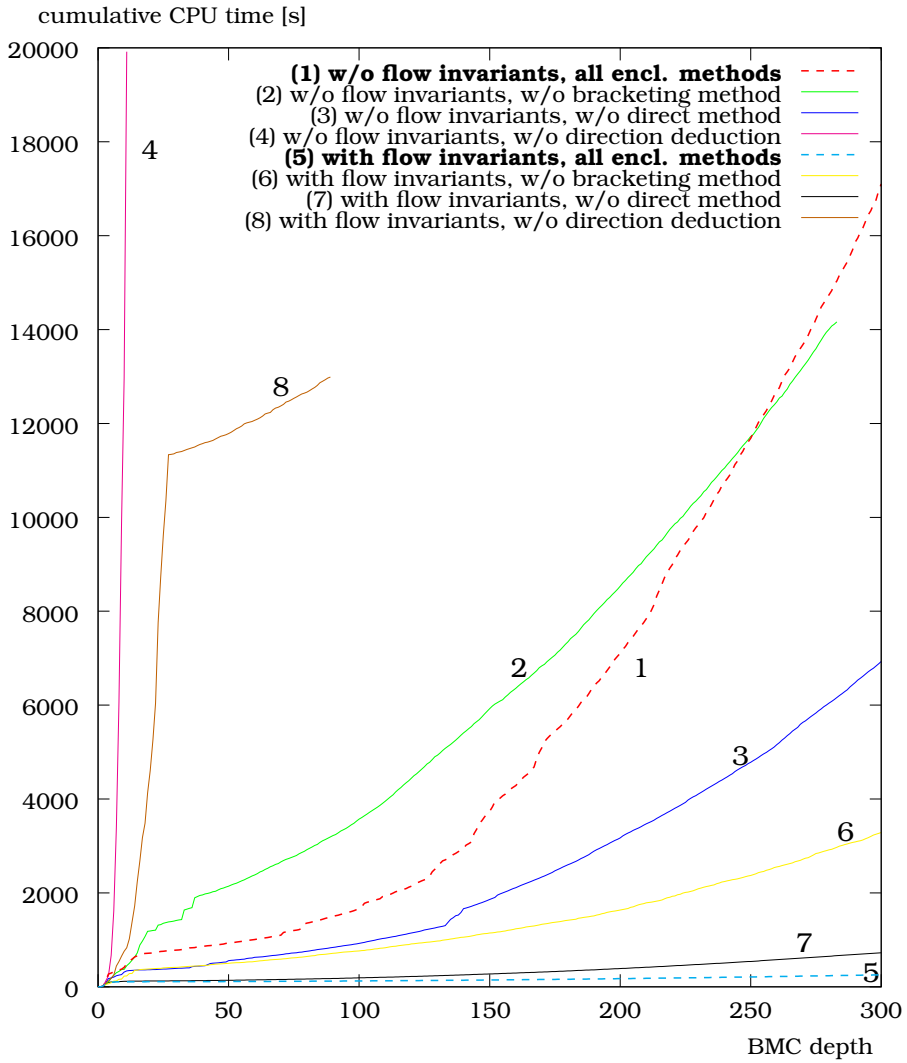


Figure 4.3: Cumulative CPU times for checking (un)reachability from region $\vec{D}$ to region $\vec{E}$ for the original variant of the two tank system without flow invariants and for a new variant which contains flow invariants for each mode. Comparison of the different solver settings, disabling one enclosure method at a time, i.e. comparing: all enclosure methods together, without bracketing, without direct enclosures, and without direction deduction.

255.7 seconds, a more than 12-fold improvement over the originally reported runtime.

Caching. To assess the quality of the tree-based cache implementation in light of the previously used projection-based cache sometimes using up significant portions of the total runtime, we have additionally measured the runtime spent in the ODE caching layer. Over all solver runs shown in Figure 4.3, the highest measured percentage is 2.5% of the total runtime spent within the ODE layer (measured for the original version of the model without flow invariants, and the solver set to not use direction deduction). In that case, the solver (before running into the 20000 seconds timeout) accumulates 544425 calls to the ODE solver, of which over 95% have been detected to be cached. We therefore conclude, that the cache implementation has been adequately prevented from becoming a dominant runtime factor even for large numbers of accesses.

Unbounded Trajectory Containment. Although the formula structure is a bounded unwinding of the transition system, inductive arguments may be used to prove unbounded properties. One can easily see that region $\vec{A} = [0.6, 0.7] \times [0.4, 0.6]$ contains an equilibrium point. However, the simulation also shows that there are trajectories leaving this region. We extend our model to show that trajectories can leave region $\vec{A}$ only on a bounded prefix, but thereafter stay in $\vec{A}$ forever.

First, we guess a $\tau > 0$ (supported by looking at some simulated trajectories). With

$$\mathcal{M}_l := \{\text{all trajectories of length} \geq l\},$$

from showing that

$$\begin{aligned} &\forall \vec{x} \in \mathcal{M}_{2\tau} : [0, 2\tau] \to \mathbb{R}^2 : \\ &\left(\vec{x}(0) \in \vec{A} \Rightarrow \forall t \in [\tau, 2\tau] : \vec{x}(t) \in \vec{A}\right) \end{aligned} \tag{4.1}$$

follows by inductive application of (4.1), as facilitated by time invariance,

$$\begin{aligned} &\Rightarrow \forall \vec{x} \in \mathcal{M}_{\infty} : [0, \infty) \to \mathbb{R}^2 : \\ &\left(\vec{x}(0) \in \vec{A} \Rightarrow \forall t \in [\tau, \infty) : \vec{x}(t) \in \vec{A}\right). \end{aligned}$$

Intuitively, we show that all trajectories of length 2τ stay in $\vec{A}$ for *delta_time* $\in [\tau, 2\tau]$ (ignoring their behavior for $[0, \tau)$). All unbounded trajectories must have these trajectories of length 2τ as prefix. At τ, they are thus (again) in $\vec{A}$. Due to time invariance, we can consider $(x_1, x_2)(\tau)$ as a new starting point. Since it lies in $\vec{A}$, we have already proven that for $[\tau + \tau, \tau + 2\tau]$, the trajectory will lie in $\vec{A}$ again. For the time in between, we already know that it is in $\vec{A}$. By repeating this process ad infinitum, we know that the trajectory can never leave $\vec{A}$ again.

Note that this proof is related to the idea of *region stability* [PW07] and can be thought of as a stabilization proof for an unknown (and maybe hard to characterize) sub-region $\vec{A}_I \subseteq \vec{A}$ into which all trajectories from $\vec{A}$ stabilize, and which is an invariant region for the system.

Table 4.1 summarizes runtimes for this proof using iSAT-ODE and the different enclosure methods. It also compares the current implementation with

	all			no bracketing			no direct			no direction		
depth	old	new	o/n	old	new	o/n	old	new	o/n	old	new	o/n
1	c, 111.9	c, 149.2	0.75	c, 42.0	c, 4.7	9.01	c, 61.5	c, 94.0	0.65	c, 111.5	c, 146.7	0.76
2	c, 467.5	c, 157.9	2.96	c, 981.0	c, 451.0	2.18	c, 346.3	c, 102.9	3.36	c, 342.0	c, 39.7	8.61
3	**u**, 674.0	**u**, 147.8	4.56	**u**, 5011.6	**u**, 196.9	25.45	**u**, 404.2	**u**, 96.5	4.19	c, 478.8	c, 126.0	3.80
4	**u**, 812.1	**u**, 237.2	3.42	**u**, 1995.1	**u**, 706.4	2.82	**u**, 499.1	**u**, 92.4	5.40	c, 547.5	c, 196.0	2.79
5	**u**, 986.0	**u**, 270.3	3.65	**u**, 2431.7	**u**, 276.1	8.81	**u**, 601.1	**u**, 125.9	4.77	c, 682.4	c, 243.7	2.80
6	**u**, 1126.1	**u**, 227.2	4.96	**u**, 3303.4	**u**, 466.7	7.08	**u**, 705.0	**u**, 227.3	3.10	c, 834.2	c, 191.7	4.35
7	**u**, 1277.2	**u**, 254.8	5.01	**u**, 2486.8	**u**, 224.7	11.07	**u**, 803.6	**u**, 143.2	5.61	c, 982.5	c, 328.6	2.99
8	**u**, 1451.4	**u**, 279.4	5.20	**u**, 5273.3	**u**, 406.5	12.97	**u**, 890.8	**u**, 159.6	5.58	c, 1115.7	c, 434.1	2.57
9	**u**, 1584.6	**u**, 328.2	4.83	**u**, 4905.2	**u**, 444.0	11.05	**u**, 966.5	**u**, 151.5	6.38	c, 1235.8	c, 1203.0	1.03
10	**u**, 1706.6	**u**, 312.2	5.47	**u**, 6396.1	**u**, 430.7	14.85	**u**, 1053.2	**u**, 152.2	6.92	c, 1356.0	c, 807.6	1.68

Table 4.1: Comparison of the old [ERNF11] and the new implementations on the two tank system for checking unbounded containment in region $\vec{A}$. Column *all* shows results and CPU times (seconds) when using all enclosure methods combined. In the subsequent columns, one of the methods is disabled. In the *o/n* columns, the old runtime is shown in multiples of the new runtime. In each cell, left to the runtime, "C" represents a candidate solution while "U" shows that the formula has been proven to be unsatisfiable and hence the proof of containment is successful.

the older one reported in [ERNF11]. Our model encodes the above proof scheme in the following way. If a trajectory exists that is shorter than 2τ or that reaches a point outside $\vec{A}$ in $\mathsf{time} \in [\tau, 2\tau]$, this trajectory satisfies the model. The proof is successful when the solver finds an unwinding depth k of the transition system upon which the model becomes *unsatisfiable*. Here, an unwinding depth of 3 suffices to prove the desired property. Without the direction deduction presented in Subsection 3.3.8, the solver fails to prove unsatisfiability, because it always finds counter examples that stay on the switching surface, spending there only tiny amounts of time. These trajectories satisfy the target condition of having $\mathsf{time} \leq 2\tau$ and do not allow proving (4.1). Direction deduction hence enables proving the property.

The runtimes show that the approach without the direct enclosure (using only bracketing enclosures and direction deductions) outperforms both, the restriction to the direct usage of VNODE-LP with direction deduction and the combination of all enclosure methods together on this benchmark in nearly all cases. The table also shows that the changes we made to our implementation have significantly accelerated the solver in nearly all cases for this benchmark instance, with slowdowns only occurring for unwinding depth one. Note that also for this and the following instances, the results reported for the old implementation from [ERNF11] may have been influenced by the incomplete set of reasons generated for direction deductions. We assume that those too-general deductions have caused at most an undue acceleration, since they prune off parts of the search space for which they should not have been valid.

Introducing Artificial Non-Determinism and Hysteresis. Trying a direct inductive proof for the region $\vec{B} = [0.4, 0.8] \times [0.4, 0.7]$ (i.e. showing that $\vec{B}$ cannot be left with one step of the transition system) fails with our tool since $\vec{B}$'s corner at $(0.4, 0.4)$ cannot be represented exactly by floating-point numbers. To compensate, $\vec{B}$ is overapproximated to capture rounding errors, hence includes points that lie slightly outside $\vec{B}$. Using the same proof scheme as above can be expected to work, as the simulated trajectories point inwards from the border of $\vec{B}$. Yet, applying this proof scheme, the solver finds trajectories that can

	all			no bracketing			no direct			no direction		
depth	old	new	o/n	old	new	o/n	old	new	o/n	old	new	o/n
1	c, 17.7	c, 2.6	6.82	c, 9.4	c, 1.2	8.09	c, 12.9	c, 1.6	7.91	c, 15.4	c, 2.6	5.96
2	c, 163.9	c, 10.2	16.02	c, 57.9	c, 5.2	11.08	c, 81.9	c, 6.9	11.82	c, 157.4	c, 8.7	18.14
3	c, 198.9	c, 16.2	12.24	c, 71.8	c, 8.9	8.03	c, 126.9	c, 10.8	11.78	c, 202.3	c, 12.6	16.02
4	c, 666.6	c, 18.0	37.07	c, 193.6	c, 8.4	22.94	c, 146.7	c, 12.2	11.99	c, 206.9	c, 14.4	14.41
5	**u**, 2334.2	**u**, 106.6	21.90	**u**, 3270.2	**u**, 62.6	52.20	c, 183.4	**u**, 67.7	(2.71)	c, 283.6	c, 15.8	17.96
6	**u**, 4615.6	**u**, 265.5	17.38	**u**, 1441.2	**u**, 59.2	24.36	c, 182.2	**u**, 146.3	(1.25)	c, 122.0	c, 18.1	6.75
7	**u**, 2967.1	**u**, 171.6	17.29	c, 1934.7	**u**, 75.5	(25.61)	c, 144.1	**u**, 106.2	(1.36)	c, 123.9	c, 21.2	5.84
8	**u**, 2559.0	**u**, 223.7	11.44	**u**, 2953.0	**u**, 74.3	39.72	c, 201.6	**u**, 398.4	(0.51)	c, 123.6	c, 21.2	5.84
9	**u**, 2184.1	**u**, 459.4	4.75	**u**, 4121.2	**u**, 115.1	35.80	c, 135.2	**u**, 181.6	(0.74)	c, 127.2	c, 21.2	5.98
10	**u**, 5541.6	**u**, 308.9	17.94	**u**, 7717.3	**u**, 166.3	46.39	c, 272.5	**u**, 333.1	(0.82)	c, 127.6	c, 21.8	5.86

Table 4.2: Comparison of results and CPU times (seconds) for checking unbounded containment in $\vec{B}$. The *old* columns again refer to our earlier implementation [ERNF11].

	all			no bracketing			no direct			no direction		
depth	old	new	o/n	old	new	o/n	old	new	o/n	old	new	o/n
1	c, 55.3	c, 5.9	9.34	c, 20.2	c, 1.6	12.42	c, 34.4	c, 4.3	7.93	c, 54.9	c, 6.2	8.79
2	c, 203.3	c, 24.7	8.22	c, 83.4	c, 6.9	12.09	c, 103.8	c, 17.4	5.96	c, 198.3	c, 25.7	7.71
3	c, 308.1	c, 35.1	8.78	c, 121.1	c, 11.4	10.62	c, 155.8	c, 24.3	6.41	c, 291.8	c, 40.3	7.24
4	c, 419.1	c, 56.1	7.47	c, 151.0	c, 15.2	9.94	c, 199.9	c, 35.7	5.60	c, 386.6	c, 54.3	7.12
5	c, 499.3	c, 60.9	8.20	c, 163.6	c, 71.2	2.30	c, 551.7	c, 42.0	13.12	c, 468.5	c, 103.0	4.55
6	c, 525.6	c, 73.0	7.20	c, 177.8	c, 44.6	3.99	c, 536.6	c, 51.3	10.46	c, 492.9	c, 74.0	6.66
7	c, 555.6	c, 102.7	5.41	c, 196.8	c, 34.2	5.75	c, 449.9	c, 100.2	4.49	c, 524.4	c, 106.6	4.92
8	c, 577.6	c, 94.8	6.10	c, 223.8	c, 49.1	4.56	c, 448.9	c, 62.7	7.15	c, 549.3	c, 98.1	5.60
9	c, 599.6	c, 492.9	1.22	c, 235.0	c, 69.6	3.38	c, 447.4	c, 89.2	5.02	c, 574.5	c, 176.5	3.25
10	c, 617.6	c, 93.8	6.59	c, 279.7	c, 52.1	5.37	c, 448.7	c, 214.2	2.10	c, 592.2	c, 159.8	3.71

Table 4.3: Comparison of results and CPU times (seconds) for checking unbounded containment in region $\vec{C}$ (not containing an equilibrium point) with old [ERNF11] and new implementation.

chatter indefinitely at $\vec{p} = (0.5, 0.5)$, since $\dot{x}_2 = 0$ in $\vec{p}$. This chattering is a valid behavior, though irrelevant for the actually intended proof of $\vec{B}$'s invariance.

We therefore identify intersections of the switching surface with $\dot{x}_2 = 0$ (i.e. solutions to the constraint system $k_2\sqrt{x_1} - k_4\sqrt{x_2} = 0 \wedge x_2 = k_3$) and, finding only this one in $\vec{p}$, add a *don't-care mode* around it—depicted in Figure 4.2 as $D\vec{C}_i = [0.49, 0.51] \times [0.49, 0.51]$. Since this region lies well inside $\vec{B}$, we allow any trajectory that reaches it to jump immediately or after an arbitrary positive amount of time to the outer border of the don't-care mode, illustrated by $D\vec{C}_o$, which is $\varepsilon = 0.0625$ away from $D\vec{C}_i$. We also forbid any trajectory to enter $D\vec{C}_i$. This modification trades in accuracy by introducing non-determinism for the benefit of an artificial hysteresis: trajectories which could formerly stutter in $\vec{p}$ can now jump to any point on the border of $D\vec{C}_o$, but must then move along the system's dynamics again, consuming time.

With this modification, we can prove that $\vec{B}$ is left for less than $\tau = 0.0625$ using unwinding depths $k \geq 5$. The results are shown in Table 4.2. For this instance of the model, the new implementation is not only faster for all unwinding depths, when the result is at least as strong as the result obtained from the old implementation, it is also capable of producing successful proofs, i.e. unsatisfiability results, more often. A potential reason could be that the evaluation of the stored Taylor coefficients using all internal solution representations computed by VNODE-LP can in some cases generate tighter enclosures than the old evaluation scheme, since it does not introduce an additional wrapping of the starting set, which was formerly unavoidable during the re-initialization of the solver.

	all			no bracketing			no direct			no direction		
depth	a	b	a/b	a	b	a/b	a	b	a/b	a	b	a/b
1	c, 149.2	c, 148.2	1.01	c, 4.7	c, 4.7	1.00	c, 94.0	c, 94.4	1.00	c, 146.7	c, 147.6	0.99
2	c, 157.9	c, 112.7	1.40	c, 451.0	c, 136.3	3.31	c, 102.9	c, 100.3	1.03	c, 39.7	c, 69.7	0.57
3	**u**, 147.8	**u**, 108.4	1.36	**u**, 196.9	**u**, 243.2	0.81	**u**, 96.5	**u**, 69.6	1.39	c, 126.0	c, 95.0	1.33
4	**u**, 237.2	**u**, 107.7	2.20	**u**, 706.4	**u**, 239.5	2.95	**u**, 92.4	**u**, 79.2	1.17	c, 196.0	c, 132.2	1.48
5	**u**, 270.3	**u**, 126.7	2.13	**u**, 276.1	**u**, 448.2	0.62	**u**, 125.9	**u**, 92.2	1.37	c, 243.7	c, 185.9	1.31
6	**u**, 227.2	**u**, 142.9	1.59	**u**, 466.7	**u**, 1182.5	0.39	**u**, 227.3	**u**, 112.2	2.03	c, 191.7	c, 217.1	0.88
7	**u**, 254.8	**u**, 160.9	1.58	**u**, 224.7	**u**, 582.0	0.39	**u**, 143.2	**u**, 131.8	1.09	c, 328.6	c, 240.6	1.37
8	**u**, 279.4	**u**, 179.4	1.56	**u**, 406.5	**u**, 696.2	0.58	**u**, 159.6	**u**, 124.5	1.28	c, 434.1	c, 450.2	0.96
9	**u**, 328.2	**u**, 199.4	1.65	**u**, 444.0	**u**, 1509.2	0.29	**u**, 151.5	**u**, 149.1	1.02	c, 1203.0	c, 266.0	4.52
10	**u**, 312.2	**u**, 217.4	1.44	**u**, 430.7	**u**, 627.7	0.69	**u**, 152.2	**u**, 157.3	0.97	c, 807.6	c, 532.4	1.52

Table 4.4: Comparison solver results and CPU times (seconds) using our new implementation on two variants of the containment check in region $\vec{A}$. Column a contains the results for the original model without flow invariants, column b those for the modified version with flow invariants.

	all			no bracketing			no direct			no direction		
depth	a	b	a/b	a	b	a/b	a	b	a/b	a	b	a/b
1	c, 2.6	c, 2.6	0.99	c, 1.2	c, 1.2	0.98	c, 1.6	c, 1.6	0.99	c, 2.6	c, 2.6	1.00
2	c, 10.2	c, 11.3	0.91	c, 5.2	c, 5.7	0.92	c, 6.9	c, 7.6	0.91	c, 8.7	c, 11.7	0.74
3	c, 16.2	c, 23.3	0.70	c, 8.9	c, 10.2	0.88	c, 10.8	c, 14.1	0.76	c, 12.6	c, 15.2	0.83
4	c, 18.0	c, 25.2	0.71	c, 8.4	c, 14.4	0.58	c, 12.2	c, 15.4	0.80	c, 14.4	c, 25.3	0.57
5	**u**, 106.6	**u**, 155.6	0.69	**u**, 62.6	**u**, 93.9	0.67	**u**, 67.7	**u**, 93.3	0.73	c, 15.8	c, 31.9	0.50
6	**u**, 265.5	**u**, 399.8	0.66	**u**, 59.2	**u**, 61.8	0.96	**u**, 146.3	**u**, 162.7	0.90	c, 18.1	c, 37.2	0.49
7	**u**, 171.6	**u**, 405.2	0.42	**u**, 75.5	**u**, 72.9	1.04	**u**, 106.2	**u**, 176.7	0.60	c, 21.2	c, 53.4	0.40
8	**u**, 223.7	**u**, 297.4	0.75	**u**, 74.3	**u**, 147.9	0.50	**u**, 398.4	**u**, 356.0	1.12	c, 21.2	c, 65.7	0.32
9	**u**, 459.4	**u**, 447.7	1.03	**u**, 115.1	**u**, 69.9	1.65	**u**, 181.6	**u**, 257.9	0.70	c, 21.2	c, 55.4	0.38
10	**u**, 308.9	**u**, 999.8	0.31	**u**, 166.3	**u**, 206.9	0.80	**u**, 333.1	**u**, 389.0	0.86	c, 21.8	c, 88.2	0.25

Table 4.5: Comparison solver results and CPU times (seconds) using our new implementation on two variants of the containment check in region $\vec{B}$. Column a contains the results for the original model without flow invariants, column b those for the modified version with flow invariants.

Evaluation on an Unstable Instance. We also applied this proof scheme to the region $\vec{C} = [0.3, 0.4]\times[0.6, 0.7]$ again with unwinding depths 1 to 10. As expected, none of the resulting formulae was proven unsatisfiable. Runtimes were again consistently faster with the new implementation, ranging from 1.6 seconds for unwinding depth 1 without bracketing system usage to 492.9 seconds observed for depth 9, using all methods in combination. Speedups were between 1.22 for depth 9 with all methods and 13.12 for depth 5 with disabled direct VNODE-LP usage. Detailed results are shown in Table 4.3.

Model Instances With Flow Invariants. Using the newly introduced feature of flow invariants, we repeated the checks for containment in region $\vec{A}$ and $\vec{B}$ on a modified version of the model in which the domain bounds and mode invariants for the $x_2 \geq k_3$ and $x_2 \leq k_3$ modes were added. Table 4.4 compares the results of the region $\vec{A}$ containment check for these two different model instances using our current implementation, Table 4.5 shows the same comparison for the containment check in region $\vec{B}$. These results show that adding flow invariants to a model can influence the solving times in both directions, yielding roughly as many speedups as slowdowns between 4-fold increases and 4.5-fold reductions in solving times.

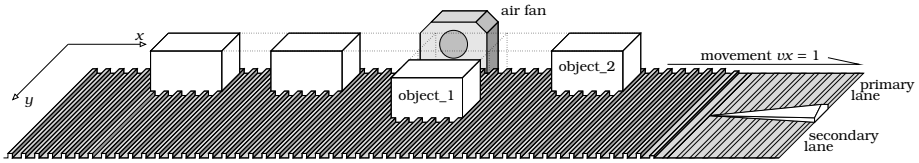


Figure 4.4: Schematic drawing of the conveyor belt system.

4.1.2 A Conveyor Belt System

In order to evaluate iSAT-ODE on a benchmark with a larger discrete and continuous state space and more complex non-linear dynamics, we have modeled a fictitious yet realistic system of a sorting facility in which light packages that are traveling on a conveyor belt can be pushed by an air blast from the primary lane on which they arrive to a secondary lane. Figure 4.4 shows a schematic drawing of this system. Objects arrive from the left $(x, y) = (0, 0)$ and move in positive x-direction with constant velocity $vx = 1$. Centered at position $x = 5$, an air fan can blow air to exert a force on the objects in its vicinity. The force applied to an object at position (x, y) is then given by $F = F_a \cdot e^{-((x-5)^2+y^2)}$, where F_a is the maximum force the air fan can exert. The force distribution over the position is shown in Figure 4.5 for a fixed $F_a = 1.0$ (in the benchmark models, F_a takes unknown but constant values from different ranges).

The controller of the system evaluates a sensor at $x = 0$, the starting position of *object_1*, and subsequently decides to activate the air flow. To capture the uncertainties in the involved measurement, the signal processing latencies, and the actuator reactiveness, we model the switching to occur at an unknown time point during $[T_{earliest\_act}, T_{latest\_act}]$. Similarly, the controller deactivates the air fan at an unknown time point during $[T_{earliest\_deact}, T_{latest\_deact}]$.

While the objects move with constant velocity $vx = 1$ in x-direction due to the mechanical coupling with the grooves in the conveyor belt, their movement in y-direction obeys a slip-stick friction model. When in sticking mode, the object's y-velocity vy is 0 and does not change. Up to a maximum static friction force, all externally applied forces are counteracted by the friction. However, as soon as the external force from the air fan overcomes this maximum friction force, the object starts to slip and the force that counteracts the acceleration caused by the air blast force is governed by kinetic friction which is substantially lower than the static friction. The object goes back to the sticking mode only when its velocity has decreased to zero again.

Flow Invariants. As described earlier, flow invariants in our formalism can only be given by simple upper or lower bounds on variables that are defined by ODEs. In the above slip-stick model, however, there is a more complex invariant on the x and y variables, representing the constraint that the object sticks as long as the force $F = F_a \cdot e^{-((x-5)^2+y^2)}$ does not exceed the maximum static friction force $F_{s\_max} = \mu_s \cdot m \cdot g$, where μ_s is the static friction coefficient, m the object's mass, and g the gravitational constant. The direct flow invariant for the mode *sticking* when the air blast is active (symbolized by *air_blast_on*) is:

$$F_a \cdot e^{-((x-5)^2+y^2)} - \mu_s \cdot m \cdot g \leq 0.$$

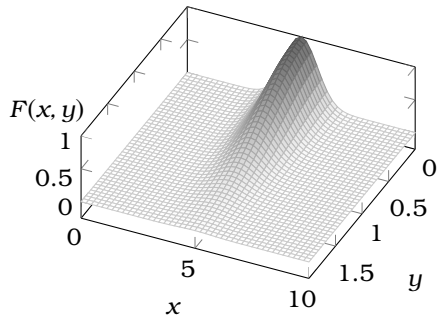


Figure 4.5: Air blast force distribution over (x, y) postion.

In order to formulate this flow invariant in a way compatible with our restriction, we introduce a new variable

$$f := F_a \cdot e^{-((x-5)^2+y^2)} - \mu_s \cdot m \cdot g \leq 0,$$

and simply calculate its derivative with respect to the time t, i.e.

$$\begin{aligned}\dot{f} &= \frac{\partial f}{\partial x} \cdot \overbrace{\dot{x}}^{=1} + \frac{\partial f}{\partial y} \cdot \overbrace{\dot{y}}^{=0} \\ &= F_a \cdot e^{-((x-5)^2+y^2)} \cdot (-2) \cdot (x-5) \cdot \dot{x} \\ &= F_a \cdot e^{-((x-5)^2+y^2)} \cdot (-2x+10).\end{aligned}$$

By adding this expression for $\dot{f}$ as an additional ODE to the system and adding as initialization of $f = F_a \cdot e^{-((x-5)^2+y^2)} - \mu_s \cdot m \cdot g$ when entering the mode, we can add $f(time) \leq 0$ as flow invariant and thus model the complex condition by means of a simple upper bound on a newly introduced variable.

Modeling and encoding. Figure 4.6 shows the complete conveyor belt model as a system of parallel hybrid automata with three components for *object_1*, *object_2*, and the controller *ctrl*. Each object automaton consists of four states to capture the different dynamics depending on whether the object is sticking or slipping and whether the air blast is active or inactive. Jumps occur hence when either the air blast is activated or deactivated by the controller (at the bottom of the figure) or when an object satisfies the condition to leave the sticking or slipping regime. Due to a restriction in iSAT-ODE, the parameters for the objects' masses m_1 and m_2 as well as for the maximum force F_a of the air blast have to be modeled explicitly as dimensions of the ODE system since their exact value is unknown. We therefore also show these explicit dimensions in the automaton. For the friction constants μ_k and μ_s and the gravitational constant g, we have assumed known exact values and therefore do not have to model them by additional dimensions. The initial values for x, F_a, m_1, and m_2 are taken from intervals denoted with X, FA, M_1, and M_2 respectively. Additionally, the controller's behavior depends on choices for $T_{earliest\_act}$, T_{latest_act}, $T_{earliest\_deact}$, and T_{latest_deact}. The instantiation of these intervals allows a significant amount of parameterization, which we exploit when using this model as a benchmark.

The nominal behavior of the system is that after reaching $t = 10$, *object_1* has reached the secondary lane of the conveyor belt, i.e. satisfies $y \geq 1$, while

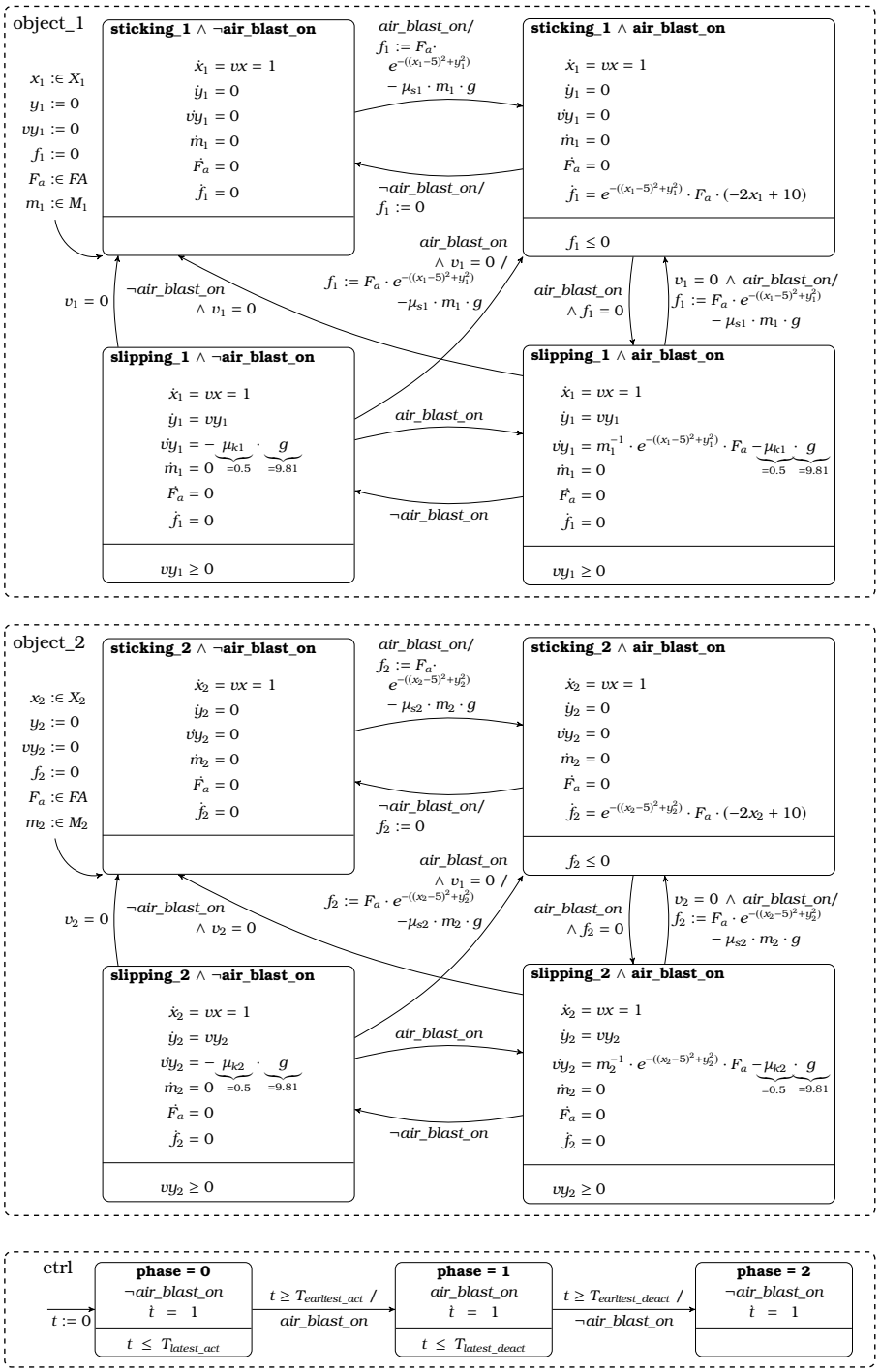


Figure 4.6: Conveyor belt system modeled by parallel automata

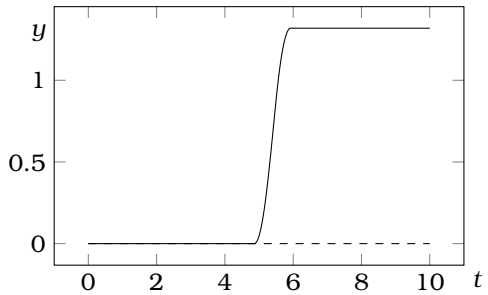


Figure 4.7: Numerically simulated nominal trajectory for the conveyor belt system of 10 seconds length. The dashed line shows the second object's y-position, which does not change, while the first object changes its lane as desired.

object_2 has not been pushed off the primary lane, despite its proximity to the first object. A simulated trajectory that satisfies these properties can be seen in Figure 4.7.

Model Checking. The goal of model checking is to find trajectories which violate this property, i.e. at least one of the objects ends up on the wrong lane.

Using simulation, we have identified ten parameter ranges which we will subsequently analyze by model checking using iSAT-ODE. Table 4.6 gives an overview over these parameters and also shows aggregated simulation results, which give a hint at the expected outcome for model checking using each of the parameter sets.

Benchmark Results. Figures 4.8–4.10 show all results obtained from 120 iSAT-ODE solver runs on a 2.6 GHz AMD Opteron machine (running multiple instances on parallel cores independently) with a memory limit of 8 GiB each and 50000 seconds timeout. In the figures, the following abbreviations have been used: *all* (all ODE enclosure methods active), *no-brsys* (all enclosure methods except the bracketing systems), *no-direct* (all except the direct usage of VNODE-LP), *no-direction* (all except direction deduction); Heuristics: *disc-fst* (split down all discrete variables first), *dyn-rel-width* (split variable whose current range is largest relative to its domain width), and *default-heur* (default heuristic: no sorting, split round robin).

Observations and Evaluation. The data obtained from the simulation runs (see last row of Table 4.6) suggests that sets 01, 03, and 03_wider01 are unsafe, i.e. lead to error trajectories, while sets 02_point to 02_wider06 are safe, i.e. the system does not have error trajectories for parameter choices from these ranges. Consistent with this expectation, iSAT-ODE finds error trajectories in the form of candidate solution boxes for sets 01 and 03_wider01 and successfully proves unsatisfiability for up to the requested limit of 14 unwindings for sets 02_point to 02_wider05 with a varying number of solver settings. For the set 02_wider06, the solver is unable to perform this proof for BMC depths above 6, indicating that the widening of the uncertainty of the controller's phase switching time

	01	02_point	02_wider01	02_wider02	02_wider03	...
X_1	$[-0.5, 0.5]$	$[0, 0]$	$[-0.1, 0.1]$	$[-0.1, 0.1]$	$[-0.1, 0.1]$	
M_1	$[0.08, 0.12]$	$[0.1, 0.1]$	$[0.1, 0.1]$	$[0.09, 0.11]$	$[0.09, 0.11]$	
X_2	$[0.0, 2.5]$	$[1, 1]$	$[1, 1]$	$[1, 1]$	$[1, 1]$	
M_2	$[0.08, 0.12]$	$[0.1, 0.1]$	$[0.1, 0.1]$	$[0.1, 0.1]$	$[0.09, 0.11]$	...
FA	$[0.7, 1.5]$	$[1, 1]$	$[1, 1]$	$[1.2, 1.3]$	$[1.2, 1.3]$	
TA	$[4.0, 4.99]$	$[4.5, 4.5]$	$[4.5, 4.5]$	$[4.5, 4.5]$	$[4.5, 4.5]$	
TD	$[5.00, 6.00]$	$[5.5, 5.5]$	$[5.5, 5.5]$	$[5.5, 5.5]$	$[5.5, 5.5]$	
#	645	0	0	0	0	

02_wider04	02_wider05	02_wider06	03	03_wider01	
$[-0.1, 0.1]$	$[-0.2, 0.2]$	$[-0.2, 0.2]$	$[-0.2, 0.2]$	$[-0.2, 0.2]$	X_1
$[0.09, 0.11]$	$[0.09, 0.11]$	$[0.09, 0.11]$	$[0.09, 0.11]$	$[0.09, 0.11]$	M_1
$[1, 3]$	$[1, 3]$	$[1, 3]$	$[1, 3]$	$[1, 3]$	X_2
... $[0.09, 0.11]$	$[0.09, 0.11]$	$[0.09, 0.11]$	$[0.09, 0.11]$	$[0.09, 0.11]$	M_2
$[1.2, 1.3]$	$[1.2, 1.3]$	$[1.2, 1.3]$	$[1.1, 1.2]$	$[1.0, 1.3]$	FA
$[4.5, 4.5]$	$[4.5, 4.5]$	$[4.4, 4.6]$	$[4.3, 4.7]$	$[4.3, 4.7]$	TA
$[5.5, 5.5]$	$[5.5, 5.5]$	$[5.4, 5.6]$	$[5.3, 5.7]$	$[5.3, 5.7]$	TD
0	0	0	4	75	#

Table 4.6: Parameter sets used for the instantiation of the conveyor belt benchmark. Each column shows the intervals used for the system parameters using the same names as in the automaton in Figure 4.6, except for $TA := [t_{earliest\_act}, t_{latest\_act}]$ and $TD := [t_{earliest\_deact}, t_{latest\_deact}]$. The very last row of the table shows how many out of 1000 simulation runs using randomly-chosen points from the column's parameter set were violating at least one property, i.e. one of the objects ended up on the wrong lane. Based on this estimate, parameters chosen from sets 02_point to 02_wider06 are expected not to cause any error trajectories.

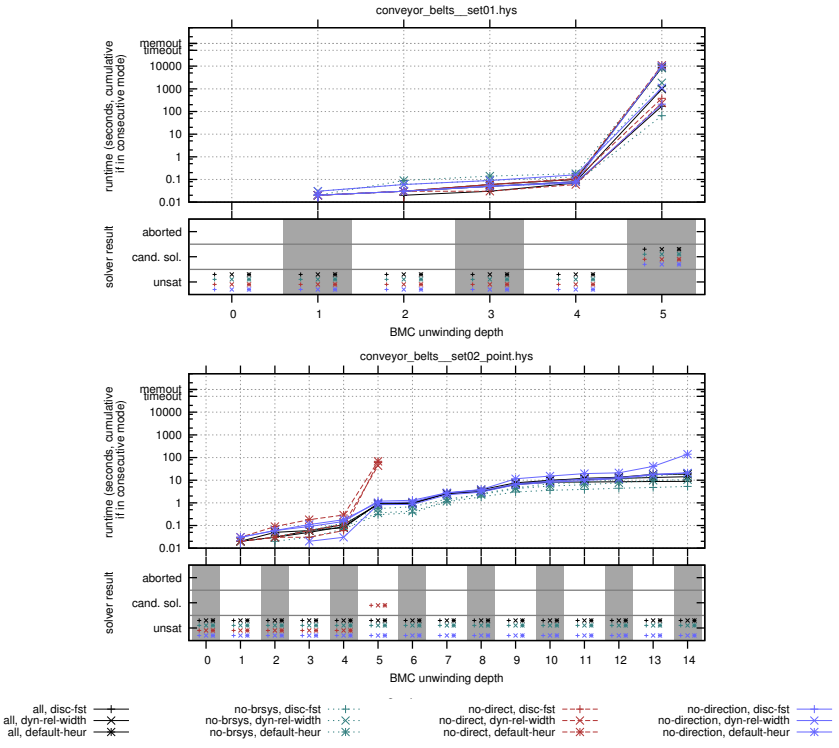


Figure 4.8: Runtimes and results for the conveyor belt benchmark. See text on page 148 for abbreviations.

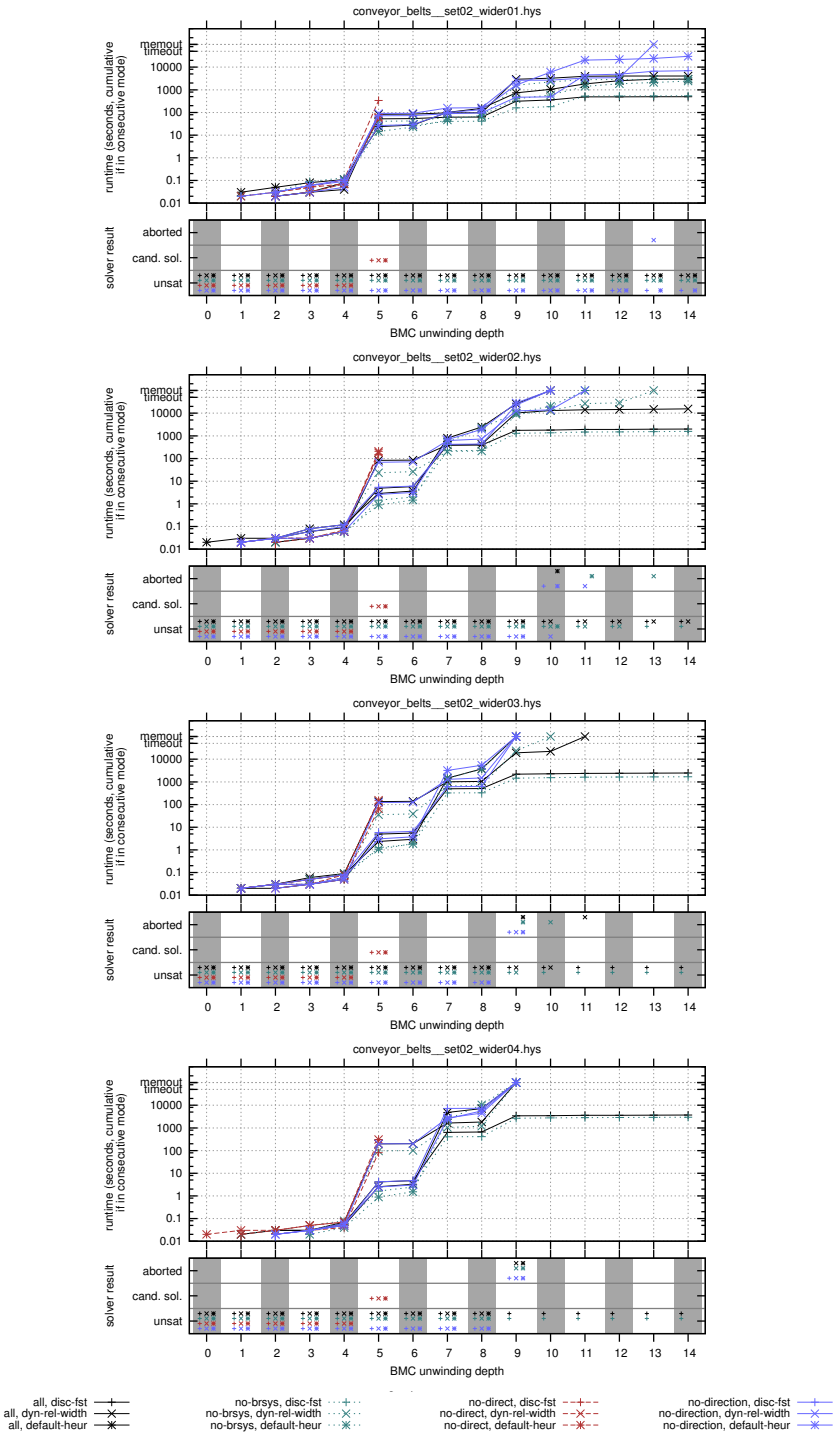


Figure 4.9: Runtimes and results for the conveyor belt benchmark (continued). See text on page 148 for abbreviations.

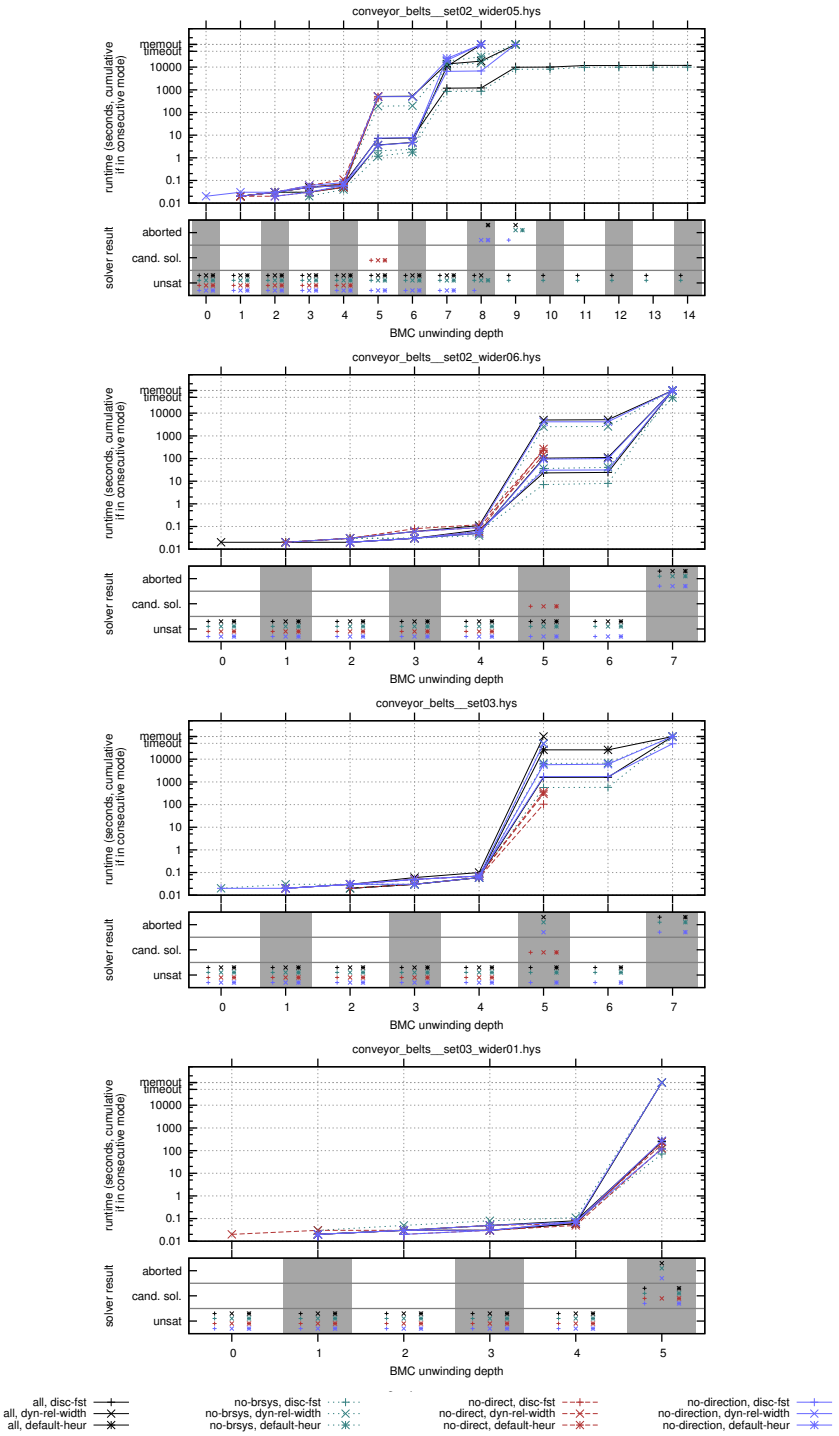


Figure 4.10: Runtimes and results for the conveyor belt benchmark (continued). See text on page 148 for abbreviations.

points makes the problem significantly harder to solve. Similarly, the solver runs into memory or time limits for set03 at depth 7.

The most striking outliers are caused by the disabling of the direct enclosure method. Restricted only to the bracketing system enclosure and the direction deduction, iSAT-ODE terminates with candidate solution boxes for early unwinding depths on all parameter sets from 02_point to 03, always in contradiction to solver runs in which the direct method is not disabled. This clearly indicates that for this benchmark, enclosures obtained from the bracketing system are insufficient to rule out those boxes that are finally reported as candidate solution boxes, whereas the direct enclosure is able to refute these spurious candidate solution boxes.

Equally noticeable is the runtime advantage of the *disc-fst* splitting heuristics on this benchmark over the other two heuristics. As explained in more detail in Subsection 3.3.7, using *disc-fst*, the solver does not split any real-valued variable, as long as there is still a Boolean or integer variable, whose width is above the minimum splitting width. The minimum splitting width was kept at its default of 0.01 for this benchmark, hence the *disc-fst* heuristic means that first all discrete variables are split down to point values (since their ranges can obviously either have a width of above 1 or of exactly 0). The effect of this heuristic is that the solver—earlier than with other heuristics—examines abstract paths of the system in which for each step only one mode or one jump can be active. This seems to guard against some unnecessary search and the costly ODE deductions it causes.

Compatible with the observation that the bracketing systems alone lead to spurious candidate solution boxes is the runtime advantage when the bracketing system is disabled. However, the difference between the *all* and *no-brsys* runtimes is not very large, indicating that the computational cost of generating the bracketing enclosures is not high in this example and that they do not significantly influence the search process either. Without knowing in advance whether or not the bracketing system's enclosures work on a given problem, this benchmark's results suggest that even if they do not contribute enough to the deduction to solve the benchmark successfully on their own, their computational cost is so low in such a case that it is a good default choice to have them activated.

4.1.3 Comparison with hydlogic

In this subsection, we compare our tool with the results published in [IUH11] for the hydlogic tool, which is the technologically most closely related competitive approach, being also based on a Satisfiability modulo ODE scheme and having a VNODE-LP core for handling of non-linear ODEs. In [IUH11], Ishii et al. present several case studies and the results they obtained for them. Where appropriate, they also compare the results to PHAVer and HSolver. Our comparison with the hydlogic results therefore also yields an indirect comparison with these other tools, which we hence do not repeat.

Based on the description as hybrid automata in [IUH11], we have remodeled some of these systems in our predicative encoding for iSAT-ODE. As even small changes in the modeling approach or subtle variations in the encoding can lead to dramatically different results (especially runtimes), we want to emphasize that such a comparison can only give a limited snapshot of the actual relation of the tools.

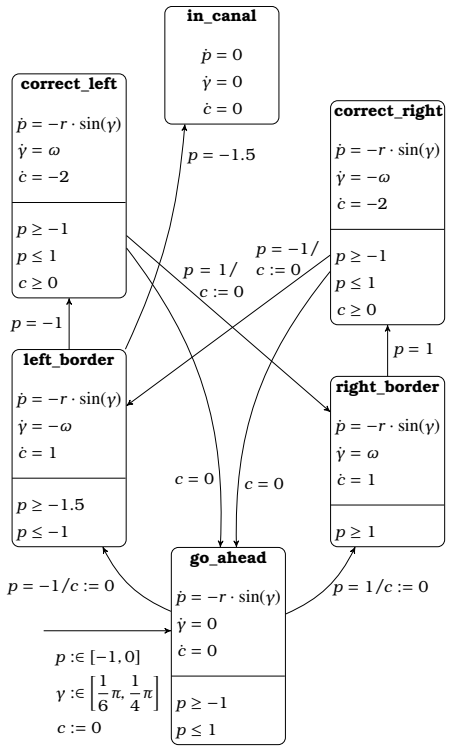


Figure 4.11: Car steering system based on [CFH+03, IUH11].

Car Steering Problem

The first benchmark from [IUH11] is based on a car steering controller originally investigated by [CFH+03]. We depict our version of the automaton for this system in Figure 4.11. The car's movement is modeled by its position p and its heading γ. Its initial position and heading are unknown, but bounded by intervals. When the car reaches one of the borders of the street, its heading is changed continuously and the time measured until the car reaches the border again (now heading inwards). The heading is now changed in the opposite direction for half the time that the car has spent outside of the road's boundaries. The maneuver ends in the unsafe *in_canal* state when the position reaches $p \leq 1.5$.

Modeling Details. The cited versions of this automaton have an additional sink mode *straight_ahead* which is reached from *correct_left* and *correct_right* when the counter reaches zero (before the obverse border is reached). If flows are allowed to take zero time or jumps to immediately follow one another, an instantaneous sequence of mode changes arises which terminates in an inappropriate mode. Under this semantics, traces may proceed immediately into *correct_left* after entering *left_border* with $p = -1$. As the counter remains $c = 0$ under these circumstances, the trace races through to and then stays forever in *straight_ahead*, although the car has actually never changed its direction when crossing the border and will definitely reach $p \leq -1.5$. Since neither of [CFH+03, IUH11] detect this race condition in the model they present, we

have changed the model in two ways: (a) we have collapsed the *straight_ahead* mode with *go_ahead*, such that this zero-time trace would not be able to hide the eventual reaching of the *in_canal* state, and (b) have added a condition that, when entering the modes *left_border* and *right_border*, there must follow a flow, and it must take strictly more than zero time. Note that we found this trace when validating our encoding of the original model with iSAT-ODE and were surprised to find this obviously unintended trajectory which is compatible with this often assumed semantics of hybrid systems (e.g. in [LJS+03]).

For the simple constant ODE components $\dot{p} = 0$, $\dot{\gamma} = \pm\omega$, and $\dot{c} = \{-2, 0, 1\}$, we added the closed-form linear solutions as redundant encodings, since they are easily obtained and may help with deduction. This step could be automated, e.g. as a preprocessing step, but has currently not been implemented in our tool. The bounds for the initial values of the car's heading γ have been overapproximated by representable interval boundaries. For the angular velocity ω, we approximate by $\omega = 0.78539816$ the value $\pi/4$ that is given in [CFH+03]. From there, we also take the value for the radius $r = 2$. In order to reduce unnecessary non-determinism, we also disallow any flows that end before they satisfy a guard condition. This stuttering in one mode is otherwise very costly for the search, since there are infinitely many points to interrupt a flow that all have to be examined, if the system is modeled in a naive way.

Results. In [IUH11], Ishii et al. analyze the following four scenarios. We summarize their results from their paper and compare them with the iSAT-ODE results as shown in detail in Figure 4.14.

For the *steering-1* scenario, which is using the model as described above and in Figure 4.11, hydlogic finds a trajectory leading to *in_canal* in two or three steps and proves its existence. With iSAT-ODE, we also find a trajectory with all tried settings for four BMC unwindings. As can be seen from the trace in Figure 4.12, these four steps amount to a sequence of one flow in *go_ahead*, a mode switch to *left_border*, a flow in that mode, and a final switch to *in_canal*. Figure 4.13 shows that this trace is consistent with a numerically approximated trajectory emerging from the starting values identified by iSAT-ODE. Our result is weaker only in the lack of a proof that the identified trace really exists. The runtime reported in [IUH11] is 5.31s on 2.4 GHz Intel Core 2 Duo processor. Our runtimes on a newer AMD Opteron 2.6 GHz processor are spread out, but the best can be considered competitive with this number.

In the *steering-2* instance of this benchmark, the canal is moved to the left by 0.5 units, such that the guard for entering *in_canal* becomes $p = -2$. We also change the invariant of mode *left_border* to cover this widened range. The hydlogic result for this safe instance of the system is reported in [IUH11] as *unknown* after three steps and 198.30s of runtime. With iSAT-ODE, the graph in the upper right corner of Figure 4.14 shows clearly that for the large majority of settings, we can prove unsatisfiability of the formula up to much larger numbers of unwindings. The solver runtimes diverge depending on the chosen heuristics. While the default heuristic leads to timeouts after depth 17, the other two heuristics allow successful refutation up to unwinding depth 30, before running into the 50000 seconds timeout limit. Again, one notable observation is that disabling the direct method leads to candidate solution boxes as has been observed already in the conveyor belt benchmark.

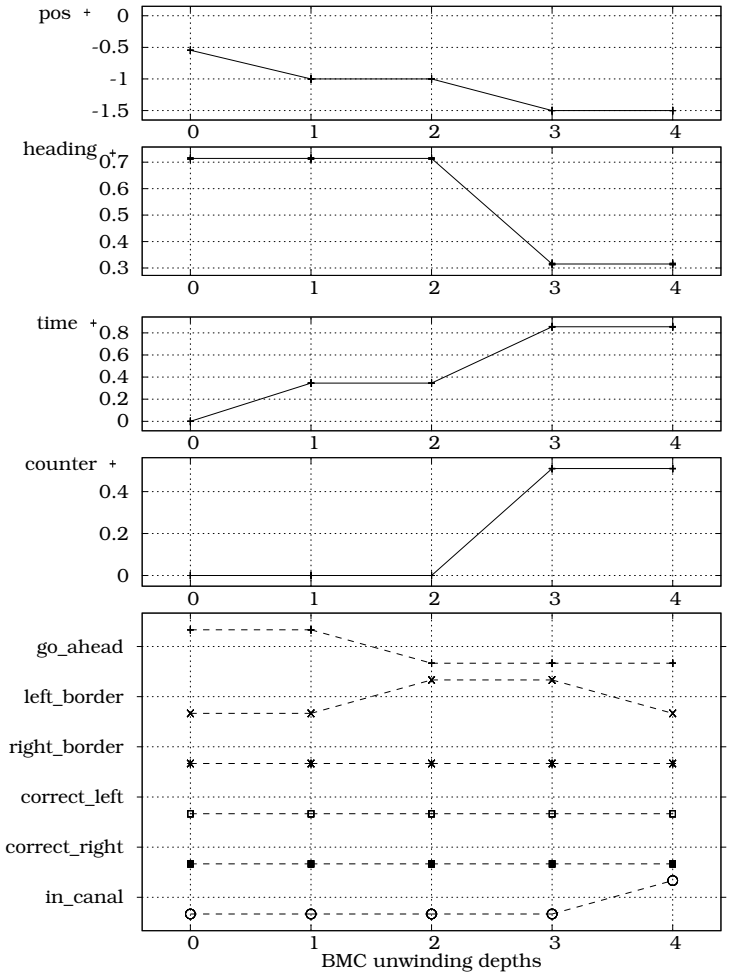


Figure 4.12: An iSAT-ODE trace for the *steering-1* benchmark instance. Value of variables at the BMC unwinding depths.

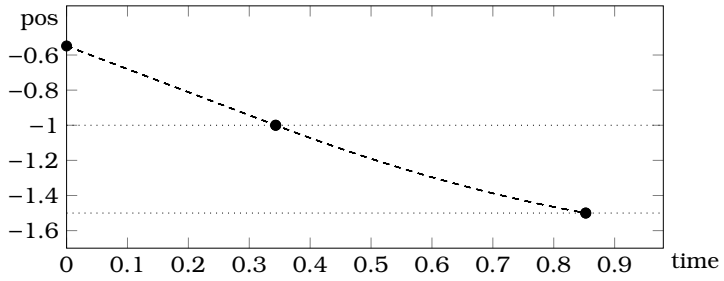


Figure 4.13: Car positions for the *steering-1* benchmark obtained from approximative numerical simulation (dashed line) compared with the candidate solution box (shown by marks, actual box is tighter) from the iSAT-ODE trace.

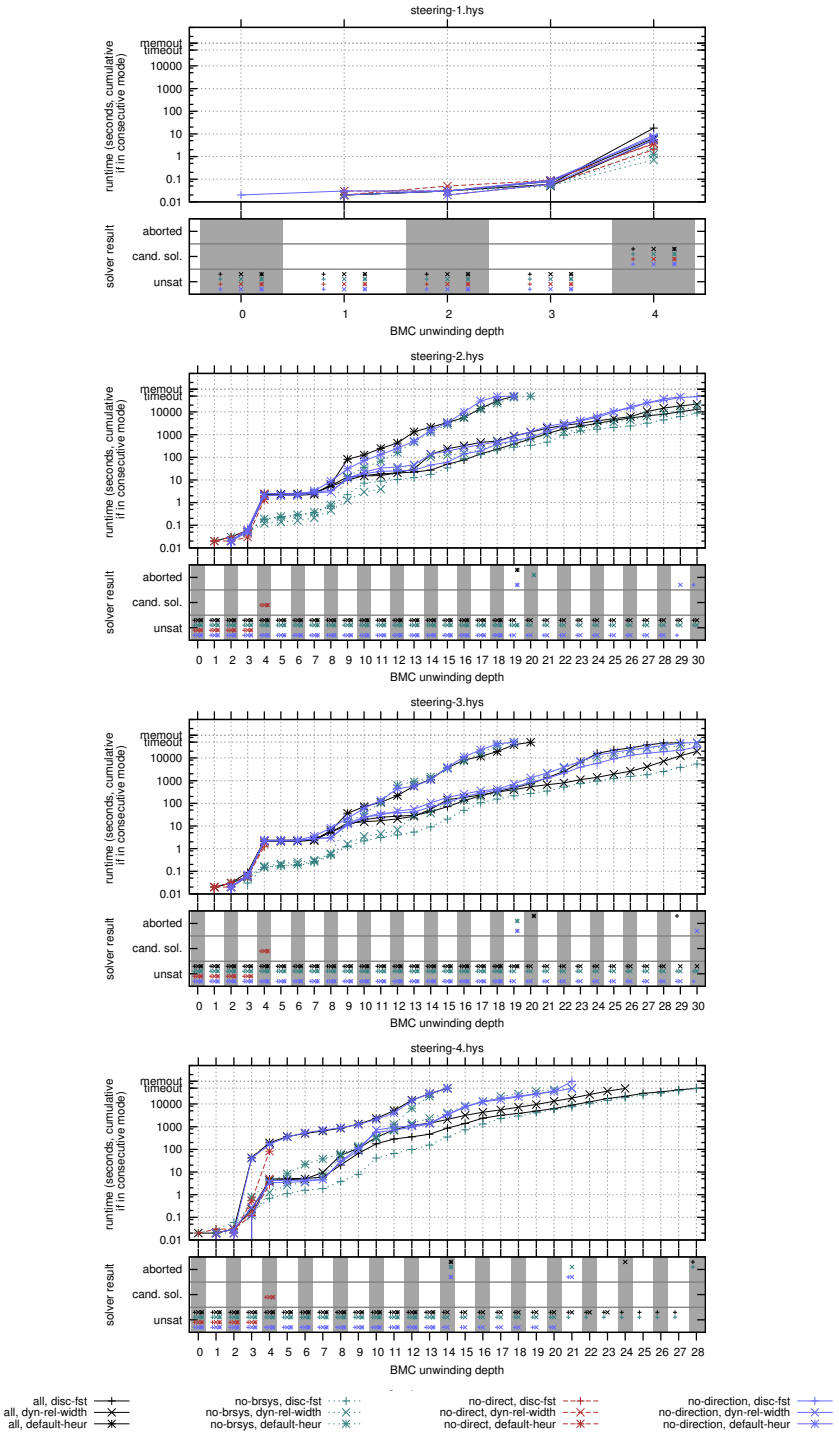


Figure 4.14: Results and runtimes for four instances of the car steering benchmark.

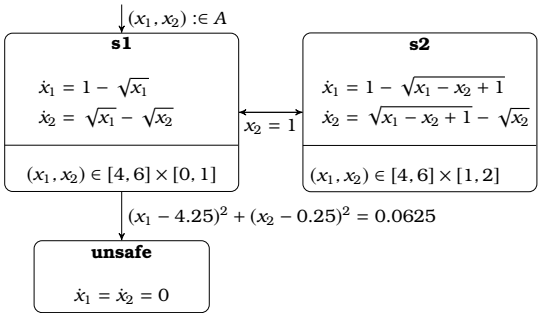


Figure 4.15: Model of the two tank system from [IUH11].

The *steering-3* instance is the same as *steering-2*, except that the initial range for the position is restricted to $p :\in [-0.9, 0]$. This restriction helps hydlogic to prove unsatisfiability for three steps within 22.16s. The results for iSAT-ODE are very similar to the *steering-2* example. If we compare the iSAT-ODE runtimes for six unwindings of the formula with hydlogic's reported runtime for three steps, we consider our runtimes to be competitive even when factoring in the newer CPU architecture.

Finally, *steering-4* is the same as *steering-3*, except that the initial ranges are set to $p :\in [-1, 1]$ and $\gamma :\in [-\pi/4, \pi/4]$. For hydlogic, [IUH11] reports a timeout at 1200 seconds of runtime for checking satisfiability for three steps of the system. The lower right graph in Figure 4.14 shows that iSAT-ODE can prove unsatisfiability up to unwinding depths of 27 with the *disc-fst* and disabled bracketing system based enclosures. Using these settings, iSAT-ODE can e.g. prove unsatisfiability for 12 unwinding depths in less than 100 seconds.

Two Tank System

Structurally the same model as has already been shown in Subsection 4.1.1, the version of the two tank system from [IUH11] uses different parameters (all $k_i = 1$), additional invariants, and different initial ranges. The model checking goal is then classical reachability rather than the stabilization properties we examined in Subsection 4.1.1. The property to be checked by BMC is the reachability of an unsafe mode, which is characterized by a circular region in the (x_1, x_2) state space. For the sake of clarity, we repeat the automaton from [IUH11] in Figure 4.15 and—since the brevity of this model allows it—also show the full encoding of it in the iSAT-ODE language in Figure 4.16.

Modeling Details. Ishii et al. instantiate the model with $\bar{A} = [5.25, 5.75] \times [0, 0.5]$. Due to the $\sqrt{x_2}$ term that occurs in the ODE system, this initial range for x_2 is problematic since it leaves no margin for numerical overapproximation where x_2 grows into the negative range. While there is no report of problems in [IUH11] for hydlogic with this issue, we have observed the VNODE-LP layer in iSAT-ODE running into long sequences of deduction failures caused by the underlying interval library's reports of encountered numerical errors. Without deduction, the solver is obviously free to split down the box in the proximity of $x_2 = 0$, consequently finding spurious traces there, whose refutation would

```
1  DECL
2      float [−10, 10] x1, x2;
3      float [0, 1000] time;
4      float [0, 1000] delta_time;
5      boole s1, s2;
6      boole flow;
7      boole unsafe;
8  INIT
9      time = 0;
10     x1 >= 5.25; x1 <= 5.75;
11     x2 >= 0.01; x2 <= 0.5;
12      s1;
13     !s2;
14     !unsafe;
15     flow;
16 TRANS
17     time' = time + delta_time;
18     s1' + s2' = 1;
19
20     flow and s1 −>
21         (d.x1 / d.time = 1 − nrt(x1, 2));
22     flow and s1 −>
23         (d.x2 / d.time = nrt(x1, 2) − nrt(x2, 2));
24     flow and s1 −> (x1(time) >= 4);
25     flow and s1 −> (x1(time) <= 6);
26     flow and s1 −> (x2(time) >= 0);
27     flow and s1 −> (x2(time) <= 1);
28
29     flow and s2 −> (d.x1 / d.time
30                         = 1 − nrt(x1 − x2 + 1, 2));
31     flow and s2 −> (d.x2 / d.time
32                 = nrt(x1 − x2 + 1, 2) − nrt(x2, 2));
33     flow and s2 −> (x1(time) >= 4);
34     flow and s2 −> (x1(time) <= 6);
35     flow and s2 −> (x2(time) >= 1);
36     flow and s2 −> (x2(time) <= 2);
37     flow −> ((s1 and s1') or (s2 and s2'));
38     flow −> delta_time > 0;
39
40     flow −> (!flow' or unsafe');
41
42     !flow −> x2 = 1.0;
43     !flow −> ((s1 and s2') or (s2 and s1'));
44     !flow −> flow';
45     !flow −> delta_time = 0;
46     !flow −> (x1' = x1 and x2' = x2);
47
48     unsafe' <−>
49         (x1' − 4.25)^2 + (x2' − 0.25)^2 = 0.0625;
50 TARGET
51     s1;
52     unsafe;
```

Figure 4.16: Two tank model encoded in the iSAT-ODE input language.

have required successful generation of ODE enclosures. In order to be able to compare our results, we have opted for a modification of the benchmark such that the initial value of x_2 is positively separated from 0 and have therefore chosen $\vec{A} = [5.25, 5.75] \times [0.01, 0.5]$.

In Figure 4.16, we show the complete encoding of the system. The first part contains the variable declaration, including the ranges for all variables and the special *time* and *delta_time* variables. The next section defines the initial states of the system: starting at time point 0, with any value for $(x_1, x_2) \in \vec{A}$, being in mode *s1* and starting with a continuous flow instead of a jump. The transition predicate contains explicitly the semantic knowledge that time progresses in each step exactly by the amount of the duration *delta_time* and that the system cannot be in both modes at the same time (only the guard condition for entering the *unsafe* mode has been modeled by a predicate in lines 48–49 that can be true independently of the current mode, as long as (x_1, x_2) is in the unsafe region). Lines 20–27 and 29–36 contain the continuous dynamics and flow invariants for modes *s1* and *s2* respectively. As noted earlier, the *nrt* symbol stands for the n-th root. Line 37 makes it explicit that flows do not change the current mode and line 38 requires that flows actually have positive duration (which is a design choice that is suitable for this model). Line 40 requests that after a flow either a jump occurs or the unsafe state is reached—enforcing that flows are not interrupted without having reached a guard condition. Line 42 encodes the guard condition: if the mode is changed (line 43), the guard $x_2 = 1$ must hold. The remainder of the section encodes that there are no two jumps following directly after each other (again a design choice for this model) and that jumps do not take time and do not change the continuous variables (since the automaton has no actions). In lines 50–51, the property to be checked is, whether the system can reach the guard condition for entering the mode *unsafe* while being in state *s1*.

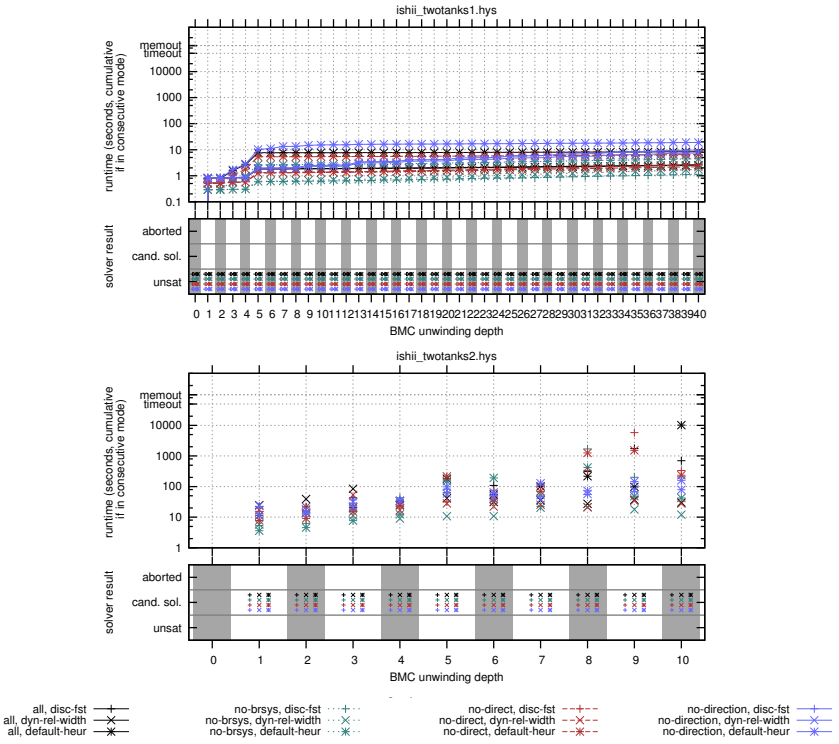


Figure 4.17: Results and runtimes for the instances of the two tank benchmark as parameterized in [IUH11].

Results. For the *twotanks-1* instance, which is exactly as shown in Figure 4.15 and Figure 4.16, hydlogic (again all results quoted from [IUH11]) reports unsatisfiability for two steps within 33.49 seconds. The upper part of Figure 4.17 shows that all iSAT-ODE instances are capable of proving unsatisfiability for up to 40 unwindings of the formula. The runtimes for checking the 40 unwindings consecutively are spread between 1.25 and 20.5 seconds. At least the fastest settings can therefore be considered to be significantly faster than hydlogic even when taking into account the differences in the CPUs on which the benchmarking was performed.

In the *twotanks-2* instance, the region of unsafety is moved such that it becomes reachable. The new guard condition for entering the mode *unsafe* is $(x_1 - 4.5)^2 + (x_2 - 0.75)^2 = 0.0625$. For this instance, hydlogic finds a trajectory of two steps length and proves its existence within 36.34 seconds. As has been detailed earlier, in our model the target property is to find a valuation that satisfies the entrance guard for the *unsafe* mode while in *s1*. Therefore, the solution trajectory in our model is reached already within one step (omitting the final jump to *unsafe*). The trace consists of just a direct flow from a state admissible in the initial condition and following the dynamics of *s1* to the unsafety region. As can be seen from the lower part of Figure 4.17, iSAT-ODE finds this one-step trajectory within just a few seconds. The shortest runtime result for the depth 1 unwinding is 3.63s with the default heuristic and disabled

bracketing, the longest runtime for depth 1 is 24.56s with the *dyn-rel-width* heuristic and all ODE enclosure methods enabled.

Again, the iSAT-ODE result is weaker than the result from hydlogic since there is no guarantee that the identified candidate solution box contains a solution. Note that in this special case where the solution consists of only one flow, using just the ODE enclosure and showing that all points from its prebox satisfy the initial condition and all points from the last enclosure lie within the unsafe region, would yield an equally strong proof.

To do a more extensive analysis of solver runtimes, we forced the solver to check unwinding depths individually instead of using the consecutive mode results that can be seen e.g. in the left part of Figure 4.17. The first observation is that runtimes for larger unwindings spread out significantly. These instances become harder to solve since the solver needs to find a trajectory that reaches the unsafe region, but still has more than the one flow step that is actually required. As can be seen in line 40 of Figure 4.16, our model requires alternating jumps and flows except when the unsafe predicate $(x_1 - 4.5)^2 + (x_2 - 0.75)^2 = 0.0625$ is true. Since this is satisfied already by the endpoint of the trajectory after the first step, the solver needs to find a valuation for the remaining variable instances (e.g. 9 remaining steps for 10 unwindings) such that this predicate still holds or find an alternating sequence of jumps and flows to satisfy line 40.

A solution to this is based on exploiting the overapproximation that occurs in the interval-based ODE enclosures. Although the constraint in line 38 enforces that flows have a duration strictly larger than 0, even the tightest enclosure will still contain the starting points of that flow (which was the argument needed to motivate the direction deduction presented in Section 3.3.8). Those solution traces for larger unwindings that we investigated further therefore lead directly to the satisfied unsafe predicate in the first step and thereafter contained steps of very short duration, e.g. $delta\_time \in (0, 0.00011517)$, which is strictly greater than zero, but still small enough such that the equality constraint for the unsafe predicate was still satisfied due to the enclosure still containing the original prebox. In a way, these results could hence be considered spurious and a stronger form of the direction deduction might have been able to rule out satisfiability for larger unwindings of the formula unless there also exist paths that really perform some alternation between the two modes before reaching the unsafe region.

Bouncing Ball on Sine-Waved Surface

The last example from [IUH11] that we use for our comparison is the model of a ball that bounces off from a sine-wave surface, called *bouncing3* in that paper.

Modeling Details. Like the classical bouncing ball hybrid automaton, this behavior can be modeled by one mode and a self-loop that is triggered when the height of the ball reaches the ground, which in this case is not flat, but satisfies the constraint $p_y = \sin(p_x)$. Ishii et al. use hydlogic to "simulate" the system for ten steps and "assumed that the ball bounces at the earliest crossing point between the ball and the ground". While this refers probably to an algorithmic assumption to search and use the first intersection of the ODE enclosure with a guard condition and pruning after all parts of the enclosure are past the guard, we think that this can be formulated explicitly inside the automaton. From our

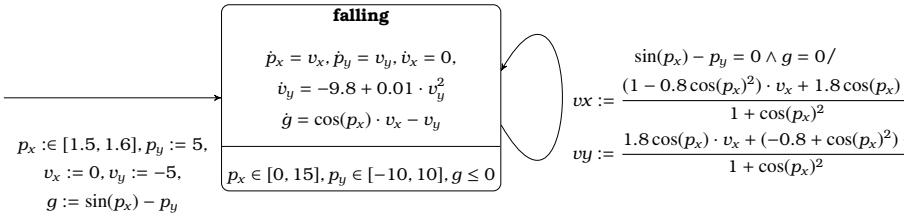


Figure 4.18: Our version of the hybrid automaton for the bouncing ball on a sine-waved surface extending the original from [IUH11] by an extra dimension g that captures the flow invariant.

perspective this means that there should be a flow invariant,

$$p_y \geq \sin(p_x) \Leftrightarrow \underbrace{\sin(p_x) - p_y}_{=:g} \leq 0$$

such that the ball cannot reach a point below the sine curve. Using the same modeling trick that we have detailed earlier, we add a new flow invariant $g \leq 0$ and the ODE $\dot{g} = \cos(p_x) \cdot v_x - v_y$, which is the derivative of the original flow invariant with respect to time. Additionally, the value of g must be initialized correctly to $g := \sin(p_x) - p_y$.

Results. For the "simulation", [IUH11] reports that hydlogic proves unsatisfiability for unwinding depth 10 within 29.15s. The meaning of the unsatisfiability result is not stated explicitly (there is no target condition, so we assume that it is simply *true*), but it could indicate that the given domain is left and therefore no trajectory of that length exists within the specified bounds. The longest trajectory that we could find with iSAT-ODE was for unwinding depth 15 and took 19060 seconds of CPU time, using the *disc-fst* heuristic with disabled bracketing systems. We were able to validate that all intermediate elements of this candidate solution box contained points on the $p_y = \sin(p_x)$ surface.

Assuming that in the hydlogic results, again, one "step" consists of a flow and a jump, we would need to solve unwinding depth 20 to analyze the same 10 steps instance for which hydlogic could report unsatisfiability. For this unwinding depth, iSAT-ODE was not able to find a candidate solution or prove unsatisfiability within 50000 seconds. We therefore consider iSAT-ODE to be clearly slower on this benchmark.

4.2 Range Estimation for Hybrid Systems

In this section, we present an application of iSAT-ODE to the problem of identifying internal system states from measurements. Given a system model and measurements, we try to derive ranges for variables that are not directly observable. Such *set membership estimation (SME)* is relevant e.g. for refining

bounds on uncertain system parameters or for fault diagnosis when the internal mode change from nominal to faulty behavior is not readily "announced" by the system, but must be extracted from the measured data collected by sensors. In practice, data-mining and statistical techniques can be applied to achieve these goals even in an on-line fashion on industrial-scale problems (see e.g. [Ban12]). Our focus lies on demonstrating the applicability of Satisfiability modulo ODE solving to this problem in order to get *guaranteed* results, which in this general class of systems are otherwise hard to obtain. It is also clear from our results, that significant progress with respect to scalability is necessary for the method to be of any practical relevance in this application. The text and graphics presented in this section are taken from [ERNF12b], copyrighted by IFAC Zürich, and reprinted here in a modified version under the permission granted by the copyright transfer statement. Also see this publication for more contextual information and related work regarding the successful application of guaranteed SME techniques for various classes of continuous and hybrid systems, all of which we omit here, since our focus lies solely on demonstrating this application of iSAT-ODE.

Encoding of SME Problems and Tool Application. The predicative encoding of hybrid systems is very flexible, since it allows the integration of arbitrary additional constraints into the formula, and thereby, to manipulate the space of solution trajectories. In the case of SME problems, such additional constraints may stem from measurements and their known error margins. Adding them to the formula, any solution that is incompatible with these measurements is discarded, leaving only those that represent the ranges compatible with the observations. Together with the algorithmic guarantees that an unsatisfiability result is only generated, if no satisfying valuation exists, iSAT-ODE can be used to identify and prune off ranges of infeasible assignments to system parameters and state vectors. The result either is refined knowledge about the parameter or state vector values that can explain the measurements, or—in case of unsatisfiability—a proof that measurements, the model, or e.g. the assumptions made about the error intervals are inconsistent with each other.

Technically, we extend the iSAT-ODE solver slightly to first use only deduction without splitting. It thereby removes parts of the variables' domains that are definitely inconsistent—leaving a range in which solutions might be found. As a second step, it searches a candidate solution box inside the remaining box with splitting reactivated. This and all further candidate solution boxes form a set of possible solutions. While it is not known whether actual trajectories exist within this box, it is clear that iSAT-ODE cannot disprove their existence in this part of the domain. The goal is then to move the borders of the space known not to contain any solutions and the border of the set of candidate solutions towards each other, such that the range estimate becomes more precise. In the third phase, the solver therefore performs a bisection search for each of the variable bounds for which a range estimate is requested: while the distance between the bound of the remaining variable range and the outer bound of the observed candidate solutions is still larger than the specified threshold, the variable is restricted to values between these two bounds, and a candidate solution is searched. In case of unsatisfiability, this range can be excluded, since it is known not to contain any solution. Otherwise, the enclosure of the encountered

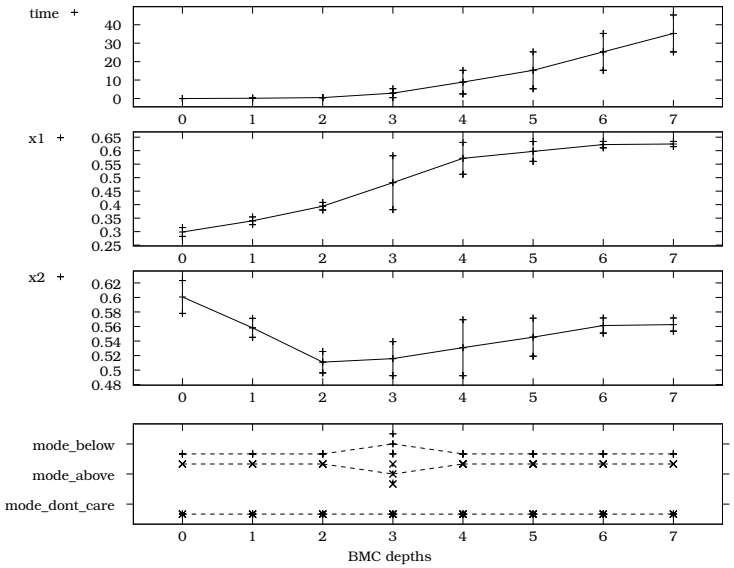


Figure 4.19: Range estimate for the two tank system with seven unwindings (BMC depths) of the transition relation.

candidate solutions is extended to cover this new box as well.

Measuring at Discrete Time Instants. Again, we use the two tank system from [SKHP97] as illustrated in Figure 4.1 with the parameters $k_1 = 0.75$, $k_2 = 1$, $k_3 = 0.5$, and $k_4 = 1$ and reuse our encoding, including the introduction of the don't-care mode.

Our first experiment with this system is based on three measurements of the variables x_1 and x_2, at time $t = 0.16$, $t = 0.48$, and $t = 5.28$, which are assumed uncertain, e.g. $x_1(0.16) \in [0.341 - \varepsilon, 0.341 + \varepsilon]$, $x_2(0.16) \in [0.557 - \varepsilon, 0.557 + \varepsilon]$. The data were obtained from a simulated trajectory emerging from $x_1(0) = 0.3, x_2(0) = 0.6$. We modify the transition relation of the model in such a way that these measurements are interleaved with the normal transitions of the system, i.e., a transition can not only be a continuous flow following the dynamics of the current mode for some duration or a discrete mode switch, but can also be the "consumption" of a measurement event. Additionally, we enforce that a transition must have its maximum possible duration. That is, each step must end with reaching a predefined final point of time after the last measurement, or with reaching the switching surface from where a jump into the other mode becomes possible, or with reaching the point of time for the next measurement event. This modification is only aimed at ruling out solution traces that contain additional interruptions without a reason.

The first goal is to calculate a number of unwindings of the transition relation, such that all measurements have been consumed, and all possible mode switches have been performed. We can now easily obtain this information by adding a target predicate that is true, when one of the measurements has not yet been consumed, or the final time is not yet reached. If this formula is satisfiable for a given unwinding depth, we know that such a trajectory still

```
step = 2 -> (     time' >= 0.30 - EPS_T -- EPS_T is 0.01
                  and time' <= 0.30 + EPS_T
                  and x2' = 0.5
                  and (mode_below' <-> mode_above));
step = 3 -> time' = 0.40 ;
```

Figure 4.20: Part of the extended transition relation: encoding of a level crossing detection event leading to a mode switch and additional observation step.

exists and increase the number of transition steps until the formula becomes unsatisfiable. For our model with the three measurements, iSAT-ODE can first prove unsatisfiability for depth 7.

Removing the target predicate and instead adding the target that all measurements have been consumed, the model can be used to tighten the bounds of the initial values $x_1(0), x_2(0)$, which are only constrained to lie within the region where the system dynamics is defined. We report the range estimate for the variable instances of x_1, x_2, and the time variable in Figure 4.19, along with the range estimates for the discrete mode in the bottom chart of the figure. Note that for step 3, the reconstructed mode is ambiguous: uncertainty makes feasible both modes *mode_below* and *mode_above*. This result must be interpreted in the following way: each trajectory of the system that is compatible with the measurements must pass through each of the boxes for steps 0 to 7. Especially, all trajectories must start in the initial box which has been determined to be $x_1(0) \in [0.285, 0.315], x_2(0) \in [0.578, 0.623]$ and consequently start in *mode_above*. This range estimate safely encloses the starting point of the simulated trajectory from which the measurements were obtained.

Measuring at a Height Threshold. In this experiment, instead of considering arbitrarily chosen measurements, we assume to have one sensor in each tank that measures when a certain height is crossed. Following the idea presented by [Kou03] for a similar model of two tanks, placing one of these sensors at the level of switching, i.e. in our case observing when $x_2(t)$ crosses k_3, allows direct observation of the time of mode changes. We can simplify our approach significantly, since most of the complexity in the previous experiment was caused by the necessity to determine an a-priori unknown number of possible mode switches, and the consequently required number of transition steps.

Starting from a basic encoding of the two tank system, we add a designated counter variable that is incremented in each step of the transition relation. Using this counter, we can iterate through the list of measurements and mode switches, directly enforcing constraints on the time to be reached in this step. While we assume the sensor to give an exact signal of whether the height level has been crossed, we allow for a small temporal uncertainty, i.e. do not enforce measurements to happen at a point of time, but instead within an interval around the assumed time of the observed level crossing. Figure 4.20 exemplifies the encoding of such an event and also shows that the a-priori knowledge of when events happen gives us the freedom to add additional interruptions of the flow at points of time, for which we know that no event occurs. When performing the range estimation, these additional steps allow better observation of the system state's evolution.

We illustrate the range estimate in Figure 4.21, where we use the valuation

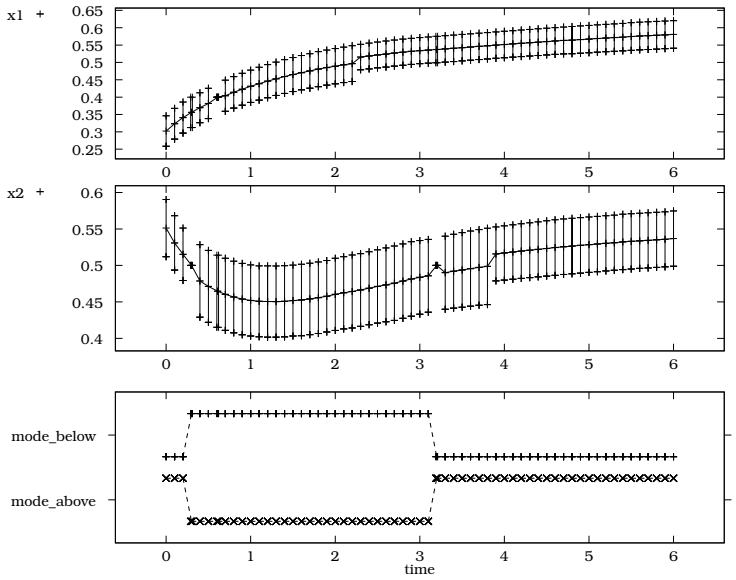


Figure 4.21: Range estimate for the measurements at discrete levels depicted over the valuation of the time variable.

of the time variable as x-axis. Each of the boxes drawn in this graph must be crossed by each trajectory satisfying all constraints. By having additional observation steps, where the time variable is forced to particular points, we now obtain actual enclosures: from the range estimate, we know that at time $t = 0.4$, the values for x_1 and x_2 must lie within the corresponding range estimate. Finally, we have also reconstructed the switching sequence.

Runtimes. Checking unsatisfiability up to unwinding depth 7 in the first experiment has taken iSAT-ODE approximately 86 minutes on an AMD Opteron 8378, 2.4 GHz processor running 64 bit Linux using a minimum splitting width of 0.01 and absolute bound progress threshold of 0.001. The range estimate for the 48 variable instances (six variables in eight instances) was completed within 5.7 hours using the same settings. The results reported in Figure 4.21, were obtained using a minimum splitting width of 0.0625 and absolute bound progress threshold of 0.001 within 7.3 hours on the same machine. Using 0.1 as a coarser setting for the minimum splitting width did not improve the runtime, whereas using a minimum splitting width of 0.01, we stopped the solver after 27 hours.

Chapter 5

Conclusions

We started with the observation that hybrid systems are everywhere, in tea cups and ice crystals, machines, or neurons. We should add to this list now: in bouncing balls and water tanks and on conveyor belts with dubious air-blowing packet sorters. What this thesis and the underlying research tried to address was the question how one can describe such a diverse zoo of systems in a formal way that lends itself to direct verification via satisfiability solving. And more centrally, how routines for satisfiability solving can be extended to handle the resulting formula class. Finally, the question that needs to be asked is: how well do we solve this problem with our approach?

5.1 Summary

First, we introduced hybrid automata from the literature, highlighting their ability to cover the relevant interactions of discrete and continuous behavior that we would like to analyze. We pointed out that they are one of many ways to describe hybrid systems, and that we see them as an input with concise and well-defined semantics to a manual translation process whose output are Satisfiability modulo ODE formulae. *These* form the actual basis of our approach, and their introduction in [EFH08] constitutes the first relevant scientific contribution presented in this thesis.

There was little doubt that a question, whose solution—or even incremental steps towards it—would have such diverse applications, would have attracted numerous researchers, shaping a giant research landscape, which we could only map to a limited degree. We tried to point out how our approach, that tries to find solutions to BMC formulae, differs from most of the other approaches, that aim at computing the entire reachable state space of the hybrid system. We focused more closely on related work from the constraint-solving and satisfiability domain and pointed out the differences, often in the form of limitations made to the allowed continuous dynamics. For the remaining competitor approaches, which try to address the same or very similar questions, we analyzed the differences, pointing out our reliance on the iSAT core and its ability to handle interval valuations directly instead of only a boolean abstraction as is the case in traditional SMT integration schemes. The closest competitor, the hydlogic tool, supports Interval Newton to prove the existence of solutions, while our tool

supports bracketing systems to overcome some of the limitations of VNODE-LP on ODE problems involving non-linear right-hand sides and large initial domains.

Having laid these foundations in formal underground and related work, we looked into the two major algorithmic ingredients of our approach: the iSAT core, for solving boolean combinations of arithmetic constraints over discrete and continuous variables, and VNODE-LP, for computing validated enclosures of sets of initial value problems of ODEs. We pointed out that little needs to be done to use VNODE-LP as a propagator for ODE constraints, but that quite a number of algorithmic enhancements are required when the propagator is expected to be more precise than VNODE-LP's a-priori enclosures and when one wants to avoid throwing away all its costly computations on every conflict-induced backjump performed by the iSAT core. These technical details, together with the tight integration of the bracketing approach, form the major scientific contribution presented in this thesis and published in [ERNF11, ERNF12a].

In the same papers, we published most of the experiments shown in the previous chapter. These bring us to the question we asked at the beginning of these conclusions: how well do we solve the problem?

From the perspective of a thesis writer, the answer is clear: we have demonstrated the applicability of the approach on established benchmarks from the literature and on a larger case study containing a challenging combination of non-linear ODEs and discrete switching. We have shown that our tool outperforms the closest competitor tool on some of their own benchmarks, yet is weaker on others, a mixed result that can be expected, given the technical complexity and the large amount of parameters that influence search behavior, deduction precision etc. which makes it very hard to directly compare tools and to deduce whether one approach is superior. We have also shown the applicability on the important question of set membership estimation, which is more than can be expected for a general-purpose solver not at all developed for that application in mind.

From a scientific and from a practical perspective, the answer may commence in the same way, but it should put these results into context. Even the most complex examples which we have analyzed are barely of industrial proportions. Most of the solver times are very far away from this approach being easily integrated into a development process, in which a system designer would not appreciate having to wait many hours for an analysis of whether the system contains an error trace that a few thousand simulation runs over randomized initial values have a good chance of exposing within seconds. These issues look merely like questions of scalability, but scalability is a giant challenge when search spaces grow exponentially and solvers—even with all clever learning and heuristics—have a very hard time giving timely answers.

On the other hand, the simplest base case, SAT solving, has long become a useful tool in industrial practice despite all its known algorithms having worst-case exponential complexity.

This thesis has shown that formulae containing arithmetic constraints and differential equations can be solved automatically, the solver's results can be interpreted as safety certificates or as error traces. This is good news. It is substantiated and not diminished by the fact that other researchers attempt to go on similar routes. The first results published by Gao et al. in [GKC13] are promising future gains in performance, and their and many others' attempts

at adding notions of robustness and decidability may lead to future algorithms that may exploit realistic restrictions on the formula class.

How well does our approach solve the problem? Well enough for this thesis, but there is a lot of work to be done until this becomes as useful as a compiler that complains about an uninitialized variable, whose value may feed into a control algorithm, whose output may lead to a malfunction, whose consequences may be fatal—at least for the developer's next coffee break.

5.2 Topics for Future Work and Open Questions

At this very point in his thesis, Christian Herde wrote: "It seems to be an invariant truth of solver development that with every solver finished, many ideas on how to do even better pop up. What just appeared to be the ultimate engine, turns out to be the precursor of a presumably much more powerful algorithm only." [Her11]. The invariable truth about invariants is that they tend to hold. While we do not want to propose any fully new algorithms, among the many notes that piled up over the years, a few open questions and ideas still feel as intriguing as they were when written down, some still hold the promise that they may substantially improve the approach presented here.

Hybrid Systems with Unknown Inputs. Currently, inputs in ODE constraints are only supported with some significant restrictions, e.g. they can easily be modeled to have an unknown value that does not change during a flow or—by allowing interruptions of flows—to change their value only finitely often. While flexible, the latter case comes with an increased unwinding depth of the transition system in the BMC formula, which has a significant impact on the size of the search space. On the other hand, by adding e.g. $\dot{x} = y$ and $\dot{y} = 0$, with x being the actual input and y being its first derivative, one can also model systems whose input changes, but only with constant slope. The slope must stay constant during a flow, but can be set arbitrarily between flows, as can be the value of the input itself if desired. This addition of intermediate derivatives can be continued further and eventually leads to covering the significant class of *smooth inputs*.

However, inputs are often used in modeling to allow an abstraction of physical entities that are not easily described or whose description would add too much complexity to the model. Under these circumstances, inputs are unfortunately sometimes only known to stem from a bounded interval—without any knowledge as to their slope or even smoothness. Since the ODE enclosures we use are based on Taylor series, not knowing the derivatives up to the order of the expansion forbids us to compute any enclosure at all.

Future work could address this issue by introducing *mixed-order ODE enclosures*. In these, the solution is expanded to the desired order only for those variables for which the derivatives can be computed, while for other variables, especially inputs, only a (much) lower expansion order is used—limited by the known interval bounds on the value of the input and potentially by known bounds on its derivatives. The goal of this approach would be to avoid having to evaluate any unknown derivatives of an input variable, which can only be bounded by $(-\infty, +\infty)$. A challenging engineering task would then be to

incorporate such a mixed-order series within a competitive enclosure algorithm, such that issues like wrapping and error accumulation can be avoided.

Generalized Bracketing Systems. While inputs in the formalism would extend the expressiveness of our approach, the next potential task for future work could be expected to contribute to the solver's ability to provide solutions in case of non-linear ODEs.

If in a Jacobian matrix most entries have uniquely determined signs, but even a single entry does not, our current implementation of the bracketing approach fails. We think that a scheme, in which some dimensions are kept as non-point intervals, might make the bracketing approach applicable also in these cases. While there is currently only the choice between using bracketing for all variables or for none at all, such a generalized scheme might allow to freely select the dimensions for which bracketing should be attempted. Some theoretical work would be necessary to reduce the number of dimensions present in the bracketing system, i.e. to avoid having to use the upper and lower bracketing variables and the original ones, which would lead to thrice the original dimensionality. Such generalized bracketing systems might then also be helpful in supporting uncertain inputs if the system depends on them in a monotone way.

Use of Taylor Models. Our integration of bracketing systems was motivated by the observation that large initial sets and non-linear ODEs may lead to weak ODE deductions when VNODE-LP alone is not capable of computing any tight enclosures. For the same purpose, we initially considered the use of Taylor models, which we discussed as related work in Section 2.4, but we were discouraged by concerns about their scalability. With the recent advent of the Flow* tool [CÁS13], it may now be time to reconsider the integration of Taylor Models as an additional enclosure mechanism to benefit from their ability to provide tighter enclosures for non-linear ODEs.

More Flexible Deduction Resolution. The use of Taylor models or bracketing systems may improve the tightness of enclosures and hence lead to stronger deduction. If the additional cost of computing them is outweighed by the amount of splitting that they make unnecessary, this may result in an acceleration of the solver. We have, however, also argued that the opposite may be true in some circumstances: computing an enclosure with high precision at high computational cost may be wasted effort when the search space is still very large and the precision of the enclosures is not needed yet. While this topic remains one of heuristics, for a more thorough evaluation of this idea, one should first try to fully exploit the entire available range of precision parameters.

Chiefly among the unused VNODE-LP parameters are those for series expansion order and absolute / relative tolerances. Using them together with the step size parameter, VNODE-LP could potentially be brought to produce rather coarse enclosures with large step sizes and very few evaluations. If the resolutions used in the optimization loops that try to tighten these boxes are chosen accordingly, this may allow a much coarser and faster deduction generation—fully controllable by the resolution parameters chosen by the solver. While the integration of these parameters into the resolution scheme is comparably simple, a lot of work

may be required to find the right balance between the many parameters and maybe even to implement mechanisms that adapt them automatically when e.g. enclosures of little precision are detected to fail too often and hence contribute only to the solver's runtime but not to finding a solution.

Next-Generation iSAT Solver. Parallel to the development of iSAT-ODE, a new generation of the iSAT solver, iSAT-3, has been developed in AVACS that outperforms the implementation of iSAT used in iSAT-ODE in many cases and that integrates stronger deduction mechanisms for linear and polynomial constraints. Overcoming the usual technical difficulties of integrating two pieces of software, one could couple the ODE solving layer with iSAT-3 to benefit from core deductions that do not solely rely on ICP and unit propagation. Just this integration alone may therefore result in a significant improvement in scalability.

Once integrated, the combined tool could exploit the better handling of linear and polynomial constraints by learning ODE deductions that are not merely boxes, or simple comparison constraints in the case of the direction deduction. Already with the current iSAT, it would be possible to generate constraints of higher order, bound their error terms by the enclosure, and add them to the constraint system as overapproximations of the trajectories such that the solver core can use them without resorting to computations in the ODE layer. With polynomial reasoning directly available, this combination may however be significantly more powerful, which may compensate for the more complex generation of these learned facts.

Protecting Coordinate Transformations. When ODE deductions are stored in clauses, they are represented as bounds on the variables of the system and do not contain any information about the coordinate transformations that were necessary during the computation of the underlying enclosures. Any interruption of a flow therefore results in additional wrapping. When the duration of a step is not yet known very tightly, it is not clear, which of the many involved coordinate systems to store. However, when the solver is able to reduce the width of the duration interval such that it becomes tight enough to be covered by only one of the enclosure steps computed by VNODE-LP, it can be beneficial for the next step's enclosure to be restarted with the corresponding coordinate transformation matrices.

However, the interruption of flows rarely occurs without reason. If the continuous evolution is only interrupted e.g. because a parallel component needs to perform a jump, the enclosure with which the preceding step ends could be reused directly. But whenever a jump occurs, its guard and action predicate may actually rule out some part of the valuation, leading to all the manipulations of enclosure representations that complicate reachable state computations for hybrid systems. Whether it is worthwhile to include these algorithms in full, is unclear, but it may improve enclosure tightness already to address those cases where enclosures and their transformation matrices stay valid and could be copied from one flow's end to another flow's beginning.

Non-Determinism, Simulation, and Benchmark Hardness. The simplest metric with which to assess a model's degree of hardness, is to count its states. For infinite systems this is obviously not so simple after all, but even for them,

a finest resolution of interest and bounded domains can be used to give an approximative finite number of boxes that need to be explored in the worst case. This metric is employed surprisingly often to describe systems by phrases like "n modes and m continuous variables", but it is easy to see that it is not very accurate in describing how hard it is to find out whether one of the system's trajectories satisfies a given condition.

For example in digital circuit verification, the inputs and initial states of memory-holding elements are sufficient to determine the outputs of the circuit. An encoding as a SAT formula will comprise a large number of intermediate variables representing connecting lines, but the high level of determinism in such systems makes large regions of this search space unreachable—and some of the actual strength of a state-of-the-art SAT solver over a mere brute-force algorithm is to detect these deterministic relationships by deduction and generalization of conflicts and therefore to avoid trying out the majority of all the possible combinations of variable valuations.

A model may hence very well have a vast state space when using the state space of all variables as a metric, but it may be very easy to analyze if there are only very few trajectories.

Looking at the problem from a forward perspective, the number of trajectories of a model is e.g. determined by its degree of non-determinism including the size of the initial domains. Assuming, we could enumerate all k-step trajectories of a model, proving that none of them satisfies a given condition would mean to check all of them. It is the benefit e.g. of the interval-based approach presented in this thesis that it needs not necessarily have to investigate these trajectories individually. The likelihood and therefore hardness of finding a satisfying trajectory then depends on the ratio of satisfying to unsatisfying trajectories and their distribution in the state space.

What fascinates and to a degree puzzles us, are the implications of this observation. First, it seems useful to find good metrics for the degree of non-determinism inherent to a benchmark, since it may be important for getting a much better category describing their hardness. Secondly, it supports the motivation for making strong deduction capabilities available—since they reduce the amount of non-determinism that the solver adds to a problem as a form of self-inflicted increase in the search space that needs to be covered by branching.

Finally, however, we think that this observation defines the niche of model checking and that it contains some bad news for solver developers. If a formula is quite deterministic, it lends itself well to simulation, since simulation thrives on computing one successor value for each variable and in these cases does not have to struggle with branches in the behavior. If, on top of that, the model has many trajectories leading to target states, randomized simulations will additionally have a good chance of finding one of them. Benchmarks with high degree of determinism and many solutions are therefore those that practitioners will most likely solve by brute-force simulations already today. What they leave behind is this: models with large degrees of non-determinism and an unknown, but likely small number of probably isolated solutions that they could not find.

We think, these models are the hardest of all, since they do not only require strong deduction powers to be able to rule out large parts of the search space, but also provide the largest challenge to splitting heuristics. When there are at most a few isolated solutions, it is simply not likely to accidentally split into the right region of the search space. Many failed branches in the search tree may

be the result before a solution is finally found or enough conflicts are amassed to prove unsatisfiability.

On a level playing field, where not all the easiest benchmarks have been solved by simulation, however, it would very much make sense to tightly couple these tools: let deductions remove infeasible parts, simulate random trajectories in the remaining space, assess by how much they violate the constraints, and thereby guide splitting heuristics. Since this combination has been successful in other domains, it is likely to work here as well.

A Verifiably Correct Solver. Apart from improving scalability, expressiveness and deductive strength, there is the aspect of human error, which must be taken into account when discussing the correctness of our approach. Mathematical proofs can be flawed, special cases be overlooked in algorithm design, implementation errors be missed. Our implementation has been tested on a number of benchmarks. We have instrumented the code with many assertions that check consistency of intermediate results with our expectations. We have added many debugging outputs to be able to validate individual steps, intermediate computations, and the interchange of information between the different solver layers. The iSAT algorithm has been described theoretically and its correctness been verified in manual proof. The use of ODE enclosures, if understood as an additional pruning operator, quite seamlessly fits into this framework. The computation of enclosures via VNODE-LP has been implemented using literate-programming to allow manual inspection of the actual code while reading the mathematical approach. *Still*, there are a lot of possibilities for making errors and not detecting them.

In a most recent paper [Imm14], an approach is presented to build a verified algorithm for ODE enclosures inside the framework of a theorem prover—allowing mechanized proofs of correctness and extraction of the verified code. One could argue that not going this way is taking quite a shortcut, and the work of this thesis definitely is guilty as charged in that respect. We cannot guarantee freedom from errors, which may be capable of invalidating some or even all of our results. We think, however, that the high additional cost involved in building truly verified algorithms (with a complete chain of trust from mechanized mathematical proofs down to verified compilation and execution platforms with error detection) can only ever be gone when it is clear that the goal is truly worthwhile and can be achieved.

Understanding this thesis as prototypical work, which may suffer from overlooked incorrectness, it may be seen as an indication that there is a potential benefit from joining these technologies together—and maybe eventually to also invest the effort into doing it based on the highest level of achievable correctness then available, once better understanding of the right way to the solution is gained.

List of Figures

2.1 Parallel hybrid automata for running example 22
2.2 Trace and visualization 28
2.3 Flow invariants and urgency 30
2.4 Input of iSAT-ODE and candidate solution 41
2.5 Encoding of the two cars model 42

3.1 Interval pruning for $x = y \cdot z$ 56
3.2 Deduction chain with two constraints 58
3.3 Box consistent valuations 60
3.4 Consistent valuation capturing empty solution set 61
3.5 Abstract representation of the iSAT algorithm 71
3.6 Rewriting into internal iSAT format and search space 73
3.7 iSAT example: step 0 74
3.8 iSAT example: step 1 74
3.9 iSAT example: step 2 75
3.10 iSAT example: step 3 75
3.11 iSAT example: step 4 76
3.12 iSAT example: step 5 76
3.13 Approximative numerical integration 79
3.14 Rudimentary ODE enclosure 84
3.15 Coordinate transformation 88
3.16 Harmonic oscillator as VNODE-LP program 92
3.17 VNODE-LP enclosures of the harmonic oscillator 93
3.18 High-level view of iSAT-ODE structure 95
3.19 iSAT-ODE input and solver state 100
3.20 Deduction for ODE constraint 103
3.21 Detection of leaving the flow invariant region 109
3.22 Conservative bounds over entire duration 111
3.23 Refinement of enclosures 114
3.24 Forward and backward ODE deductions 117
3.25 Spurious solution without direction deduction 125
3.26 Corner trajectories do not lead to enclosures 130
3.27 Bracketing system versus direct method 135

4.1 Structure of the two tank system 138
4.2 Two tank system: simulated trajectories 139
4.3 Two tank system: runtimes for checking unreachability 140
4.4 Schematic drawing of the conveyor belt system 145

4.5 Conveyor belt system: air blast distribution 146
4.6 Conveyor belt system modeled by parallel automata 147
4.7 Conveyor belt system: simulated nominal trajectory 148
4.8 Conveyor belt system: results and runtimes 149
4.9 Conveyor belt system: results and runtimes (continued) 150
4.10 Conveyor belt system: results and runtimes (continued) 151
4.11 Car steering system 153
4.12 Car steering system: iSAT-ODE trace 155
4.13 Car steering system: simulation and candidate solution 155
4.14 Car steering system: results and runtimes 156
4.15 Two tank system: automaton 157
4.16 Two tank system: iSAT-ODE encoding 158
4.17 Two tank system: results and runtimes 159
4.18 Bouncing ball on sine-waved surface 161
4.19 Range estimate for two tank system 163
4.20 Two tank system: encoding of a level-crossing detection 164
4.21 Range estimate for measurements at discrete levels 165

Bibliography

[ABCS05] Gilles Audemard, Marco Bozzano, Alessandro Cimatti, and Roberto Sebastiani. Verifying industrial hybrid systems with MathSAT. *Electronic Notes in Theoretical Computer Science*, 119(2):17 – 32, Elsevier, 2005. doi:10.1016/j.entcs.2004.12.022. (on page 32).

[ADG03] Eugene Asarin, Thao Dang, and Antoine Girard. Reachability analysis of nonlinear systems using conservative approximation. In Oded Maler and Amir Pnueli, editors, *Hybrid Systems: Computation and Control*, volume 2623 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2003. doi:10.1007/3-540-36580-X_5. (on page 44).

[ADLBZB10] R[oberto] Armellin, P[ierluigi] Di Lizia, F[ranco] Bernelli-Zazzera, and M[artin] Berz. Asteroid close encounters characterization using differential algebra: the case of Apophis. *Celestial Mechanics and Dynamical Astronomy*, 107(4):451–470, Springer Netherlands, 2010. doi:10.1007/s10569-010-9283-5. (on page 48).

[Ban12] Patrick Bangert. *Optimization for Industrial Problems*. Springer, 2012. doi:10.1007/978-3-642-24974-7. (on page 162).

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. doi:10.1007/3-540-49059-0_14. (on page 32).

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, ACM, 1975. doi:10.1145/361002.361007. (on page 128).

[BG06] Frédéric Benhamou and Laurent Granvilliers. Continuous and interval constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, pages 571–603. Elsevier, Amsterdam, Netherlands, 1st edition, 2006. (on pages 56, 57, 59, 60).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, Massachusetts, London, England, 2008. (on pages 24, 31).

[BM89] Pavel Bochev and Svetoslav Markov. A self-validating numerical method for the matrix exponential. *Computing*, 43(1):59–72, Springer, 1989. doi:10.1007/BF02243806. (on page 44).

[BM98] Martin Berz and Kyoko Makino. Verified integration of odes and flows using differential algebraic methods on high-order taylor models. *Reliable Computing*, 4(4):361–369, Kluwer Academic Publishers, 1998. doi:10.1023/A:1024467732637. (on page 48).

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-825. (on page 38).

[BT00] Oleg Botchkarev and Stavros Tripakis. Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In Nancy Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2000. doi:10.1007/3-540-46430-1_10. (on page 44).

[CÁS13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 258–263. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-39799-8_18. (on pages 48, 170).

[CESS08] Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sörensson. Sat-solving in practice. In *9th International Workshop on Discrete Event Systems, 2008. WODES 2008*, pages 61–67, 2008. doi:10.1109/wodes.2008.4605923. (on page 37).

[CFH+03] Edmund M. Clarke, Ansgar Fehnker, Zhi Han, Bruce H. Krogh, Olaf Stursberg, and Michael Theobald. Verification of hybrid systems based on counterexample-guided abstraction refinement. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2003. doi:10.1007/3-540-36577-X_14. (on pages 153, 154).

[Dan06] Thao Dang. Approximate reachability computation for polynomial systems. In João Hespanha and Ashish Tiwari, editors, *Hybrid Systems: Computation and Control*, volume 3927 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2006. doi:10.1007/11730637_13. (on page 44).

[DLGM09] Thao Dang, Colas Le Guernic, and Oded Maler. Computing reachable states for nonlinear biological models. In Pierpaolo Degano and Roberto Gorrieri, editors, *Computational Methods in Systems*

Biology, volume 5688 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2009. doi:10.1007/978-3-642-03845-7_9. (on page 44).

[DT12] Thao Dang and Romain Testylier. Reachability analysis for polynomial dynamical systems using the Bernstein expansion. *Reliable Computing*, 2(17):128–152, 2012. Available from: http://interval.louisiana.edu/reliable-computing-journal/volume-17/reliable-computing-17-pp-128-152.pdf. (on page 44).

[EFH08] Andreas Eggers, Martin Fränzle, and Christian Herde. SAT modulo ODE: A direct SAT approach to hybrid systems. In Sungdeok (Steve) Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA'08)*, volume 5311 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2008. doi:10.1007/978-3-540-88387-6_14. (on pages 16, 32, 37, 94, 104, 118, 167).

[Egg06] Andreas Eggers. Einbettung sicherer numerischer Integration von Differentialgleichungen in DPLL-basiertes arithmetisches Constraint-Solving für hybride Systeme. Master's thesis, Carl von Ossietzky Universität Oldenburg, Department für Informatik, 2006. Advisors: Martin Fränzle and Christian Herde. Available from: http://www.avacs.org/Publikationen/Open/eggers.msc06.pdf. (on pages 16, 85, 118).

[ERNF11] Andreas Eggers, Nacim Ramdani, Nedialko S. Nedialkov, and Martin Fränzle. Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Proceedings of the Ninth International Conference on Software Engineering and Formal Methods (SEFM)*, volume 7041 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2011. doi:10.1007/978-3-642-24690-6_13. (on pages 16, 126, 133, 137, 138, 139, 142, 143, 168).

[ERNF12a] Andreas Eggers, Nacim Ramdani, Nedialko S. Nedialkov, and Martin Fränzle. Improving the SAT modulo ODE approach to hybrid systems analysis by combining different enclosure methods. *Software and Systems Modeling*, Springer, 2012. doi:10.1007/s10270-012-0295-3. (on pages 17, 32, 37, 39, 40, 41, 94, 104, 107, 118, 137, 138, 168).

[ERNF12b] Andreas Eggers, Nacim Ramdani, Nedialko S. Nedialkov, and Martin Fränzle. Set-membership estimation of hybrid systems via sat modulo ode. In Michel Kinnaert, editor, *Prooceedings of the 16th IFAC Symposium on System Identification*, pages 440–445. International Federation of Automatic Control (IFAC), 2012. doi:10.3182/20120711-3-BE-2027.00292. (on page 162).

[FH05] Martin Fränzle and Christian Herde. Efficient proof engines for bounded model checking of hybrid systems. In *Proceedings of*

the Ninth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2004), volume 133 of *Electronic Notes in Theoretical Computer Science*, pages 119–137, 2005. doi:10.1016/j.entcs.2004.08.061. (on page 32).

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), ACM, June 2007. GNU MPFR is available from http://www.mpfr.org. doi:10.1145/1236463.1236468. (on page 57).

[FHR+07] Martin Fränzle, Christian Herde, Stefan Ratschan, Tobias Schubert, and Tino Teige. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT Special Issue on Constraint Programming and SAT*, 1:209–236, 2007. Available from: http://jsat.ewi.tudelft.nl/content/volume1/JSAT1_11_Fraenzle.pdf. (on pages 16, 37, 38, 52, 55, 120).

[FLGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011. doi:10.1007/978-3-642-22110-1_30. (on page 45).

[Frä99] Martin Fränzle. Analysis of hybrid systems: An ounce of realism can save an infinity of states. In Jörg Flum and Mario Rodriguez-Artalejo, editors, *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 126–139. Springer, 1999. doi:10.1007/3-540-48168-0_10. (on page 38).

[Fre05] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005. doi:10.1007/978-3-540-31954-2_17. (on page 45).

[FRZ11] Peter Franek, Stefan Ratschan, and Piotr Zgliczynski. Satisfiability of systems of equations of real analytic functions is quasi-decidable. In Filip Murlak and Piotr Sankowski, editors, *MFCS 2011: 36th International Symposium on Mathematical Foundations of Computer Science*, volume 6907 of *Lecture Notes in Computer Science*, pages 315–326. Springer, 2011. doi:10.1007/978-3-642-22993-0_30. (on page 38).

[Gir05] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2005. doi:10.1007/978-3-540-31954-2_19. (on page 44).

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. Satisfiability modulo ODEs. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2013. ISBN 978-0-9835678-3-7/13. Available from: http://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD13/papers/25-SAT-Modulo-ODEs.pdf. (on pages 16, 38, 53, 168).

[GMEH10] Alexandre Goldsztejn, Olivier Mullier, Damien Eveillard, and Hiroshi Hosobe. Including ordinary differential equations based constraints in the standard CP framework. In David Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2010. doi:10.1007/978-3-642-15396-9_20. (on pages 40, 50, 51, 53).

[Gol09] Alexandre Goldsztejn. On the exponentiation of interval matrices. Computing Research Repository (arXiv), 2009. Available from: http://arxiv.org/abs/0908.3954. (on page 44).

[GP07] Antoine Girard and George J. Pappas. Approximation metrics for discrete and continuous systems. *IEEE Transactions on Automatic Control*, 52(5):782–798, 2007. doi:10.1109/TAC.2007.895849. (on page 38).

[GPB05] Nicoló Giorgetti, George J. Pappas, and Alberto Bemporad. Bounded model checking of hybrid dynamical systems. In *44th IEEE Conference on Decision and Control, 2005 and 2005 European Control Conference (CDC-ECC '05)*, pages 672–677, 2005. doi:10.1109/CDC.2005.1582233. (on page 32).

[GvVK94] J[an] F[riso] Groote, S[ebastiaan] F.M. van Vlijmen, and J. W[ilco] C. Koorn. The safety guaranteeing system at station Hoorn-Kersenboogerd. Technical Report 121, Department of Philosophy, Utrecht University, Logic Group Preprint Series, October 1994. Extended abstract published at COMPASS'95 conference, cf. DOI. Available from: http://dspace.library.uu.nl/bitstream/handle/1874/26459/preprint121.pdf, doi:10.1109/CMPASS.1995.521887. (on page 32).

[HEFT08] Christian Herde, Andreas Eggers, Martin Fränzle, and Tino Teige. Analysis of Hybrid Systems using HySAT. In *The Third International Conference on Systems (ICONS 2008)*, pages 196–201. IEEE Computer Society, 2008. doi:10.1109/ICONS.2008.17. (on page 32).

[Heh84] Eric C. R. Hehner. Predicative programming, part II. *Communications of the ACM*, 27(2):144–151, 1984. doi:10.1145/69610.357990. (on page 37).

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Logic in Computer Science (LICS), Symposium on*, pages 278–292, Los Alamitos, CA, USA, 1996. IEEE Computer Society. doi:10.1109/lics.1996.561342. (on pages 19, 20, 22, 23, 30).

[Her11] Christian Herde. *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure - Proof Engines for the Analysis of Hybrid Discrete-Continuous Systems*. Vieweg+Teubner Research, 1st edition, 2011. doi:10.1007/978-3-8348-9949-1. (on pages 32, 37, 39, 44, 55, 59, 65, 67, 69, 71, 169).

[HHMWT00] Thomas A. Henzinger, Benjamin Horowitz, Rupak Majumdar, and Howard Wong-Toi. Beyond HyTech: Hybrid systems analysis using interval numerical methods. In Nancy Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2000. doi:10.1007/3-540-46430-1_14. (on pages 47, 138).

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, Springer-Verlag, 1997. doi:10.1007/s100090050008. (on page 43).

[HHWT98] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):540–554, 1998. doi:10.1109/9.664156. (on page 45).

[Hic00] Timothy J. Hickey. Analytic constraint solving and interval arithmetic. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 338–351, New York, NY, USA, 2000. ACM. doi:10.1145/325694.325738. (on page 50).

[HK06] Zhi Han and Bruce H. Krogh. Reachability analysis of large-scale affine systems using low-dimensional polytopes. In João Hespanha and Ashish Tiwari, editors, *Hybrid Systems: Computation and Control*, volume 3927 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 2006. doi:10.1007/11730637_23. (on page 44).

[HPWT01] Thomas A. Henzinger, Jörg Preussig, and Howard Wong-Toi. Some lessons from the HyTech experience. In *Proceedings of the 40th IEEE Conference on Decision and Control*, volume 3, pages 2887–2892, 2001. doi:10.1109/.2001.980714. (on page 43).

[HW04] Timothy J. Hickey and David K. Wittenberg. Rigorous modeling of hybrid systems using interval arithmetic constraints. In Rajeev Alur and GeorgeJ. Pappas, editors, *Hybrid Systems: Computation and Control*, volume 2993 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2004. doi:10.1007/978-3-540-24743-2_27. (on pages 49, 50).

[Imm14] Fabian Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In *Proceedings of the 6th NASA Formal Methods Symposium*, volume 8430 of *Lecture Notes in Computer Science*. Springer, 2014. to appear. Available from: http://home.in.tum.de/~immler/documents/immler2014enclosures.pdf. (on page 173).

[IUH11] Daisuke Ishii, Kazunori Ueda, and Hiroshi Hosobe. An interval-based SAT modulo ODE solver for model checking nonlinear hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–13, Springer, March 2011. doi:10.1007/s10009-011-0193-y. (on pages 16, 52, 53, 152, 153, 154, 157, 159, 160, 161).

[IUHG09] Daisuke Ishii, Kazunori Ueda, Hiroshi Hosobe, and Alexandre Goldsztejn. Interval-based solving of hybrid constraint systems. In Alessandro Giua, Cristian Mahulea, Manuel Silva, and Janan Zaytoon, editors, *Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems*, volume 3, pages 144–149. International Federation of Automatic Control (IFAC), 2009. doi:10.3182/20090916-3-ES-3003.00026. (on pages 51, 52, 53).

[KGG+09] Stefan Kowalewski, Mauro Garavello, Hervé Guéguen, Gerlind Herberich, Rom Langerak, Benedetto Piccoli, Jan Willem Polderman, and Carsten Weise. Hybrid automata. In Jan Lunze and Françoise Lamnabhi-Lagarrigue, editors, *Handbook of Hybrid Systems Control*, chapter 3. Cambridge University Press, Cambridge, UK, 2009. (on page 31).

[Kou03] Xenofon D. Koutsoukos. Estimation of hybrid systems using discrete sensors. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, volume 1, pages 155 – 160, December 2003. doi:10.1109/cdc.2003.1272552. (on page 164).

[LJS+03] J. Lygeros, K.H. Johansson, S.N. Simic, Jun Zhang, and S.S. Sastry. Dynamical properties of hybrid automata. *Automatic Control, IEEE Transactions on*, 48(1):2–17, jan 2003. doi:10.1109/TAC.2002.806650. (on page 154).

[Loh88] Rudolf Lohner. *Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben und Anwendungen*. PhD thesis, Universität Karlsruhe, Fakultät für Mathematik, June 1988. (on pages 16, 47, 48, 81, 82, 85, 90).

[LTG+06] Michael Lerch, German Tischler, Jürgen Wolff von Gudenberg, Werner Hofschuster, and Walter Krämer. Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw.*, 32(2):299–324, ACM, June 2006. FILIB++ is available at http://www2.math.uni-wuppertal.de/~xsc/software/filib.html. doi:10.1145/1141885.1141893. (on pages 57, 94).

[Mar11] Peter Marwedel. *Embedded System Design - Emdedded Systems Foundations of Cyber-Physical Systems*. Springer, 2nd edition, 2011. doi:10.1007/978-94-007-0257-8. (on page 43).

[MB03] Kyoko Makino and Martin Berz. Suppression of the wrapping effect by taylor model- based validated integrators. Technical Report MSU Report MSUHEP 40910, Michigan State University, 2003. Available from: http://bt.pa.msu.edu/pub/papers/VIRC03/VIRC03.pdf. (on page 48).

[MCRTM14] Moussa Maïga, Christophe Combastel, Nacim Ramdani, and Louise Travé-Massuyès. Nonlinear hybrid reachability using set integration and zonotopic enclosures. In *Proceedings of the 13th European Control Conference (ECC)*, pages 234–239, 2014. doi:10.1109/ECC.2014.6862491. (on page 88).

[Moo66] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1966. (on pages 47, 57, 80, 81, 84, 85).

[Ned99] Nedialko Stoyanov Nedialkov. *Computing Rigorous Bounds on the Solution of an Initial Value Problem for an Ordinary Differential Equation*. PhD thesis, Department of Computer Science, University of Toronto, Ontario, Canada, 1999. Available from: https://tspace.library.utoronto.ca/bitstream/1807/13081/1/NQ41256.pdf. (on pages 92, 121).

[Ned06] Nedialko S. Nedialkov. VNODE-LP – a validated solver for initial value problems in ordinary differential equations. Technical Report CAS-06-06-NN, Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, 2006. Available from: http://www.cas.mcmaster.ca/~nedialk/vnodelp/doc/vnode.pdf. (on pages 15, 47, 92, 94).

[Neu03] Arnold Neumaier. Taylor forms—use and limits. *Reliable Computing*, 9(1):43–79, Kluwer Academic Publishers, 2003. doi:10.1023/A:1023061927787. (on pages 48, 62).

[NJ01] Nedialko S. Nedialkov and Kenneth R. Jackson. A new perspective on the wrapping effect in interval methods for initial value problems for ordinary differential equations. In Ulrich Kulisch, Rudolf Lohner, and Axel Facius, editors, *Perspectives on Enclosure Methods*, pages 219–264. Springer, Vienna, 2001. doi:10.1007/978-3-7091-6282-8_13. (on pages 85, 90).

[NJP01] Nedialko S. Nedialkov, Kenneth R. Jackson, and John D. Pryce. An effective high-order interval method for validating existence and uniqueness of the solution of an ivp for an ode. *Reliable Computing*, 7(6):449–465, Kluwer Academic Publishers, 2001. doi:10.1023/A:1014798618404. (on page 92).

[Oeh11] Jens Oehlerking. *Decomposition of stability proofs for hybrid systems*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2011. urn:nbn:de:gbv:715-oops-14554. Available from: http://oops.uni-oldenburg.de/id/eprint/1375. (on page 46).

[OM88] Edward P. Oppenheimer and Anthony N. Michel. Application of interval analysis techniques to linear systems. II. The interval matrix exponential function. *IEEE Transactions on Circuits and Systems*, 35(10):1230–1242, 1988. doi:10.1109/31.7598. (on page 44).

[PQ08] André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated*

Reasoning, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2008. doi:10.1007/978-3-540-71070-7_15. (on page 49).

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, UK, second edition, 1992. (on page 90).

[PW07] Andreas Podelski and Silke Wagner. Region stability proofs for hybrid systems. In Jean-François Raskin and P. S. Thiagarajan, editors, *Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 4763 of *LNCS*, pages 320–335. Springer, 2007. doi:10.1007/978-3-540-75454-1_23. (on page 141).

[RMC09] Nacim Ramdani, Nacim Meslem, and Yves Candau. A hybrid bounding method for computing an over-approximation for the reachable set of uncertain nonlinear systems. *IEEE Transactions on Automatic Control*, 54(10):2352–2364, 2009. doi:10.1109/TAC.2009.2028974. (on pages 16, 130, 131, 135).

[RMC10] Nacim Ramdani, Nacim Meslem, and Yves Candau. Computing reachable sets for uncertain nonlinear monotone systems. *Nonlinear Analysis: Hybrid Systems*, 4(2):263–278, Elsevier, 2010. doi:10.1016/j.nahs.2009.10.002. (on pages 130, 131).

[RS07] Stefan Ratschan and Zhikun She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *ACM Transactions on Embedded Computing Systems*, 6(1), ACM, 2007. Article No. 8. doi:10.1145/1210268.1210276. (on pages 48, 127, 138).

[Sht00] Ofer Shtrichman. Tuning sat checkers for bounded model checking. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2000. doi:10.1007/10722167_36. (on pages 99, 119).

[SKHP97] Olaf Stursberg, Stefan Kowalewski, Ingo Hoffmann, and Jörg Preußig. Comparing timed and hybrid automata as approximations of continuous systems. In Panos Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems IV*, volume 1273 of *LNCS*, pages 361–377. Springer, 1997. doi:10.1007/bfb0031569. (on pages 138, 163).

[ST98] Karsten Strehl and Lothar Thiele. Symbolic model checking of process networks using interval diagram techniques. In *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*, pages 686–692, Nov 1998. Available from: http://www.cs.york.ac.uk/rts/docs/SIGDA-Compendium-1994-2004/papers/1998/iccad98/pdffiles/11d_3.pdf, doi:10.1109/ICCAD.1998.144343. (on page 128).

[Sta97] Ole Stauning. *Automatic validation of numerical solutions*. PhD thesis, Danmarks Tekniske Universitet (DTU), Kgs. Lyngby, Denmark, 1997. IMM-PHD-1997-36. Available from: http://orbit.dtu.dk/services/downloadRegister/5268944/imm2462.pdf. (on pages 47, 81, 91).

[THF+07] Tino Teige, Christian Herde, Martin Fränzle, Natalia Kalinnik, and Andreas Eggers. A generalized two-watched-literal scheme in a mixed boolean and non-linear arithmetic constraint solver. In José Neves, Manuel Filipe Santos, and José Manuel Machado, editors, *Proceedings of the* 13^{th} *Portuguese Conference on Artificial Intelligence (EPIA 2007)*, New Trends in Artificial Intelligence, pages 729–741. APPIA, December 2007. Workshop on Search Techniques for Constraint Satisfaction (STCS). Available from: http://hs.informatik.uni-oldenburg.de/tino/papers/TeigeEtAl_EPIA07.pdf. (on page 67).