Induction-based Verification of Timed Systems

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
an der
Fakultät für Elektrotechnik, Informatik und Mathematik
der
Universität Paderborn

vorgelegt von

Tobias Isenberg, M.Sc.

Paderborn, Februar 2016

Abstract

Computer controlled systems are a foundation to today's modern society. Their use and impact are ever-growing, in particular when considering trends like smart homes and Industry 4.0. The development of such systems is non-trivial. Correctness of the functionality provided by these systems is of importance, since they are often deployed in safety-critical scenarios, where a failure could lead to a loss in production value or the risk of life. Many of these systems are real-time systems or include some kind of timed behavior.

In order to ensure a certain quality of these systems, model-based design approaches can be employed. Within these structured procedures, the systems under development are designed using models that specify certain aspects of them. This structured procedure provides for fewer errors being made. In combination with formal methods, the absence of erroneous behavior can be ensured. To this end, formal languages with a mathematically defined semantics are employed, and formal verification, based on this semantics, is used to reason about the absence of erroneous behavior. There exist several formalisms for the verification of timed systems, e.g., the formalism of networks of timed automata used in this thesis. However, today's techniques and tools for this task often suffer from the same deficiency. They easily run out of memory when exploring large, complex models due to the enormous amount of states (state explosion problem) they need to keep track of.

An additional challenge arises when considering the procedure during the design phase. The models are repeatedly reconfigured, i.e., changed to reflect design choices until a final design is found. In addition, they may also be reconfigured during lifetime, in particular when considering systems of Industry 4.0 that are self-optimizing and adaptable. In consequence, the respective models are reconfigured to reflect these adaptations. These reconfigurations raise the need to redo verifications since the state spaces of the models might have changed. Techniques employed for these redone verifications have to meet specific demands, in particular efficiency, since they might be employed in a sort of online verification during the lifetime of a system.

In this thesis, we approach both of the challenges mentioned above. First, we provide a technique for the verification of safety properties specifying the absence of erroneous behavior for networks of timed automata. Our technique deliberately works distinct from other state-of-the-art approaches in this field, as it employs induction and, thus, avoids explicit exploration and storage of states. In consequence, it is a valuable complement to existing technologies as its strengths and weaknesses differ to those of other technologies. Our approach combines the IC3 algorithm, well known in the hardware verification domain, with the Zone abstraction, used for timed verification. The result, *IC3 with Zones* is competitive and includes the strengths of both components, namely the time abstracting efficiency of zones and the efficiency on discrete structured of IC3. We show the practicality of it in numerous experiments on different aspects of scalability.

In our continued work, we employ our algorithm for the verification of reconfigured models. To this end, we reuse an inductive invariant computed by *IC3 with Zones*.

This basic idea works particularly well for a special class of systems, denoted Parameterized Timed Systems. They contain an arbitrary, but fixed number of instances of a process as is the case, e.g., in client-server settings with an arbitrary number of clients. We propose an approach to avoid the online verification for such systems, as would be needed whenever the number of processes is changed, e.g., a client is added. Our technique enables an *a priori verification* of the entire system, irrespective of the actual number of instances. To this end, we verify the safety property for the smaller models of the family and reuse the computed inductive invariants. These are adapted using the symmetry inherent in Parameterized Timed Systems, and employed for the reasoning about the entire system. For this purpose, we propose and prove a Termination Theorem that enables this reasoning. The practicality of our approach is shown using several experiments.

In addition, we examine the reusability of the inductive invariant for general models and reconfigurations with the aim to speed up the verification for reconfigured models. To this end, the same acceleration technique is employed as in the special case for Parameterized Timed Systems. We discuss in detail, why it is hardly feasible to adapt the inductive invariant to reflect a reconfiguration in general. As a result, we give a *best-guess approach* that adapts the invariant where possible. Numerous experiments show that this technique is of value even so, in particular when the reconfigurations are small.

Zusammenfassung

Rechnergesteuerte Systeme bilden den Grundstein der heutigen Gesellschaft. Ihr Einsatz und ihre Verbreitung wachsen stetig, unterstützt durch Trends wie Smart Homes und Industrie 4.0. Die Entwicklung solcher Systeme ist jedoch schwierig. Ihre korrekte Funktionalität ist von großer Wichtigkeit, da diese Systeme oftmals in sicherheitskritischen Szenarios eingesetzt werden, in denen eine Fehlfunktion zu finanziellem Schaden oder sogar zu Gefahr für Leib und Leben führen kann. Viele dieser Systeme sind Echtzeitsysteme oder besitzen zeitgesteuertes Verhalten.

Um die Qualität der Systeme sicherzustellen, können Modell-basierte Design-Ansätze verwendet werden. In diesen strukturierten Vorgehensweisen werden die zu erstellenden Systeme mithilfe von Modellen entworfen, die ein System unter bestimmten Gesichtspunkten spezifizieren. Ein solch strukturierter Ansatz sorgt dafür, dass weniger Fehler gemacht werden. Durch die zusätzliche Nutzung formaler Methoden kann die Abwesenheit von fehlerhaftem Verhalten sichergestellt werden. Hierzu werden formale Sprachen mit mathematisch definierter Semantik genutzt, sowie formale Verifikation, die basierend auf der formalen Semantik über die Abwesenheit von fehlerhaftem Verhalten schlussfolgert. Etliche Formalismen existieren für die Verwendung zur Verifikation zeitlicher Systeme, zum Beispiel die Netzwerke von zeitlichen Automaten, die in dieser Arbeit verwendet werden. Die meisten der heutigen Techniken für die Verifikation zeitlicher Systeme leiden jedoch unter dem gleichen Problem. Bei der Exploration großer, komplexer Modelle benötigen sie ein enormes Maß an Speicher, um die besuchten Zustände zu speichern. Durch das enorme Anwachsen der Anzahl an Zuständen (Zustandsexplosionsproblem) schlägt die Verifikation oft fehl.

Eine zusätzliche Herausforderung entsteht durch den Ablauf der Design-Phase. Bis das finale Design entschieden ist, können die Modelle wiederholt rekonfiguriert, d.h. geändert, werden um Designentscheidungen umzusetzen. Zusätzlich kann dies auch zur Laufzeit der Systeme auftreten, insbesondere bei Systemen der Industrie 4.0, die selbstoptimierend und anpassungsfähig sind. Dementsprechend können sich die entsprechenden Modelle zur Laufzeit ändern. Diese Rekonfigurationen verlangen eine erneute Verifikation, da sich der Zustandsraum geändert haben kann. An die Techniken, die für diese erneuten Verifikationen eingesetzt werden, werden spezielle Anforderungen gestellt, insbesondere Effizienz, da sie während der Laufzeit als Online-Verifikationen durchgeführt werden.

In dieser Arbeit werden beide zuvor genannten Herausforderungen angegangen. Es wird eine Technik zur Verifikation von Sicherheitseigenschaften, die die Abwesenheit von fehlerhaftem Verhalten definieren, in Netzwerken von zeitlichen Automaten vorgestellt. Diese Technik hat ein grundlegend anderes Funktionsprinzip als andere aktuelle Ansätze, da sie Induktion nutzt und somit die explizite Exploration und Speicherung von Zuständen umgeht. Hierdurch ist sie eine wertvolle Ergänzung zu den bestehenden Ansätzen, da die Stärken und Schwächen unterschiedlich ausfallen. Der Ansatz kombiniert den IC3 Algorithmus, der in der Hardware Verifikation erfolgreich eingesetzt wird, mit der Zonen-Abstraktion, die Grundlage vieler zeitlicher Verifikationsverfahren ist. Der resultierende Algorithmus *IC3 mit Zonen* ist wettbewerbsfähig und enthält

Stärken von beiden genannten Komponenten. Dies sind insbesondere die gute Fähigkeit zur Zeit-Abstraktion von Zonen und die Effizienz auf diskreten Strukturen von IC3. Die Anwendbarkeit und Skalierbarkeit der Technik wird in vielen Experimenten gezeigt.

Aufbauend auf dieser Technik werden Ansätze vorgestellt, die auf die Verifikation von rekonfigurierten Modellen abzielen. Sie nutzen dazu die induktive Invariante, die von *IC3 mit Zonen* berechnet wird.

Die Idee der Wiederverwendung der Invariante ist besonders gut einsetzbar bei einer speziellen Art von Systemen, genannt Parametrisierte Zeitliche Systeme. Diese enthalten eine beliebige, aber feste Anzahl an Instanzen eines Prozesses, wie beispielsweise in einem Client-Server System mit beliebiger Anzahl Clients. Unser Ansatz umgeht die Notwendigkeit einer Online-Verifikation für solche Systeme, die beispielsweise notwendig wäre, wenn die Anzahl an Clients geändert würde. Die Technik basiert auf einer Verifikation der gesamten Familie an Modellen, die von vornherein ausgeführt wird. Hierzu wird die Sicherheitseigenschaft für die kleineren Modelle verifiziert. Die dabei berechneten induktiven Invarianten werden anhand der Symmetrie, die durch die Parametrisierung gegeben ist, angepasst und danach verwendet, um über die gesamte Familie an Modellen zu urteilen. Hierfür wird ein Terminierungstheorem gegeben und bewiesen. Etliche Experimente zeigen den Nutzen der Technik.

Zusätzlich wird untersucht, wie man die induktiven Invarianten für allgemeine Rekonfigurationen und Modelle nutzen kann, so dass die Verifikation für rekonfigurierte Modelle schneller wird. Es werden die gleichen Techniken zur Beschleunigung von Verifikationen mithilfe der Invarianten benutzt, wie in der Arbeit zu parametrisierten Systemen. Detailliert wird dargelegt, warum diese Invarianten im allgemeinen Fall kaum an die Rekonfiguration angepasst werden können. Schließlich wird ein Ansatz vorgestellt, der die Invariante soweit wie möglich anpasst. Experimente zeigen, dass diese Technik eine Bereicherung im Kontext der Online-Verifikation darstellt.

Acknowledgments

Most importantly, I would like to express my sincere gratitude to my supervisor Prof. Dr. Heike Wehrheim for her guidance and support. The last three years have been a rewarding, interesting time and I am very thankful for the opportunity to work in her research group and to write my thesis. I also want to express my gratitude to my second supervisor Prof. Dr. Oliver Niggemann for his support.

Sincere thanks go to Prof. Dr. Hans Kleine Büning, Jun.-Prof. Dr. Heiko Hamann and Dr. Theodor Lettmann for being part of my PhD committee.

In addition, I want to thank all current and former members of the research group that have crossed my path: Steffen Beringer, Dr. Galina Besova, Marie-Christine Jakobs, Julia Krämer, Dr. Thomas Ruhroth, Elisabeth Schlatt, Alexander Schremmer, Dr. Dominik Steenken, Dr. Nils Timm, Manuel Töws, Oleg Travkin, Sven Walther and Steffen Ziegert. They provided for a pleasant atmosphere and a great time.

Furthermore, I owe special gratitude to Dennis Wolters and Anna Lena Isenberg for proof reading parts of this thesis.

Finally, I want to thank my parents, Werner and Ulrike Isenberg, for always encouraging me and providing, together with my twin brother Florian and my sister Anna Lena, for an amazing childhood that made me who I am.

Last, but not least, I want to express my loving gratitude to my girlfriend Nadja. She knows how to make me smile and she has always been there to support me.

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
- 1.1 Problem Definition . . . 3
- 1.2 Contribution . . . 5
- 1.3 Thesis Outline . . . 9

2 Background **11**
- 2.1 Timed Automata . . . 11
  - 2.1.1 Decidability and Abstractions . . . 19
  - 2.1.2 Related work . . . 20
- 2.2 SAT- & SMT-Solving . . . 23
  - 2.2.1 SAT-Solving . . . 23
  - 2.2.2 SMT-Solving . . . 24
- 2.3 Induction based Reasoning . . . 25
- 2.4 IC3 . . . 28
  - 2.4.1 Algorithm and Explanation . . . 28
  - 2.4.2 Optimizations . . . 35
  - 2.4.3 IC3 with SMT . . . 37
  - 2.4.4 IC3 for TA . . . 38

3 Timed Automata Verification via IC3 with Zones **41**
- 3.1 SMT-Encoding . . . 42
  - 3.1.1 Variables . . . 43
  - 3.1.2 Encoding of the Initial States . . . 44
  - 3.1.3 Encoding of Clock Constraints . . . 45
  - 3.1.4 Encoding of Integer Constraints . . . 46

3.1.5 Encoding of Invariants . . . 46
3.1.6 Encoding of Clock and Integer Updates . . . 48
3.1.7 Encoding of Edges . . . 49
3.1.8 Encoding of Transition Relation . . . 51
3.1.9 Encoding of Safety Property . . . 52
3.1.10 Usage of the Encodings in the Queries . . . 53
3.2 State Extraction . . . 54
3.3 Zone Abstraction . . . 56
3.3.1 Zone Computation . . . 58
3.3.2 Integer Constraint Computation . . . 60
3.4 Algorithm . . . 60
3.4.1 Termination . . . 66
3.5 Evaluation . . . 68
3.5.1 Benchmark Models . . . 68
3.5.2 Implementation . . . 76
3.5.3 Heuristics for ordering Literals . . . 76
3.5.4 Experiments . . . 78
3.5.5 Inductive Strengthening Experiments . . . 87
3.6 Summary . . . 88

4 Incremental Inductive Verification of Parameterized Timed Systems 89
4.1 Restrictions . . . 92
4.2 Symmetry . . . 95
4.3 Specification via Templates . . . 97
4.4 Decidability . . . 100
4.5 Basic Approach . . . 101
4.6 Optimizations . . . 104
4.6.1 Feedback Loop . . . 104
4.6.2 Generalization . . . 106
4.6.3 Combination . . . 106
4.7 Evaluation . . . 108
4.7.1 Experiments . . . 108
4.7.2 Comparison with fixed Models . . . 113
4.8 Related Work . . . 114
4.9 Summary . . . 116

5 Verification of Extended Parameterized Timed Systems 117
5.1 Extension of the Modeling Approach . . . 118
5.2 Symmetry . . . 121
5.3 Experiments . . . 128
5.4 Summary . . . 129

6 Inductive Verification of Reconfigured Models 131
6.1 Adaptation of the Formula . . . 135
6.2 Approach in Summary . . . 140

6.3 Experiments . . . 140
6.3.1 Experiments with Addition . . . 141
6.3.2 Experiments with Deletion . . . 141
6.3.3 Experiments with Large Models . . . 142
6.3.4 Experiments with Constants . . . 143
6.3.5 Counterexample Experiment . . . 146
6.4 Related Work . . . 147
6.5 Summary . . . 149

7 Conclusion **151**
7.1 Discussion . . . 153
7.1.1 Design Decisions . . . 153
7.1.2 Application Scope and Restrictions . . . 155
7.1.3 Strengths and Weaknesses . . . 156
7.2 Future Work . . . 157
7.2.1 *IC3 with Zones* - Technique . . . 157
7.2.2 Parameterized Timed Systems - Technique . . . 158
7.2.3 General Reconfigurations - Technique . . . 158
7.3 Summary . . . 159

A Experimental Results **161**
A.1 Runtime and Memory Consumption of Experiments . . . 161

B Proofs **175**
B.1 Templates guarantee Symmetry . . . 175
B.2 Proof of Termination Theorem . . . 183

Bibliography **191**

List of Figures

2.1 Timed Automaton model of the Fischer Mutual Exclusion algorithm [UPP] 15

3.1 Network of Timed Automata modeling two processes executing the Fischer Mutual Exclusion algorithm . . . 45
3.2 An infinite number of satisfying interpretations might exist . . . 55
3.3 The surrounding zone might not be unique . . . 57
3.4 The CSMA/CD model from Uppaal [UPP] . . . 70
3.5 The FDDI model from Uppaal [UPP] . . . 70
3.6 The FDDIcount model . . . 72
3.7 The Fischer model by Bruttomesso [Bru+12] . . . 72
3.8 The Lamport model by Bruttomesso [Bru+12] . . . 73
3.9 The Shavit-Lynch model by Bruttomesso [Bru+12] . . . 74
3.10 The shrunk Lamport model . . . 74
3.11 The Shavit-Lynch model from PAT [PAT] . . . 75
3.12 The Lemgo model . . . 75
3.13 Results of scalability Experiments with *IC3 with Zones* (*Fischer_U* model) 79
3.14 Results of scalability experiments with *IC3 with Zones* (*CSMA/CD* model) 79
3.15 Results of scalability experiments with *IC3 with Zones* (*FDDI* model) . . . 80
3.16 Results of scalability experiments with *IC3 with Zones* (*FDDIcount* model) 81
3.17 Results of experiments with *IC3 with Zones* on the effect of identifier assignments (*Fischer_U* model with distinct location identifier assignments) 82
3.18 Results of experiments with *IC3 with Zones* on the effect of the number of locations (two *Fischer* models with distinct number of locations) . . . 83
3.19 Results of experiments with *IC3 with Zones* on the effect of the number of locations (two *Lamport* models with distinct number of locations) . . . 84
3.20 Comparison of *IC3 with Zones* with Kindermann's approach [KJN12b] (*Fischer_U* model) . . . 85

3.21 Comparison of *IC3 with Zones* with Kindermann's approach [KJN12b] (*CSMA/CD* model) . . . 86
3.22 Comparison of *IC3 with Zones* with Kindermann's approach [KJN12b] (*FDDI* model) . . . 86

4.1 Overview of our incremental workflow . . . 91
4.2 Unbalanced comparison operator destroys symmetry . . . 94
4.3 Template for the *Fischer_U* model . . . 98
4.4 Basic workflow . . . 101
4.5 Optimized workflow . . . 107
4.6 Templates for the models *Fischer_U* and *Fischer_B* . . . 108
4.7 Templates for the models *Lamport_B* and *Lamport_S* . . . 109
4.8 Templates for the models *ShavitLynch_B* and *ShavitLynch_P* . . . 109

5.1 Illustration of the Train-Controller-Gate model . . . 120
5.2 Parameterized Train-Controller-Gate model . . . 120
5.3 Optimized workflow . . . 125

6.1 Reconfigured Fischer model with all time constants changed to 2048 . . . 134
6.2 Reconfigured Fischer model with a single time constant changed to 2048 138
6.3 Reconfigured Lemgo model with time constant 8001 changed to 10000 . . 139
6.4 Results of experiments with added timed automata (*CSMA/CD*) . . . 142
6.5 Results of experiments with added timed automata (*FDDI*) . . . 143
6.6 Results of experiments with deleted timed automata (*CSMA/CD*) . . . 144
6.7 Results of experiments with deleted timed automata (*FDDI*) . . . 145
6.8 Reconfigured Lemgo model with time constant 8001 changed to 7000 . . 146

List of Tables

3.1 Runtime comparison using different heuristics for variable ordering . . . 77
3.2 Scalability in matters of the size of time constants 85
3.3 Results of experiments with *IC3 with Zones* (*Lemgo* model) 87

4.1 Comparison of inductive strengthenings for different number of automata 91
4.2 Example of the extrapolation procedure 103
4.3 Example for the generalization procedure 107
4.4 Results of experiments with parameterized systems (basic workflow) . . 110
4.5 Results of experiments with parameterized systems (using *Generalization*) 110
4.6 Results of experiments with parameterized systems (using *Feedback*) . . . 111
4.7 Results of experiments with parameterized systems (both Optimizations) 112
4.8 Comparison of experimental results with verifications of fixed instances . 113
4.9 Validation times for computed inductive strengthenings 114

5.1 Results of experiments with extended parameterized timed system (1) . . 128
5.2 Results of experiments with extended parameterized timed system (2) . . 129

6.1 Comparison of inductive strengthenings for reconfigured model 133
6.2 Results of experiments reusing inductive strengthening 134
6.3 Results of experiments reusing adapted inductive strengthening 136
6.4 Comparison of inductive strengthenings regarding time constants 139
6.5 Results of experiments with and without adaptation (*Fischer_U* model with single constant changed) 144
6.6 Results of experiments with and without adaptation (*Lemgo*(10000) model) 146
6.7 Results of experiments with *Lemgo*(7000) model incl. counterexample trace 147

A.1 Results of experiments with *IC3 with Zones* (*CSMA*/*CD* model) 161
A.2 Results of experiments with *IC3 with Zones* (*Fischer_U* model) 162
A.3 Results of experiments with *IC3 with Zones* (*FDDI* model) 163

A.4 Results of experiments with *IC3 with Zones* (*Fischer_B* model) 164
A.5 Results of experiments with *IC3 with Zones* (*Lamport_B* model) 164
A.6 Results of experiments with *IC3 with Zones* (*Lamport_S* model) 165
A.7 Results of experiments with *IC3 with Zones* (*ShavitLynch_B* model) 165
A.8 Results of experiments with *IC3 with Zones* (*ShavitLynch_P* model) 166
A.9 Results of experiments with *IC3 with Zones* (*FDDIcount* model) 167
A.10 Results of experiments with location identifiers (*Fischer_U* model) 168
A.11 Inductive strengthenings: Validation and size (*Fischer_U* model) 169
A.12 Inductive strengthenings: Validation and size (*Fischer_U*(*switched*) model) 170
A.13 Inductive strengthenings: Validation and size (*CSMA/CD* model) 171
A.14 Inductive strengthenings: Validation and size (*FDDI* model) 171
A.15 Inductive strengthenings: Validation and size (*FDDIcount* model) 172
A.16 Inductive strengthenings: Validation and size (*Fischer_B* model) 172
A.17 Inductive strengthenings: Validation and size (*Lamport_B* model) 172
A.18 Inductive strengthenings: Validation and size (*Lamport_S* model) 172
A.19 Inductive strengthenings: Validation and size (*ShavitLynch_B* model) . . 173
A.20 Inductive strengthenings: Validation and size (*ShavitLynch_P* model) . . 173
A.21 Inductive strengthenings: Validation and size (*Lemgo* model) 173

1 Introduction

More than ever does today's society rely on the correct functionality of software and hardware, as more and more tasks are carried out by computer-controlled systems. This trend can be observed in everyones personal environment, as well as in the industry. Examples of importance are the usage of connected controllers in *smart homes*, e.g., for doors and radiators, or the paradigm of *Industry 4.0* that facilitates the interconnection and adaptation of all components engaged in a production system. As a result, the systems and their interaction grow more and more complex.

The increased complexity poses a serious threat to the correct functionality of the systems, as a mistake might more easily stay unnoticed within a large system. With most of the systems being safety critical, every possible effort must be made to avoid and find such mistakes that result in unintended, erroneous behavior. Considering again the above examples, a mistake might result in the smart door opening erroneously, or a sudden, undesired stop in a production system. The consequence might be a loss in production value or the risk of life.

In response to this problem, an increasing effort is made to research and introduce structured approaches that are able to ensure a certain quality of a system. One such approach for software is the model driven software development (MDSD) [B+05] that starts as early as possible during development, namely in the design phase. In such structured approaches, a system is specified via models that depict different aspects of it. For example, the unified modeling language (UML) [RJB04] is typically employed for this task during the development of software systems. UML contains numerous types of diagrams to specify different aspects of the system, e.g., use case diagrams to capture behavioral requirements or class diagrams to capture structural relations. These diagrams are not only employed for specification, but are also used for documentation purpose and are often involved in the construction of the final system. To this end, automated transformations and generators are employed that fully automate the task of transformation into other modeling formalisms, or the generation of code. This structured procedure for specification and generation of

parts of a software system is often used and helps to achieve a good code quality and architecture. However, every manual specification and modeling might introduce unintended, erroneous behavior in the models and, using the automated generators and transformations, into the final system. For this reason, formal methods are employed that are able to guarantee the absence of such behavior in the models and, in consequence, in the final system, provided the transformations are correct. These formal methods basically draw on three important elements. First, the unintended, erroneous behavior has to be specified. Second, the models that are to be checked need to be specified using a formalism based on a rigorous mathematical semantics in order to enable reasoning about erroneous behavior. Third, the methods that check the absence of the unintended behavior in the given model are required to be sound.

There exist many distinct formalisms for modeling the system under development. Many of them are specifically designed to capture certain aspects of the system, e.g., its structure, communication or timed behavior. They are based on a rigorous defined mathematical semantics that allows the reasoning about properties. Using the mathematical foundation, the reasoning about properties of the models can be done using one of the approaches for formal verification. There exist several such techniques, where deductive verification and model checking represent the most well-known categories. Among the most common modeling formalisms are automata, petri nets [Mur89], process calculi [Mil80; Mil99], Z [SA92] and the B-method [AAH05]. In addition, there exist formalisms that encompass an explicit notion of time, e.g., timed automata [AD90], hybrid automata [Hen00], timed petri nets [Ram73], Timed CSP [RR86], Duration Calculus [CHR91] and Timed Graph Transformation Systems [HHH10]. These are of special interest, as real-time systems are of growing importance.

As an example, consider the numerous embedded real-time controllers that operate many safety-critical systems, e.g., the time-critical sensing of a car accident with almost immediate inflation of an air bag. Additionally, in the industry more and more real-time systems are employed, as the paradigm of Industry 4.0 often requires the components to communicate using real-time protocols in order to achieve adaptivity and self-optimization of all components in a plant. There exist several additional examples that illustrate today's importance of real-time systems and time based behavior.

In general, such behavior is of interest in many development scenarios and so are the respective formalisms and formal verification methods. As in the untimed case, the formalisms are employed to model specific aspects of the system under development, and verification is employed to check whether specified properties hold true for the model.

As explained above, the systems and their interaction grow more and more complex. In consequence, the respective models of these systems become increasingly complex. The formal verification carried out to ensure the quality of the system needs to be able to handle such large models. But with models growing intensely, the verification of properties for them becomes even more challenging due to the

state explosion problem. An additional problem is the fact that models are often reconfigured, i.e., changed. In the MDSD process mentioned above, reconfigurations may occur repeatedly in the design phase until a final design is found. A reason might be that required properties are not met or that additional parts have to be included.

In addition, there exist scenarios in which a model is reconfigured at lifetime of a developed system. These reconfigurations, i.e., changes, to the model reflect changes to the running system. They are of huge importance, in particular, when considering the adaptivity and self-optimization proposed for Industry 4.0 plants.

Examples for such reconfigurations at runtime are the adaptation of a plant in which it switches to another operating mode in order to save energy, or the modification of the plant including new components. Furthermore, the substitution of mechanical parts might result in a different timing behavior if the parts possess distinct operating characteristics.

In such cases, the system is reconfigured at lifetime, possibly in ways that could not be foreseen within the design-phase. There exist works that are concerned with intelligent assistant systems for such scenarios [JN12], but they do not consider the context of formal verification. Every change in the system might lead to unintended behavior. Thus, there exists a need to verify the properties, which have already been verified for the original model, again for the reconfigured one.

To this end, the reconfigured model could be provided manually by a model designer, or even be learned automatically by machine learning algorithms, e.g as timed automata [Mai14] or hybrid automata [Nig+12].

In any case, the verification should be as efficient as possible, in particular, when considering that the properties have already been verified for the original model that might be very similar. Moreover, when taking into account that the reconfigured system might already be running, the urgency of this task becomes obvious.

Either way, a reconfigured model would result in a renewed need for verification. With the growing complexity of systems and their models in mind, these verifications should be as efficient as possible. It is, thus, not desirable that these verifications for reconfigured models employ techniques that start from scratch.

1.1 Problem Definition

Many of today's systems exhibit timed behavior. They rely on time in diverse ways, e.g., using real-time communication protocols as can be found for example in the PROFINET standard [Fel04]. Furthermore, the interaction of several components can be seen as timed interdependency, e.g., since the distribution of a product in the plant relies on its completed creation. The quantity of systems with such behavior shows the significance of timed verification. For this task, models are created that reflect the time-based aspects of the system. As illustrated above, the formal modeling and verification can be done at design time of the system, e.g., in the context of model-based design processes, or later on during runtime, e.g., using learned models. Either way, a formal modeling language is required in order to

model the mentioned timed behavior. There exist formalisms that are capable of modeling continuous real-time since the early 1990s. They offer various levels of expressiveness and usability.

In this thesis, we employ the formalism of networks of timed automata [AD90]. It is one of the most well-known formalisms with roughly 25 years of research. The modular structure as a network consisting of several timed automata enables an elaborate way of modeling individual components, e.g., representing software processes or different hardware controllers. It is, thus, well suited for the modeling of interconnected components as is the case in Industry 4.0 plants or smart homes. The decidability of reachability questions in this formalism enables the use of formal verification [AD90].

In networks of timed automata, the interconnection of the distinct components' behaviors is given implicitly via time. In addition, the formalism allows an explicit interaction enforced via synchronization and shared variables.

Like in other automata formalisms, timed automata are based on a discrete structure that offers an easy means to specify distinct states of the various components. This discrete structure captures the overall, untimed behavior of the system. However, in timed systems, this behavior highly depends on time.

To this end, constraints are introduced that restrict the allowed behavior as a function of the elapsed time. This intuitive mechanism of observing the progress of time and, in dependence, limiting the allowed behavior is a basic concept, also included in several other timed formalisms, e.g., timed petri nets or timed graph transformation systems.

In this thesis, we are concerned with the verification of safety properties that specify the non-reachability of error states. These error states are an intuitive way to specify unintended or undesired behavior. The absence of such erroneous behavior can be guaranteed via a formal verification of the safety properties.

For the formalism of networks of timed automata, there already exist techniques and tools to verify such properties. Considering the requirements of today's systems and Industry 4.0, these approaches are not well suited.

Most of them are optimized for runtime, i.e., they are fast at the expense of used memory, mostly due to explicit exploration. Taking into account the increasing size of the systems and their interconnectivity, an approach is desirable that is capable of verifying large models without running out of memory easily. Furthermore, most of the current techniques are not suited to cope with reconfigured models as may happen often, e.g., during model-based design processes. These approaches require a verification of the properties for the reconfigured model that starts from scratch.

These are the two problems we approach in this thesis. They are summed up in the following.

- The verification of large models, in particular those with a large set of reachable states, poses a problem for many existing techniques. Many of these techniques rely on an explicit exploration and representation of the forward or backward reachable states. These states have to be stored in order to know which states

have already been explored. Trivially, a large number of such states results in huge memory requirements, which is often the reason that a verification is aborted. We will focus on a technique that does not require the exploration of each reachable state. To this end, it employs induction. We seek to develop an approach that handles complex, large models well.

- The second problem in the above setting is that many models might be reconfigured during design time or lifetime. Current verification techniques for timed automata are not designed for an efficient verification of the reconfigured models. They will start a verification from scratch and, thus, waste the chance to reuse a previous verification result. Even if they would be capable of reusing a previous outcome, their explicit method would need to check every such state again, as the state space has changed. Our focus on an inductive method establishes a distinct chance, as it can be reused in a more elaborate way. We seek to develop a technique that efficiently verifies safety properties for reconfigured models in order to support repeated reconfigurations during design time, and in order to enable online verification, i.e., verifications during lifetime of a system.

In the following, we will state our contributions for these two problems.

1.2 Contribution

This thesis is concerned with both of the problems pointed out above. We propose a novel technique for the verification of safety properties for networks of timed automata. Based on an inductive invariant computed by this technique, we accelerate the verification of safety properties for reconfigured models.

In detail, our contribution is as follows.

IC3 with Zones As mentioned above, many of the verification techniques for safety properties in networks of timed automata suffer from large memory requirements. When handling large models with a huge set of reachable states, the explicit exploration that is used in many state-of-the-art approaches easily runs out of memory. The reason is that the algorithm needs to store the already explored states in order to detect whether a new state was already discovered. Thus, their memory need can be seen as one of the most important weaknesses of the current verification techniques.

We avoid the explicit exploration and discovery of every reachable state. To this end, we transfer the IC3 algorithm to the domain of timed verification in a competitive way. We present, implement and evaluate a concept that combines the strengths of the following two elements.

- The IC3 algorithm [Bra11] has proven to be successful and efficient, both regarding runtime and memory, for the verification of safety properties on discrete structures. It is based on induction and SAT-solving and, thus, works differently than the exploration algorithms usually employed in timed verification. Its mechanisms avoid the discovery and storage over every reachable

state, but instead compute an inductive invariant stored as a compact propositional formula. We employ this efficiency and distinctiveness in our approach in order to avoid the common problem of storing explicitly explored states.

- The Zone abstraction has been employed numerous times for timed verification. It is versatile in that it can be very fine grained or very coarse and there exist sophisticated algorithms to efficiently manipulate zones. Using these efficient algorithms and the proper coarseness, we employ this abstraction in order to handle the infinite state space imposed by the semantics of timed automata.

By combination of these two elements, we achieve a transfer of the IC3 algorithm to the domain of timed verification, which is competitive, unlike previous attempts. The result is a technique that works in a way distinct than other state-of-the-art techniques in timed verification, namely by using induction. It combines the strengths of the two included elements and, thus, avoids the weakness of storing explicitly explored states, i.e., the enormous need for memory.

Our technique employs SMT-solving [BST10] and is based on induction. The main challenge is the infinite state transition system imposed by the semantics of timed automata. As explained, we employ the Zone abstraction in order to cope with this problem and ensure termination. The integration of this abstraction into the IC3 algorithm is based on the structure of the SMT-queries. Our combination achieves termination and efficiency.

We show this efficiency and practicality of our proposed technique in numerous experiments. Using standard examples from literature, we illustrate that our approach is able to outperform state-of-the-art tools by models up to three times as large. We present experiments to examine the scalability of our approach with regard to the model size, the used location encoding and the usage of integer variables.

Summing up, we present a combination of two techniques from distinct domains. It does, unlike most other techniques in the domain, not explicitly explore and store the reachable state space. Instead, our technique employs induction for the verification of safety properties in timed systems modeled as networks of timed automata. Due to its distinct working paradigm, the presented technique is well suited to be used for large, complex models as in today's systems. It is a relevant alternative to long-established approaches and might give thought-provoking impulses.

Our technique yields a valuable additional outcome in case of verification success, which we employ for the second problem defined above (Section 1.1). It computes an inductive invariant that we store and reuse for the verification of properties for reconfigured models.

In the following, we state our contribution in this direction. The first work deals with specific reconfigurations that add an instance of a process to an already existing number of such instances. Doing so numerous times creates a system with an arbitrary large, but fixed number of processes. These systems are denoted parameterized timed systems.

Parameterized Timed Systems Parameterized Timed Systems occur frequently in modern systems. For example, consider a scenario in which a number of robots requires the exclusive access to a resource, e.g., to the top element of a pile of raw materials in order to start working. In order to avoid problems, the decision which robot gains access can be negotiated by their controllers using a mutual exclusion algorithm. To ensure error-free functioning, the involved controllers might have been modeled and mutual exclusion has been verified. However, if an additional robot is added, e.g., to increase the productivity, the same property needs to be verified again for the reconfigured system including the additional controller. There exist many other scenarios that also raise the need for a new verification whenever the system is reconfigured with a new number of processes.

We have proposed a novel approach for the verification of safety properties for such parameterized timed systems that consist of an arbitrary, but fixed number of instantiations of a process. Our technique enables an a priori verification of the entire system, where the actual number of instances during runtime does not matter. Thus, the system can be reconfigured by addition or deletion of processes without raising the need for a new verification. Instead, based on the a priori verification one can be sure that the safety property holds irrespective of the number.

This technique avoids the need for online verification for systems that can be modeled as such a parameterized timed system. It is competitive and able to reason about the entire family of models in the system by considering only a few fixed instances. To this end, it reuses verification results obtained during the verification for these fixed instances in order to reason about all larger models. This reuse is extremely efficient.

Our approach consists of several important parts. We propose a workflow that incrementally verifies the safety property in question for models with increasing size. It employs our algorithm *IC3 with Zones* presented in the previous paragraph and, thus, profits from possible improvements in the future. The inductive invariants computed by the algorithm are adapted using the symmetry inherent in parameterized timed systems and later on used to reason about the entire system. To this end, we propose and prove a Termination Theorem that is the basis for this reasoning.

Additionally, we reuse the adapted inductive invariants in order to accelerate subsequent verifications of the safety property for reconfigured models, which means in this context that they include an additional process.

Both reuses of previously computed results are extremely successful as shown in numerous experiments. We were able to verify mutual exclusion for all considered parameterized timed systems, i.e., for any number of instances. This ability is a significant benefit over the single verification for a fixed model. As an example, consider a model with one million timed automata, which can be verified by our technique in the context of parameterized timed systems.

Summing up, we present an approach that entirely avoid the need for online verification in the context of parameterized timed systems. Using an a priori verification, it allows the reconfiguration of the system in terms of the number of

processes without raising the need for a new verification. It is competitive and efficient in that it reuses computed inductive invariants.

Reusing previously computed inductive invariants to accelerate the verification of reconfigured models does not only work in the parameterized setting, but also in a general one. We denote this reuse as Feedback-mechanism and apply it in the following work for general models and reconfigurations.

Verification in the Event of General Reconfigurations In the previous setting, the specific kind of systems allows an easy estimation of the effects of a reconfiguration based on the inherent symmetry. For generic reconfigurations, however, the effects can not be estimated in general.

For example, consider a self adaptation of a system in a plant. Although the future behavior mode is determined by this adaptation, it can not be estimated how this behavior will work out considering the entire system in the future. In general, a reconfiguration as little as the change of a timing constant might ultimately lead to unintended behavior.

As the effect on the entire system can not be estimated, we propose a best-guess approach that tries to adapt the inductive invariant as good as possible. Afterwards the adapted invariant is used in the Feedback-mechanism in order to accelerate the verification of the property for the reconfigured model.

Based on the reconfigurations that have been carried out, we adapt the invariant such that it is usable in the new verification and reflects the changes applied to the model. However, due to the IC3 algorithm and the zone computation, this adaptation is extremely limited.

Even so, often the adapted inductive invariant can successfully be applied to accelerate the verification of the safety property for the reconfigured model. We have conducted several experiments to show the value of this technique. Even with a reconfiguration that introduces a violation of the safety property, our feedback mechanism has shown to be of help.

Clearly, the value of this technique in general is limited. Reconfigurations that introduce too much change in the model will ultimately result in a useless reuse of the inductive invariant. Nevertheless, we have introduced and examined a reuse mechanism that is able to accelerate the verification of safety properties for reconfigured timed systems in general. It is successful for many instances, in particular, when considering small reconfigurations for large models as might often be the case in model-based design processes or reconfigurations of timed systems during lifetime. The technique gives an important impulse for online verification of reconfigured models, as might be needed increasingly in the future due to paradigms like Industry 4.0.

In summary, the following contributions are made. We successfully transfer the IC3 algorithm into the domain of timed systems, such that it is competitive. To this end, we propose a combination with the Zone abstraction, which works in a distinct way than other techniques and, thus, gives new impulses and is of value as

a complement to existing approaches. Numerous experiments show its value and practicality.

Using the inductive invariant computed by the proposed approach, we introduce an approach for a priori verification of an entire parameterized timed system. It allows the verification of safety properties for any number of instantiations of a processes in these systems, such that the addition or deletion of a process does no longer raise the need for a new verification. Several experiments are employed to show the practicality, in particular when reusing previously computed inductive invariants for the acceleration of new verification runs for reconfigured models.

Finally, we employ this reuse in a general setting. We present a best-guess approach that allows an efficient verification of safety properties for reconfigured models and, thereby, enables online verification for reconfigurations during lifetime of a system. Our experiments are promising, even though general reconfigurations are hard to handle.

Our work is, thus, of value for the intended use, namely the verification of large, complex models in the context of reconfigurations.

The contributions are presented in the thesis in the following order.

1.3 Thesis Outline

Following this introduction, Chapter 2 introduces the formalism used throughout this thesis. We formally define the employed modeling formalism and its semantics before specifying the considered safety properties. Elaborating the related work, we highlight important work during the 25 years of research in this field. Afterwards, we start with an introduction to SAT-based verification, which includes a detailed section about the employed IC3 algorithm. Its optimizations and related work are given subsequently. We finish the chapter with relevant related work that employs IC3, e.g., for timed verification using the region abstraction.

Chapter 3 contains our work on the combination of the IC3 algorithm with the Zone abstraction. We start with the encoding of our formalism via SMT-formulae. Subsequently giving a detailed explanation of our integration of the zone computation in IC3, we close the chapter with a thorough section showing the practicality and value of our work including numerous experiments.

Chapter 4 presents our work on parameterized timed systems. It starts with a general introduction of parameterized systems, before defining the models considered in this work. To this end, we introduce some restrictions necessary to yield the notion of symmetry we intent and exploit in our approach. We illustrate the incremental workflow that is the heart of our approach, before giving the Termination Theorem that enables our reasoning about the entire parameterized system. Subsequently, we introduce two promising optimizations that accelerate the workflow and increase the applicability of the theorem. Next, we present the numerous experiments we have conducted and their results.

In the next chapter, we propose an extension to the formalism used in our incremental workflow for parameterized timed systems. It significantly improves the

application area without a downside. To this end, we refine some of the definitions. Using a demonstrative example, we illustrate the purpose of the extension and finally show the performance of our technique for this example.

Chapter 6 finally considers the acceleration of verifications for reconfigured models in general. As the effect of a reconfiguration can not be estimated in general, we give a best-guess approach that adapts the inductive invariant where possible. It is injected in the new verification run to hinder the costly rediscovery of some already computed parts of the formula. Our experiments show promising results, although the benefit is extremely dependent on the used model and the executed reconfiguration.

Lastly, Chapter 7 completes this thesis. We sum up the accomplished work and discuss design decisions. Finally, we explain our ideas for future work.

The Appendix contains some more detailed results of our experiments, as well as the proofs for Chapters 4 and 5.

2 Background

Today's industry deploys an increasing quantity of real-time systems. These systems are timing based, e.g., rely on information being communicated in a certain amount of time. With many of these systems being safety critical, the demand for verification techniques considering the timed behavior increased.

Using model-based design and development processes, these systems are modeled and specified properties, in form of the absence of unintended behavior, can be formally verified. To this end, time-based modeling formalisms are required.

From the 1980s on, various efforts have been made to incorporate time in models and verification. Most of the early attempts [AK83; Bur90; AH94] employ a discrete time, approximating continuous time by a fixed step length. Based on a global integer variable counting the number of elapsed time steps, they rely on a special tick transition to increase the time counter.

It was not until the early 1990s that researchers augmented modeling formalisms with capabilities for expressing continuous real time. Nowadays, there exist various real time formalisms for modeling Timed Systems, all of which offer different levels of expressiveness and usability. The list of formalisms includes, but is not limited to, Timed Petri Nets [Ram73], Timed CSP [RR86], Duration Calculus [CHR91], Timed Graph Transformation Systems [HHH10] and others. Pioneering, however, was the work of Alur and Dill in 1990 [AD90; ACD90], who invented the formalism of timed automata. Based on its decidability, it achieved great success and subsequently inspired an entire field of research for improvements, variants, abstractions and tools. The following section presents the formalism of timed automata as used within subsequent chapters.

2.1 Timed Automata

In 1990, Alur and Dill extended Büchi Automata with mechanisms to model and reason about real time [AD90; ACD90]. To this end, they employ real valued

variables that are allowed to be reset individually, but are required to progress simultaneously as time elapses. These variables are called *clocks*, phrasing their utilization for counting time since their last reset. They are formally defined as follows.

Definition 2.1.1 (Clock)**.** A *clock* is a non-negative, real valued variable. The set of clocks is denoted by C. Mapping each clock $x \in C$ to a value $v^c(x) \in \mathbb{R}_{\geq 0}$ is called a *clock valuation* v^c (over C). For a clock valuation v^c and some $\delta \in \mathbb{R}_{\geq 0}$, the *elapse* of δ time units, $v^c + \delta$, is defined as

$$\forall x \in C : (v^c + \delta)(x) = v^c(x) + \delta.$$

A subset $R \subseteq C$ of the clocks can be *reset* to 0, while keeping the remaining valuation, formally defined by

$$\forall x \in C : v^c[R](x) = \begin{cases} 0 & \text{if } x \in R, \\ v^c(x) & else. \end{cases}$$

Each clock characterizes the time that has passed since its last reset. All clocks are initially set to zero keeping track of the time that passes from the start. The clock valuation depicting these initial values is called the *initial clock valuation* v^c_0, formally $\forall x \in C : v^c_0(x) = 0$.

In order to be of use other than bookkeeping, there exist mechanisms to steer and restrict the behavior of the automaton depending on the current clock valuation. To this end, clock constraints are employed. They are control mechanisms describing which clock valuations are allowed at certain points of a run of the automaton.

Definition 2.1.2 (Clock Constraint)**.** Let C be a set of clocks. $\Phi(C)$ is the set of *clock constraints* ϕ defined by $\phi := x \bowtie n \mid (x - y) \bowtie n \mid \phi_1 \wedge \phi_2 \mid true$ with $x, y \in C$, $\bowtie \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$[1]. If a clock valuation v^c satisfies a clock constraint $\phi \in \Phi(C)$, we write $v^c \models \phi$.

The use of clocks and clock constraints allows for a precisely defined automaton behavior in dependence of time. The following example illustrates the intention behind these constraints.

Example 2.1.3. For a given clock $c_1 \in C$, imagine a clock constraint ϕ defined as $\phi := c_1 \leq 1024$. It depicts the restriction that the clock c_1 must not have a value larger than 1024.

In addition, most models demand for extra variables that can be used to store non-timed information. To this end, our formalism includes integer variables that are not subject to time-elapse, but may be used to store and share other information.

[1] As usual, we restrict these bounds to be natural numbers. This is due to the fact, that for bounds in $\mathbb{Q}$ we could upscale the whole timed automaton in order to obtain clock constraints that are solely bound by natural numbers.

Definition 2.1.4 (Integer Variable)**.** Let $\mathcal{IV}$ be a set of *integer variables*. Mapping each integer variable $iv \in \mathcal{IV}$ to a value $v^i(iv) \in \mathbb{Z}$ is called an *integer valuation* v^i. $\Psi(\mathcal{IV})$ is the set of *integer constraints* ψ defined by $\psi := iv \bowtie n | \psi_1 \wedge \psi_2 | true$ with $iv \in \mathcal{IV}$, $n \in \mathbb{Z}$ and $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$. If an integer valuation v^i satisfies an integer constraint $\psi \in \Psi(\mathcal{IV})$, we write $v^i \models \psi$.

For each integer variable, an initial value has to be specified. Thus, the initial integer valuation v^i_0 maps each integer variable $iv \in \mathcal{IV}$ to its initial value $v^i_0(iv) \in \mathbb{Z}$. In contrast to clocks, which can only be reset to zero, we allow more complex assignments for integer variables, defined as follows.

Definition 2.1.5. Let $\mathcal{IV}$ be the set of integer variables. $\Omega(\mathcal{IV})$ is the set of sequences of *integer assignments* ω defined by

$$true \mid iv := n \mid iv := iv + n \mid \omega_1; \omega_2$$

with $iv \in \mathcal{IV}$ and $n \in \mathbb{Z}$. The latter definition creates a sequence of assignments, which are applied from left to right. The resulting integer valuation $v^i[\omega]$ for integer assignment $\omega = \omega_1; \omega_2$ is, thus, defined as $(v^i[\omega_1])[\omega_2]$. For the non-recursive integer assignments, the resulting integer valuation $v^i[\omega]$ is defined as:

$$\forall iv \in \mathcal{IV} : v^i[\omega](iv) = \begin{cases} n & \text{if } \omega = iv := n, \\ v^i(iv) + n & \text{if } \omega = iv := iv + n, \\ v^i(iv) & \textit{else}. \end{cases}$$

Note, that due to assignments only including the addition of a constant or the reset to a constant, each sequence of assignments can be combined into a sequence, in which each integer variable exists at most once as the left hand side of an assignment. The order in these resulting sequences does not matter, since no interdependencies between distinct variables exist. It is possible to extend the concept of assignments using assignment $iv := iv_2$ for $iv, iv_2 \in \mathcal{IV}$. Their combination, however, is more complex since combinations like $iv := iv_2 + iv_3 + 3$ can be created for $iv, iv_2, iv_3 \in \mathcal{IV}$, which in principle still rely on a specified sequence. Thus, we omit such assignments here to provide a better understanding. Note, however, that the concept can be extended to include these assignments by small adaptations to the techniques presented in this thesis.

The following example illustrates the use of integer variables in constraints and assignments.

Example 2.1.6. Let the integer variables $\mathcal{IV} = \{id, cnt\}$ be given. As example consider the integer constraint $\psi := cnt > 1$. It requires the integer variable cnt to have a value larger than 1. Consider the valuation $v^i = \{id = 0; cnt = 0\}$. Applying the sequence of assignments ω defined as $id := 2; cnt := cnt + 1; id := 3$ results in the valuation $v^i[\omega] = \{id = 3; cnt = 1\}$. As can be seen, the leftmost assignment ($id := 2$) is applied before the other ones. In addition, it is obvious that the sequence can be combined into a sequence $cnt := cnt + 1; id := 3$ in which each integer variable exists at most once as the left hand side of an assignment.

Using clocks and integer variables with their respective constraints and assignments, a timed automaton can be defined, whose behavior depends on time and integer values stored in the variables. When modeling huge systems, however, timed automata easily grow large, which complicates the modeling itself. Thus, compositional modeling is often employed that models different parts of a system separately. Timed automata also benefit from compositional modeling, implemented as several timed automata that run in parallel. One of the mechanisms for their interaction is the synchronization of edges in distinct automata, such that they can only be taken simultaneously. To this end, a global set of synchronization labels is defined including an empty label (ϵ) as to allow for non-synchronized edges. Timed automata communicate in a CCS-like fashion [Mil80], where senders of messages ($m!$) synchronize with receivers ($m?$). We define synchronization labels as follows.

Definition 2.1.7. Let Σ be a set of channels. We define the set of *synchronization labels* $\Sigma_{sync} = \{\epsilon\} \cup \{a?|a \in \Sigma\} \cup \{a!|a \in \Sigma\}$.

In the compositional context, we define our integer variables to be globally shared among all timed automata, i.e., they can be used in constraints and assignments within every automaton of the compositional model. The set of clocks, however, is split in disjoint subsets of clocks available globally or locally. The latter, denoted with C^l, is defined within each automaton, whilst the first set of clocks is defined once and provided for all timed automata. The set of global clocks is denoted as C^g and is required to be disjoint to all sets of local clocks.

Finally, we formalize timed automata as used in the subsequent chapters. It resembles parts of the formalism used in Uppaal [LPY95], rather than the original one by Alur and Dill [AD90].

Definition 2.1.8 (Timed Automaton)**.** Let the global set of integer variables $\mathcal{IV}$ and the set C^g of global clocks be given, as well as the global set of channels Σ. A *timed automaton* $\mathcal{A}$ defined over globally shared $C^g, \mathcal{IV}$ and Σ is a tuple $A = (L, l_0, C, \mathcal{IV}, \Sigma, Inv^c, Inv^i, E)$ such that

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- $C = C^l \cup C^g$ is the union of the finite and disjoint sets of local and global clocks with initial valuation v_0^c,
- $\mathcal{IV}$ is the finite set of shared integer variables with initial valuation v_0^i,
- Σ is the finite set of shared synchronization channels,
- $Inv^c : L \to \Phi(C)$ is a total function of clock invariants, s.t. $v_0^c \models Inv^c(l_0)$,
- $Inv^i : L \to \Psi(\mathcal{IV})$ is a total function of integer invariants, s.t. $v_0^i \models Inv^i(l_0)$, and
- $E \subseteq L \times \Sigma_{sync} \times \Phi(C) \times \Psi(\mathcal{IV}) \times \Omega(\mathcal{IV}) \times 2^C \times L$ is the set of edges.

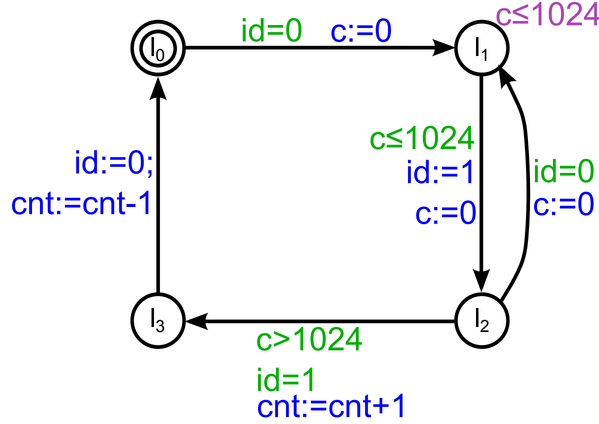


Figure 2.1: Timed Automaton from Uppaal [UPP] modeling a process using the Fischer Mutual Exclusion algorithm

The semantics of a single timed automaton can informally be described as follows. A state of the timed automaton consists of a location l, a clock valuation v^c with $v^c \models Inv^c(l)$ and an integer valuation v^i with $v^i \models Inv^i(l)$. The initial state is given via the initial location, as well as the initial clock and integer valuations.

An edge $(l \xrightarrow{\sigma,\phi,\psi,\omega,R} l') \in E$ can be taken, if the clock and integer valuations of the source state satisfy the constraints ϕ and ψ, i.e., $v^c \models \phi$ and $v^i \models \psi$. When taking the edge, the resulting target state is determined by application of the reset R and assignment ω, such that the invariants of the target location l' are satisfied, i.e., $v^c[R] \models Inv^c(l')$ and $v^i[\omega] \models Inv^i(l')$. An edge is taken instantaneously (without time elapse), whilst time can pass arbitrarily in a location (as long as the location invariant is satisfied). When considering only a single timed automaton, only those edges with synchronization label ϵ can be taken. All other edges require a synchronization partner and are therefor only taken into account when combining several timed automata in a network, which we define later. To illustrate this informal description, we consider the following example.

Example 2.1.9. Consider the timed automaton depicted in Figure 2.1 with no global clocks ($C^g = \emptyset$), no synchronization channels ($\Sigma = \emptyset$), synchronization label ϵ omitted, and two integer variables ($\mathcal{IV} = \{id, cnt\}$). It models a single process executing the Fischer Mutual Exclusion algorithm [Lam87]. Starting with location l_0, all clocks and integer variables are set to 0 in the initial clock and integer valuations. Time may pass and eventually the edge leading from l_0 to l_1 may be taken, if integer variable id still has value 0. Then, the local clock $c \in C^l$ is reset. Taking this edge models that the process checks whether some other process is currently trying to access the critical section. Afterwards some time may elapse, but not more than 1024 time units. Before exceeding that boundary, the process is required to actually announce his request for the critical section. This step is modeled as the edge from location l_1 to l_2, where the process identifier (1) is assigned to the shared variable id.

In addition the clock is reset once more. To gain access to the critical section (location l_3), the process must wait for more than 1024 time units, forcing other processes requesting access to update the value of *id*. If there are no other processes, *id* still contains the same identifier (1) and the process is allowed to take the edge leading from l_2 to l_3. Otherwise, he would have to take the edge leading again to l_1, which may only be taken after the process in the critical section left it and reset the shared variable *id* to 0. Note, that the variable *cnt* counts the number of processes in the critical section.

The semantics illustrated above can be formalized as a transition system as was done, e.g., by Behrmann [Beh+04]. It is denoted as *concrete semantics* due to a state containing only concrete values, meaning a single location, clock and integer valuation.

Definition 2.1.10. Let $A = (L, l_0, C, \mathcal{IV}, \Sigma, Inv^c, Inv^i, E)$ be defined over $C^g, \mathcal{IV}$ and Σ as in Def. 2.1.8. The transition system $TS = (S, s_0, \rightarrow)$ defines the *concrete semantics*:

- $S = L \times \mathbb{R}^{C}_{\geq 0} \times \mathbb{Z}^{\mathcal{IV}}$ is the set of states,
- $s_0 = (l_0, v^c_0, v^i_0) \in S$ is the initial state,
- $\rightarrow \subseteq S \times S$ contains delay transitions $\rightarrow_d$ and edge transition $\rightarrow_e$:
  - $(l, v^c, v^i) \rightarrow_d (l, v^c + \delta, v^i)$ iff $\forall 0 \leq \delta' \leq \delta : (v^c + \delta') \models Inv^c(l)$
  - $(l, v^c, v^i) \rightarrow_e (l', v^{c\prime}, v^{i\prime})$ iff $\exists (l \xrightarrow{\epsilon, \phi, \psi, \omega, R} l') \in E$, s.t. $v^c \models \phi$, $v^{c\prime} = v^c[R]$, $v^{c\prime} \models Inv^c(l')$, $v^i \models \psi$, $v^{i\prime} = v^i[\omega]$, $v^{i\prime} \models Inv^i(l')$.

As can be seen only unsynchronized edges (with synchronization label ϵ) can be taken in a single timed automaton since no synchronization partner is available. However, one of the most convenient aspects in modeling timed automata is the ability for compositional modeling. We call a composed model, consisting of several timed automata running in parallel, a *network of timed automata*. These automata are modeled separately, but interact with each other via clocks, integer variables and synchronized edges. The downside, however, is the exponential blowup of states, as the semantics of such a network equals the product automaton. We formally define the composition of timed automata A_1,...,A_n as follows.

Definition 2.1.11 (Network of Timed Automata)**.** Let C^g, $\mathcal{IV}$ and Σ be given, as well as the timed automata A_1 to A_n defined over them. For distinction, their parts are marked with subscripts such that $A_j = (L_j, l_{0j}, C_j, \mathcal{IV}, \Sigma, Inv^c{}_j, Inv^i{}_j, E_j)$. All sets of clocks $(C^g, C^l_1, \ldots C^l_n)$ are required to be mutually distinct. The product automaton defining the *network of timed automata* $NTA = \langle A_1, ..., A_n \rangle$ is defined over $C^g, \mathcal{IV}$ and Σ as $A = (L, l_0, C, \mathcal{IV}, \Sigma, Inv^c, Inv^i, E)$ with

- $L = L_1 \times ... \times L_n$ with initial state $l_0 = (l_{01}, ..., l_{0n}) \in L$,
- $C = C^g \cup C_1^l \cup ... \cup C_n^l$ with initial valuation according to local valuations v_0^c,
- $Inv^c(l_1, ..., l_n) = Inv^c{}_1(l_1) \wedge ... \wedge Inv^c{}_n(l_n)$,
- $Inv^i(l_1, ..., l_n) = Inv^i{}_1(l_1) \wedge ... \wedge Inv^i{}_n(l_n)$,
- E is defined as
  - $\forall i \in \{1, ..., n\} : ((..., l_i, ...) \xrightarrow{\sigma, \phi, \psi, \omega, R} (..., l_i', ...)) \in E$

    if $(l_i \xrightarrow{\sigma, \phi, \psi, \omega, R} l_i') \in E_i$
  - $\forall i \neq j \in \{1, ..., n\} : ((..., l_i, ..., l_j, ...) \xrightarrow{\epsilon, \phi, \psi, \omega, R} (..., l_i', ..., l_j', ...)) \in E$

    if $(l_i \xrightarrow{a!, \phi_1, \psi_1, \omega_1, R_1} l_i') \in E_i$ and $(l_j \xrightarrow{a?, \phi_2, \psi_2, \omega_2, R_2} l_j') \in E_j$
    with $\phi = \phi_1 \wedge \phi_2$, $\psi = \psi_1 \wedge \psi_2$, $\omega = \omega_1; \omega_2$, $R = R_1 \cup R_2$.

The product automaton contains a non-synchronized edge (with synchronization label ϵ) wherever two edges have successfully been synchronized, i.e., that edge is allowed to be taken. Note, that the assignments of the sender edge ($m!$) are applied before those of the receiver edge ($m?$). Additionally, it still contains the original edges (see the first bullet point defining E above) including those requiring a synchronization partner. These are included for further composition, but are not allowed to be taken as defined in the concrete semantics in Definition 2.1.10 since they contain a synchronization label distinct from ϵ.

Verification of properties for timed automata was done right from the start supported by decidability results of Alur and Dill in 1990 [AD90]. One of the most important and interesting verification question is reachability of a state s. It asks whether there exists a finite number of transition steps leading from an initial state to s. This verification question is challenging not only due to the infinite transition system introduced by the real valued clocks, but also due to the state explosion problem in general.

The notion of reachability allows for the definition of safety properties, which specify that something bad should never happen. To this end, *error state specifications* are defined that describe the bad situation. The safety property holds true, i.e., the model is safe w.r.t. the safety property, if no error state is reachable. The verification of safety properties is of fundamental importance [Hal93] as many verification questions of interest can be expressed as safety properties.

We formally define reachability and safety properties as follows.

Definition 2.1.12 (Reachability)**.** Let a timed automaton A be given with concrete semantics $TS = (S, s_0, \rightarrow)$. A state $s \in S$ is *reachable*, if there exists a finite number of transitions leading from the initial state to s, formally $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_n \rightarrow s$.

When considering some states as bad, or *error states*, we employ reachability to define safety properties that require these error states to be unreachable. To this end,

we define *error state specifications* that are able to efficiently characterize error states. We employ integer and clock constraints, which allows us to specify more than a single concrete state at once.

Definition 2.1.13 (Error State Specification)**.** Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11 with concrete semantics $TS = (S, s_0, \rightarrow)$. An *error state specification* is an abstract formalization of a set of undesired states. The set of error state specifications is defined as $ERR = ((L_1 \cup \{*\}) \times \cdots \times (L_n \cup \{*\})) \times \Phi(C) \times \Psi(\mathcal{IV})$. Each error state specification $err = (\bar{l}, \phi, \psi) \in ERR$ includes a clock and an integer constraint and additionally a (partial) location vector $\bar{l} \in (L_1 \cup \{*\}) \times \cdots \times (L_n \cup \{*\})$ specifying at most one location for each timed automaton. The element $*$ stands for an undefined location. To refer to specific locations in the vector, we denote the location specified for automaton A_i as $\bar{l}[i]$. A state $s = ((l_1, ..., l_n), v^c, v^i) \in S$ is included in an error state specification $err = (\bar{l}, \phi, \psi)$, denoted $s \models err$, iff

- $\forall i \in \{1, \ldots, n\} : \bar{l}[i] = *$ or $\bar{l}[i] = l_i$,
- $v^c \models \phi$,
- $v^i \models \psi$.

The states that satisfy an error state specification are called *Error States*.

The safety properties used within this thesis specify that no reachable state is allowed to be an error state. We formally define it as follows.

Definition 2.1.14 (Safety Property)**.** Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11 with concrete semantics $TS = (S, s_0, \rightarrow)$. Using the notation of of the LTL-operator G, we define a *safety property* $\rho := G(\neg err_1 \wedge \neg err_2 \ldots)$ to be the conjunction of the negations of error state specifications err_1, err_2, ... It holds true, when no reachable state in S is an error state, formally $\forall s \in S : s \textit{ is reachable} \Rightarrow (s \not\models err_1 \wedge s \not\models err_2 \wedge \ldots)$. Then, we say the safety property is invariant. Otherwise, at least one state $s \in S$ is reachable via a finite number of transitions $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n \rightarrow s$ and satisfies one of the error state specifications $s \models err_i$. This path violates the safety property and is, thus, called a *counterexample trace* for the given safety property.

Usually, the verification of properties for networks of timed automata is done *on-the-fly* without the computation of the product automaton, due to the enormous increase in size of states. However, it still needs to take into account the interdependencies of the individual automata resulting in enormous effort. Thus, since the beginning diverse attempts have been made to establish the practicality of verification for timed automata.

2.1.1 Decidability and Abstractions

The feasibility of verifying reachability properties for timed automata was established in 1990 by Alur and Dill [AD90]. They proved decidability, which is not intuitive due to the real valued clocks. To this end, they proposed a finite abstraction of the clock valuations based on the observation that some of them have equal characteristics. Given the fact that clocks are only compared to integer constants within clock constraints the actual fractional part of a clock value does not matter. This fractional part is only of interest to identify which clock will change its integral part first. Taking into account that time progresses simultaneously for all clocks, Alur et al. described the region abstraction, which partitioned the clock valuations into equivalence regions.

Definition 2.1.15. Let A be a given timed automaton as in Def. 2.1.8. For every clock $x \in C$ let n_x be the largest constant with which x is compared to. Two clock valuations v^c and $v^{c'}$ are in the same *region*, iff:

- $\forall x \in C : \lfloor v^c(x) \rfloor = \lfloor v^{c'}(x) \rfloor$ or $v^c(x) > n_x \wedge v^{c'}(x) > n_x$,
- $\forall x, y \in C$ with $v^c(x) \leq n_x$ and $v^c(y) \leq n_y$: $fract(v^c(x)) \leq fract(v^c(y))$ iff $fract(v^{c'}(x)) \leq fract(v^{c'}(y))$,
- $\forall x \in C$ with $v^c(x) \leq n_x$: $fract(v^c(x)) = 0$ iff $fract(v^{c'}(x)) = 0$.

with $fract$ meaning the fractional part of the value.

Since the number of clocks and the largest constants are fixed within a timed automaton, the number of regions is finite [AD94]. Thus, decidability of reachability was shown since every reachability question for a timed automaton can be decided via this finite abstraction.

Unfortunately, the number of regions grows exponentially with the size of constants and number of clocks [AD94]. Hence, decidability via region abstraction is an interesting theoretical result, but not of substantial value for practical verification purposes. This drawback is tackled by the zone abstraction.

Definition 2.1.16. A *Zone* Z is a convex set of clock valuations, specified as a conjunction of clock difference constraints $x_i - x_j \bowtie n$ with $x_i, x_j \in C \cup \{x_0 = 0\}, \bowtie \in \{<, \leq\}$ and $n \in \mathbb{Z}$.

Each zone is a convex union of regions and can be described via upper and lower bounds on single clocks and clock differences. It can efficiently be stored as a Difference Bound Matrix (DBMs) [Dil90], which allows for an efficient computation of the zones of predecessor or successor states.

Thus, it gives rise to a significantly coarser symbolic transition system that is of practical relevance. It is used within many algorithms and tools.

In the following, we survey the research that was done to establish practicality of the verification for timed systems.

2.1.2 Related work

We start with a discussion of some algorithms for the verification of safety properties for timed automata, followed by special data structure and tools. The same sequence was also used in a survey paper by Yovine in 1998 [Yov98].

There exist various approaches for the reachability analysis of timed automata. Given that standard exhaustive exploration on the region abstraction is not efficient, some tools apply digitization to completely abstract away the time domain. To this end, a finite set of representatives is computed to represent each region [Göl+94]. In general, digitization enables the use of untimed verification algorithms, which are often more sophisticated than simple exhaustive search. For some restricted subclasses of timed automata, a BDD-based fixpoint analysis [Bey01] has proven to be specifically successful. However, these techniques still suffer from a search space exponentially in the size of the used constants in the model.

Hence, coarser abstractions were sought after. One way is to find an equivalence relation that is smaller and, thus, better suited for verification. There exist various approaches searching the minimal finite transition system equivalent to the region graph up to time-abstracting bisimulation [Yov98; TY96; Alu+92]. Other approaches try to minimize the timed automaton itself, while maintaining a similar notion of bisimulation [DY96].

When considering basic forward or backward search, the utilization of clock constraints to describe sets of clock valuations comes naturally. Their conjunction, denoted a *Zone*, describes a convex set of clock valuations. Due to being closed under time elapse and transition steps, zones are optimally suited for exhaustive exploration of the search space. Furthermore, the resulting zones are efficiently computable.

All these different ways of abstraction show the non-triviality of timed verification. In the end, the success of the abstraction strongly depends on its combination with clever verification algorithms and data structures.

Most often, in particular in combination with the region or zone abstraction, a basic exhaustive search is performed. Starting from the initial state, all successor states are computed and added to the queue of unprocessed states. This queue is worked off until there are no states left that have not been processed before. Upon discovery, every single state is checked for violation of the safety property. Exhaustive search can also be done in a backwards-manner, where states are discovered starting from the error states. Especially when using the zone abstraction, this basic algorithm provides for an easy and efficient analysis. Stored in DBMs, the memory usage is passable, while providing easy mechanisms to compute successor-, predecessor- or time-elapse-zones [Dil90; DT98]. More sophisticated approaches try to minimize memory usage by exploiting the fact, that most often some clock constraints in a DBM are redundant. There exist various publications on how to compute and maintain a minimal list of such constraints [YPD94; Lar+97]. Despite these optimizations in memory efficiency, one major problem of the exhaustive search approach in combination with zones remains difficult. Each discovered state is stored to be able to tell whether a state has already been explored. A short check if a new state

is already stored suffices to stop further exploration on this state. Considering zones, however, a simple check whether a zone (for a specific location and integer valuation) has already been stored is inefficient. The reason is that the zone could be covered by the union of several previously explored zones. Hence, a suitable coverage check might be needed that stops the exploration of zones covered by the union of previously explored zones. With zones not being closed under union, several data structures have been proposed offering efficient mechanisms to store and maintain non-convex unions of zones with a fast inclusion check. However, in the end these data structures only provide for a trade-off reducing the enormous amount of memory needed for storing each single zone, but (most often) slightly increasing the runtime.

Most of these non-convex data structures are build as decision diagrams. Proposed in 1997, numerical decision diagrams were the first data structure aiming at a small representation of unions of zones [Asa+97]. The approach is based on discretization of time and employs binary decision diagrams [Lee59; Ake78] for storing a bitwise encoding of the time values. Thus, it heavily depends on the size of the encoded constants rendering it inefficient for the verification of safety properties for most timed automata. The same disadvantage is observed in region encoding diagrams [Wan00]. They represent a region by the integer parts of the clock values and the ordering of the fractional parts. Given all the necessary algorithms for data manipulation, it is suitable as a decision diagram, but as mentioned lacks the capability to handle large constants. Later approaches are not tied to a discretized time or region encoding and are, thus, independent of the size of timing constants. Larsen et al. introduced clock difference diagrams (CDD) in 1998 [Lar+98; Beh+99] pursuing the goal of a data structure for completely BDD-based verification approaches. Clock difference diagrams are defined over the real valued difference of clock values. In addition, they branch with regard to intervals of the reals, while previous techniques branched only for single values. This makes them significantly less dependent on the size of constants. Designed to share redundant substructures, clock difference diagrams are well suited for space-efficient coverage decisions. Other approaches were not optimized for coverage checks, but more on space efficiency. Reduced clock restriction diagrams [Wan01a] utilize the small number of constraints needed to represent a zone [Lar+97]. Given such a minimal constraint system, a compact representation is achieved, which lacks the capability for efficient determination of zone containment. Thus, Wang enhanced his approach resulting in cascading clock restriction diagrams [Wan02; Wan03]. The technique is based on the cascading form of zones, which may require more constraints than the reduced form, but still less than a complete DBM.

All these data structures aim to reduce the memory required for verification, mostly as a trade-off to a slightly increased runtime. While the presented data structures were able to reduce memory consumption in general, they still run out of memory easily when verifying properties for large models. Despite these shortcomings, they have been applied in various verification tools, which we show in the following. The implementations of all the different concepts and algorithms

have been extremely successful in the verification tasks both for academic and real world models.

Over the last 20 years, research has led to the creation of several tools, most of which are not actively maintained any more. The concepts used in these tools are very diverse resulting in different modeling and verification capabilities. In the following, we give a brief survey.

The development of the first tools started in the early 1990s. Kronos [Daw+96; Boz+98] uses several of the techniques presented above. It implements symbolic analysis, as well as time-abstracting bisimulation. It employs several data structures, e.g., DBMs and NDDs and triggered a lot of research. The last release of Kronos has been in 2002 and since then it has not been developed any further. Another early tool for the verification of safety properties for timed automata is called Uppaal [LPY95; LPY97] developed by researchers from the universities of Uppsala and Aalborg. It is actively developed and maintained down to the present day. Given its graphical user interface it offers an easy mechanism for modeling and verification, which might be one of the reasons for its success even in commercial applications. The other reason for its huge success is the efficient verification, which has been applied to several interesting real-world studies [Hav+97; LPY98; Ben+96]. It is based on constraint solving with DBMs represented as minimal constraint systems. Furthermore, it can apply CDDs and several options for optimization and approximation. With more than 15 years of development [Beh+11], it includes a lot of great ideas and has become the most well known tool and also quasi-standard for the verification of safety properties for timed automata.

Thus, later tools most often had to compare their performance with Uppaal. The tool Red [Wan01b] that was first implemented using region encoding diagrams was developed in the early 2000s. Later its verification engine was based on the presented clock restriction diagrams. The development of Red, however, was discontinued in 2003.

Other tools are based on other techniques. A perfect example for this sort of diversity is the tool Rabbit [BR00], which is based on digitization in order to apply BDD-based techniques. Thorough investigations into the best variable orderings, and adjusted BDD-based algorithms showed a great performance. However, the approach is still unsatisfying as it highly depends on the size of time constants.

In the last few years, two new tools have been developed that are capable of verifying timed automata. Synthia [PEM11] is one of these tools, but has only been actively developed for 2 years, being discontinued in 2011.

The tool PAT [Sun+09], however, is developed and maintained starting in 2009 up to now. It includes a wide variety of verification techniques including digitization and BDD-based algorithms.

In summary, there has been an enormous amount of research in the field of timed automata over the last 25 years. Many techniques have been investigated and implemented in various tools. However, today only two tools are left that are actively maintained. These two represent the most successful verification approaches, as Uppaal heavily relies on zone based verification and PAT utilized digitization

with BDD-based verification. With the latter being dependent on the size of time constants, it is conceptually less relevant. Thus, we will examine the techniques presented in the following chapters in comparison with Uppaal. Even after 15 years of research, Uppaal can easily be pushed to its limits. As a reason for failed verification attempts, most often its enormous need of memory has to be named. In consequence, for this thesis we are interested in techniques with better memory efficiency.

2.2 SAT- & SMT-Solving

In addition to the techniques presented before, there exist various concepts employing SAT- or SMT-solvers for verification. We present a short overview on SAT- and SMT-solving that might be helpful to understand Chapters 3 to 6.

2.2.1 SAT-Solving

The *boolean satisfiability problem* is defined as follows:

Definition 2.2.1. Let a boolean formula be given. The *boolean satisfiability problem* (SAT) asks whether there exists an interpretation, meaning a consistent assignment of truth values $true$ or $false$ to the propositional variables, that evaluates the formula to $true$. It is called a *satisfying* interpretation.

If there exists a satisfying interpretation, the formula is said to be *satisfiable*, otherwise it is called *unsatisfiable*.

Various tools, called SAT-solvers, exist that try to answer the boolean satisfiability problem. The problem is fundamental for complexity theory and has been subject to research for many decades. In 1971 it was proven to be $NP-complete$ [Coo71], however, a lot of research on efficient algorithms and heuristics build up the practicality of today's tools, e.g., MiniSAT [ES05], PicoSAT [Bie08] or others [Bie12; AS12; LP10].

Most of the tools are based on conflict-driven clause learning algorithms building on the DPLL approach, named after the researchers Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland [DP60; DLL62]. The algorithm assigns a truth value to literals and afterwards propagates them to simplify the remaining clauses. Upon discovery of a conflict, the algorithm learns a clause representing the cause of the conflict and backtracks. Most SAT-solvers require the boolean formula to be in conjunctive normal form, which is not a problem due to algorithms for satisfiability-preserving transformations [Tse83; PG86] in polynomial time.

In 1999 the usage of SAT-solvers found its way into the domain of untimed formal modeling and verification for finite state transition systems. The reason was that earlier techniques such as explicit model checking or BDD-based approaches reached their limits due to scalability issues. Biere et al. [Bie+99] introduced the concept of *Bounded Model Checking* in order to check transition systems for counterexample traces of bounded length. The technique relies on an encoding of a bounded unrolling of the transition relation as a boolean formula to check for an error path of

fixed length in the model. The formula is issued to a SAT-solver. In case the solver returns *unsatisfiable*, there exists no such path. Otherwise, the returned satisfying interpretation embodies the found error path.

Due to the success of Bounded Model Checking, the application of SAT-based techniques for formal verification advanced. Concepts were presented that were not limited to a bounded exploration of the state space, e.g., by McMillan [McM02]. Other work extended the capabilities of bounded model checking by computing *Craig Interpolants* from unsatisfiability proofs [McM03]. Integrated in an iterative fixpoint computation, these are used to compute an over-approximation of the reachable states and prove the absence of counterexample traces.

In addition, SAT-solving is applied also in abstraction-based reasoning, called counterexample-guided abstraction refinement (CEGAR) [Cla+02; Cha+02]. To this end, abstract counterexamples are checked for spuriousness using SAT-queries and possible refinements are done based on the results of the solver.

Further applications of SAT-solving include approaches relying on *induction*. We discuss these techniques in more detail in Section 2.3.

Many of the above techniques have been transfered to the domain of timed systems. However, with time being represented by unbounded real valued clocks, an encoding using boolean variables is non-trivial. Thus, most approaches rely on an extension of boolean satisfiability, which we present in the following.

2.2.2 SMT-Solving

The *Satisfiability Modulo Theories*-problem is a generalization of the boolean satisfiability problem. It denotes the search for a satisfying interpretation for a formula in first-order logic, where some symbols have fixed interpretations determined by background theories. There exists a large variety of such theories, some of which are specifically designed to reason about data structures like bit vectors or arrays. In the context of this work, however, we are most interested in the theory of reals, which enables the SMT-solver to handle real-valued variables in combination with operators, e.g., addition and multiplication. Furthermore, the theory of integers may be employed to encode integer variables. The use of these theories results in an easy and natural way to encode timed systems as logical formulae.

These can be issued to an SMT-solver asking for satisfiability. If the SMT-formula is satisfiable, the satisfying interpretation, also called *SAT-Model* is returned. Otherwise, the formula is unsatisfiable and the solver may return an *UNSAT-Core*, which identifies those parts of the formula that take part in its unsatisfiability.

Various SMT-solvers exist [DB08; Dut14; CHN12; Cim+13b; Bar+11] and although most of them accept the same input format, standardized as SMT-Lib [BST10], they differ in their capabilities and algorithms. Most modern SMT-solvers use the *lazy* approach, which is based on an extension of the DPLL-algorithm, called DPLL(T) [NOT06]. In general, a run of a SAT-solver determines candidate assignments of the boolean structure whose satisfiability would result in the satisfiability of the whole formula. Afterwards, these assignments are checked by theory solvers

whether they comply with the theories. A close integration of those theory solvers within the DPLL-algorithm has shown as successful basis for SMT-solvers. Further improvements and the rich use of heuristics led to a huge success of these tools.

Although their efficiency is a bit behind the one of SAT-solvers, SMT-solvers are employed efficiently in many domains and are capable of solving very diverse SMT-instances. Given their expressiveness, they are an easy and natural way of encoding timed systems.

Thus, many of the SAT-based model checking approaches have been transferred to the domain of SMT, including their employment for the verification of safety properties for timed automata. Bounded Model Checking has been one of the first employments of SMT-solvers for the verification of safety properties for timed automata. Starting in 2002, a large number of works has been published from a variety of researchers [PWZ02; Aud+02; Sor03; KJN12a]. All these works use SMT-encodings of paths in the timed systems, but differ in nuances and research focus. SMT is the enabling technique for these contributions, as they employ the theory of reals to encode the real-valued clocks.

Interpolation-based reasoning has also been transferred to be used with infinite systems [McM05] like timed automata. However, apart from the mentioned approach for infinite systems in general, it was not in the focus of research.

In addition to the presented work, a lot of techniques based on induction have been transferred from the SAT domain to SMT-solving.

2.3 Induction based Reasoning

Induction is a proof-principle of high interest. Checking whether a property is inductive or not is efficient and trivial since it does not involve an unrolling of the transition system.

The following explanations rely on the notion of a transition system, which may be infinite, as can be seen in Definition 2.1.10.

Definition 2.3.1 (Transition System)**.** A transition system $A = (S, I, T)$ is defined over a set of states S with initial states $I \subseteq S$. A transition relation $T \subseteq S \times S$ defines the passage from state to state. We denote a transition system to be a *finite state transition system* (FTS), whenever S is finite.

We define induction and related concepts using SAT- or SMT-encodings. These encodings usually employ distinct sets of variables $\underline{x}, \underline{x'}, \underline{x''}, \ldots$ to encode the different states occurring on a path. We let $\|I(\underline{x})\|$ and $\|\rho(\underline{x})\|$ denote the encodings of the initial states, and those states that satisfy the property ρ, respectively. Additionally, $\|T(\underline{x}, \underline{x'})\|$ encodes the transition relation, where the primed variables refer to the next state as explained above. Using these encodings within queries to SAT- or SMT-solvers, their unsatisfiability can be used to check whether certain properties like induction are satisfied.

Definition 2.3.2 (Induction)**.** Let a transition system A be given as in Definition 2.3.1. A safety property ρ is *inductive*, if the following two characteristics are met.

- **Initiation:** $\|I(\underline{x})\| \wedge \neg\|\rho(\underline{x})\|$ is unsatisfiable
- **Consecution:** $\|\rho(\underline{x})\| \wedge \|T(\underline{x}, \underline{x'})\| \wedge \neg\|\rho(\underline{x'})\|$ is unsatisfiable

Initiation requires all initial states to satisfy the safety property. In order to check this requirement a query is issued encoding an initial state violating the safety property. Initiation holds true if the query is unsatisfiable. Consecution states that all successor states of all states satisfying the safety property must also satisfy the safety property. It is also encoded as a query checked for satisfiability in order to identify whether it holds. In combination, these two properties specify that all reachable states satisfy the safety property, meaning it is invariant and the model is safe w.r.t. the safety property. In consequence, induction is an efficient way of verifying safety properties for systems. Sadly, most safety properties are not inductive and, thus, the presented approach will not work. In case the property is not inductive, one of the two queries will be satisfiable, meaning the respective property does not hold. The Initiation-formula being satisfiable denotes the existence of an initial state violating the safety property, which is therefore not invariant. However, in case of a satisfiable Consecution-formula no conclusion can be made whether the safety property is invariant or not. The reason is that there exists at least one pair of successive states in which the successor violates ρ while the predecessor satisfies it. If this pair of states is reachable, the model would not be safe, otherwise if no such pair is in the set of reachable states, it would be safe. Thus, in such a case, induction is not strong enough to verify safety properties, which is the case for the majority of safety properties.

In order to decide the above question of reachability, a generalized version of induction can be employed. By considering several transition steps instead of only a single one, the requirements are strengthened, possibly ruling out the above concerns. This generalized version of induction is known as $k - Induction$.

Definition 2.3.3 (k-Induction)**.** Let a transition system A be given as in Definition 2.3.1. A safety property ρ is *k-inductive*, if the following two characteristics are met.

- **k-Initiation:** $\|I(\underline{x^1})\| \bigwedge_{i=1}^{k-1} \|T(\underline{x^i}, \underline{x^{i+1}})\| \wedge \neg \bigwedge_{i=1}^{k} \|\rho(\underline{x^i})\|$ is unsatisfiable
- **k-Consecution:** $\bigwedge_{i=1}^{k} \|T(\underline{x^i}, \underline{x^{i+1}}\| \bigwedge_{i=1}^{k} \|\rho(\underline{x^i})\| \wedge \neg\|\rho(\underline{x^{k+1}})\|$ is unsatisfiable

As can be seen, $k - Induction$ using $k = 1$ is the same as the usual induction rule defined above in Definition 2.3.2. Otherwise, it relies on the use of more than a single transition step which strengthens the properties. There exist algorithms that verify models by means of $k - Induction$, while k is incremented until either the model is proven to be safe or a counterexample occurs. These techniques have been proposed in 2000 [SSS00; BC00] and make use of additional optimizations, e.g., a loop-free encoding of paths, to ensure completeness.

Induction-based reasoning has also been employed for the verification of safety properties for timed systems. Using an SMT-encoding that employs the region

abstraction to ensure loop-free paths, Kindermann et al. [KJN12b] proposed a complete k-Induction-based approach for timed systems in 2012. Several years earlier, in 2003, de Moura et al. also employed $k - Induction$ in combination with simulation relations [DRS03]. In addition, their approach utilized the failed Consecution-property in order to compute a small formula that strengthens the safety property.

In the untimed domain, the idea of strengthening has also been used in combination with the usual induction rule ($1 - Induction$). The approach of Bradley and Manna [BM07] employs relative induction.

Definition 2.3.4 (Relative Induction)**.** Let a transition system A be given as in Definition 2.3.1. Given a property ψ, the safety property ρ is *inductive relative* to ψ, if the following two characteristics are met.

- **Initiation:** $\|I(\underline{x})\| \wedge \neg\|\rho(\underline{x})\|$ is unsatisfiable
- **Relative Consecution:** $\|\psi(\underline{x})\| \wedge \|\rho(\underline{x})\| \wedge \|T(\underline{x}, \underline{x'})\| \wedge \neg\|\rho(\underline{x'})\|$ is unsatisfiable

Their approach utilizes predecessor states that violate Consecution. These states are blocked either in the safety property or added to a strengthening assertion. The distinction is based on relative inductiveness denoting if the states non-reachability is essential for the verification of the safety property or the state only needs to be blocked in order to not find it again. In addition, they propose a technique called generalization, which is used to block a set of states rather than a single state.

We will see this generalization procedure in more detail in the next section. It has led to the development of the algorithm $IC3$, which is the basic algorithm used within this thesis.

The $1 - Induction$ rule and the techniques based on it are well suited for verification, in particular in combination with reconfigured models, as considered in the next chapters. Due to the two characteristics, inductive properties can easily be verified. They denote an over-approximation of the set of reachable states and may be utilized in this spirit as we will see later. One of the primary efficiency reasons is the fact that they do not need an unrolling of the transition relation, as they consider only a single transition step. The activity of strengthening non-inductive properties, however, is expensive and, thus, efficient algorithms for this task are interesting.

As mentioned before, one such algorithm is based on the presented approach by Bradley and Manna. We explain it in the following. Since the approach, as well as the rest of the thesis only reasons about a single transition step, we declare the following abbreviations for better readability of the formulae. We omit the set of variables $\underline{x}$ and $\underline{x'}$ over which the formulae are defined and mark those that reason about the next state as primed formulae $\|.\|'$. Thus, all formulae reasoning about a single state are now written as, e.g., $\|I\|$ and $\|\rho\|$, when reasoning about the current state, and written as $\|I\|'$ and $\|\rho\|'$, when reasoning about the next state. The encoding $\|T(\underline{x}, \underline{x'})\|$ of the transition relation always reasons about the transition from a current to a next state and is, thus, abbreviated as $\|T\|$. These notations are used throughout the rest of the thesis.

2.4 IC3

In 2011, Aaron Bradley introduced a very successful technique capable of computing inductive strengthenings of safety properties [Bra11]. Originally called *Incremental Construction of Inductive Clauses for Indubitable Correctness* or short *IC3*, it was instantly recognized as an interesting and promising approach, especially after its first implementation achieved third place in the hardware verification competition 2010 [BC10].

Later implementations became even more successful and ensured a constant attention of researchers. Most implementations nowadays are based on the work of Een et al. [EMB11], proposing several optimizations of the original algorithm. His version of the algorithm is called *Property Directed Reachability*, which we will use interchangeably with IC3 since they rely on the same basic principles. The understanding of these principles was enforced by numerous publications [SB11; Bra12b; Bra12a] explaining the development process of IC3 and its fundamentals. These thorough descriptions enabled the transfer of the algorithm and some of its principles to other research areas, generated interest in optimizations and, in general, provided for a lot of interesting research. In the following, we give a brief overview over IC3, mostly following the representation given by Een et al. [EMB11].

2.4.1 Algorithm and Explanation

IC3 is an induction-based algorithm for verifying safety properties for finite state transition systems. It heavily relies on relative induction in order to compute a strengthening of the safety property that is inductive, or a counterexample trace. Being based on SAT-solvers, part of its efficiency is due to the fact that the queries to the solver encode at most one transition step.

IC3 is a completely SAT-based algorithm originally designed and applied to hardware verification. Thus, it verifies safety properties for finite state transition systems, which can be used to describe sequential circuits.
In general, IC3 incrementally builds and refines sets of states that over-approximate k-step reachability, meaning the i-th set includes at least all those states that are reachable in i transition steps. During the run of the algorithm, these sets are refined to exclude states found along a path to an *error state*, a state that violates the safety property.

During this process, four special properties are preserved for the sets of states constructed by IC3. These sets are called frames and denoted F_0 to F_k, where F_k is called the frontier, as it is the largest set. The algorithm ensures the following four properties:

1. $\|I\| \land \neg\|F_0\|$ is unsatisfiable,
2. $\forall i \in \{0, \ldots, k-1\} : \|F_i\| \land \neg\|F_{i+1}\|$ is unsatisfiable,
3. $\forall i \in \{0, \ldots, k-1\} : \|F_i\| \land \|T\| \land \neg\|F_{i+1}\|'$ is unsatisfiable,
4. $\forall i \in \{0, \ldots, k\} : \|F_i\| \land \neg\|\rho\|$ is unsatisfiable.

The smallest frame F_0 is required to include at least all initial states (I). In practice, in most implementations it includes exactly all initial states, meaning $F_0 = I$. Furthermore, the sequence of frames is required to be non-decreasing. Every frame F_{i+1} includes at least all the state that are included in the next smaller set F_i (Property 2). In practice, this is ensured by sharing the same clauses. Frames are stored as boolean formulae in conjunctive normal form (CNF), meaning a conjunction of clauses. Thus, an added clause refines (strengthens) a set of states. Property 2 is ensured by the clauses of F_{i+1} being a subset of the clauses of F_i. In addition, each set over-approximates the successor states of the next smaller set. This is depicted as Property 3, in which the primed formula $\|F_{i+1}\|'$ denotes that the next state, after the transition step was applied, has to be an element in set F_{i+1}. Lastly, each frame F_i includes only states that satisfy the safety property. Thus, each such set is a strengthening of the safety property. The goal of the algorithm, in order to ensure that the safety property holds, is to find such a strengthening that is inductive.

Definition 2.4.1 (Inductive Strengthening)**.** Let a finite transition system be given with a safety property ρ. F is an *inductive strengthening* of ρ if and only if it is inductive and strengthens ρ:

- **Initiation:** $\|I\| \wedge \neg\|F\|$ is unsatisfiable,
- **Consecution:** $\|F\| \wedge \|T\| \wedge \neg\|F\|'$ is unsatisfiable,
- **Strengthen:** $\|F\| \wedge \neg\|\rho\|$ is unsatisfiable.

An inductive strengthening is found when two successive sets F_i and F_{i+1} are equal. As explained, Property 4 ensures that both sets are a strengthening of the safety property. Additionally, Property 3 in combination with $F_i = F_{i+1}$ ensures consecution and Properties 1 and 2 provide for the initiation.

Thus, IC3 proves the safety property to hold when two of its sets F_0 to F_k become equal. Otherwise, is runs until it finds a counterexample trace leading from an initial state to a state violating the safety property. Below, we describe the algorithm following the presentation in the original paper. The main loop is listed in Listing 2.1.

First, the algorithm searches for counterexample traces of length 0 or 1 (lines 2 and 4), meaning an error state is initial or reachable from an initial state via a single transition step, respectively. The search is executed by issuing two queries to the SAT-solver. If one of them is satisfiable, a counterexample exists and the safety property is obviously not invariant. Otherwise, the first two sets are created and initialized to $F_0 = I$ and $F_1 = \rho$, as can be seen in line 6. Since there exist no 0- and 1-step counterexamples, these two frames satisfy the 4 properties explained above.

Listing 2.1: Main loop of IC3 algorithm

```
1  IC3(){
2      if (‖I‖ ∧ ¬‖ρ‖ satisfiable)
3          return 0-length counterexample;
4      if (‖I‖ ∧ ‖T‖ ∧ ¬‖ρ‖' satisfiable)
5          return 1-length counterexample;
6      ‖F0‖ := ‖I‖, ‖F1‖ := ‖ρ‖, k=1;
7      while(true){
8          if(!strengthenClauses(k))
9              return counterexample trace;
10         propagateClauses(k);
11         if(∃i < k, s.t. Fi = Fi+1)
12             //Safety Property holds
13             return Fi; //ind. strengthening
14         ‖Fk+1‖ := ‖ρ‖, k++;
15     }
16 }
```

The algorithm proceeds in a loop as follows. In the subroutine *strengthenClauses*, it checks for predecessor states on a path leading to an error state and excludes these in a blocking phase. Afterwards, a propagation phase provides for an accelerated spread of learned clauses. The loop ends either when two successive frames are found to be equal after the propagation or if the strengthenClauses routine was unable to refine the frames while preserving the 4 properties. In the first case, an inductive strengthening has been found as explained before and the safety property is proven to be invariant. In the latter case, a counterexample trace has been found as we will see in the following. Listing 2.2 shows the *strengthenClauses* function.

Listing 2.2: Algorithm to strengthen the frames in IC3

```
17 strengthenClauses(int k){
18     while(‖Fk‖ ∧ ‖T‖ ∧ ¬‖ρ‖' satisfiable){
19         //using the satisfying interpretation
20         state s = extract predecessor
21         if(!blockCTIs({(s,k)}))
22             //found counterexample trace
23             return false;
24     }
25     return true;
26 }
```

IC3 checks whether the frames are refined enough to add a new one, as the creation of a new frame F_{k+1} (line 14) has to preserve the 4 properties. The new frame is created as $F_{k+1} = \rho$, since it must not include an error state. However, it has

to contain all successors of the states in F_k. In order to fulfill the latter, IC3 checks whether there exist states in F_k that are predecessors of error states. To this end, it issues the query

$$\|F_k\| \wedge \|T\| \wedge \neg\|\rho\|'$$

to the SAT-solver (line 18). If the query is unsatisfiable, no such state exists and the sequence of frames is strong enough to be extended by the next frame $F_{k+1} = \rho$ preserving the 4 properties. Otherwise, a state s is extracted from the satisfying interpretation (line 20). With s being a state in F_k and having a successor violating ρ, Property 3 would not be preserved when extending the sequence of frames. Thus, s must be excluded from F_k. This is done in the recursive blocking procedure *blockCTIs*, which is explained below. Listing 2.3 shows the pseudocode.

Listing 2.3: Algorithm for excluding states from the frames in IC3

```
27  blockCTIs(Set Q){
28      while(Q≠ ∅){
29          get (s,n) ∈ Q with smallest n;
30          if(||F_{n−1}|| ∧ ¬||s|| ∧ ||T|| ∧ ||s||' unsatisfiable){
31              Q := Q\{(s,n)};
32              generalizeAndBlock(s,n);
33          }else if (n−1==0){
34              //found counterexample
35              return false;
36          }else{
37              //using the satisfying interpretation
38              state t = extract predecessor
39              Q := Q ∪ {(t,n−1)};
40          }
41      }
42      return true;
43  }
```

During the blocking phase, IC3 keeps and updates a set of pairs (s, n), each of which denotes the necessity to block a state s in a frame F_n. Such a pair is called obligation and the state included in it is named *Counterexample to Induction* (CTI), as it hinders the inductiveness of ρ (predecessors of error states) or the relative inductiveness of the negation of CTIs (line 38). The blocking phase tries to work off all obligations starting from the one that needs to be blocked in the smallest frame (line 29). For this obligation (s, n), the algorithm checks inductiveness of $\neg s$ relative to frame F_{n-1}. Imagine blocking s in frame F_{n-1}, the query issued in line 30 to the solver asks whether state s can be reached from the remaining states in frame F_{n-1} by a single transition step. If the query is unsatisfiable, s is not reachable and can safely be blocked in all frames F_0 to F_n. The reason is that $\|F_i\| \wedge \neg\|s\| \wedge \|T\| \wedge \|s\|'$ is unsatisfiable for all $i < n$ since each such F_i is a subset of F_{n-1}.

This blocking process can be seen as a sequence of exclusions starting from F_0 up to F_n. With s being no initial state, it is not a member of F_0. The unsatisfiability of $\|F_0\| \wedge \neg\|s\| \wedge \|T\| \wedge \|s\|'$ assures that s is unreachable in one transition from F_0 and can be excluded in F_1. The unsatisfiability of $\|F_1\| \wedge \neg\|s\| \wedge \|T\| \wedge \|s\|'$ assures that s is unreachable in one transition from $F_1 \setminus \{s\}$ and can be excluded in F_2. The sequence goes forth until s can be excluded from F_n.

However, the query for inductive relativeness might be satisfiable for some obligations. In that case, a state t (distinct to s) exists that is a predecessor of s and is a member of F_{n-1}. The state t prevents the exclusion of s from F_n, since it hinders $\neg s$ from being inductive relative to F_{n-1}. It is, thus, a *Counterexample to Induction* and needs to be excluded from F_{n-1} in order to be able to eventually exclude s from F_n. Line 39 shows the creation of a new obligation $(t, n-1)$ that expresses this fact.

In a very specific case, such an obligation does not need to be created as the exclusion of t from F_{n-1} is not possible. Line 33 checks whether $F_{n-1} = F_0$, which means we need to block a state from F_0, which is not possible since F_0 contains exactly the initial states (cf. line 6). Thus, IC3 found the predecessor t of another CTI s, which is itself a predecessor.

Following the found list of predecessors creates a path from an initial state to an error state, meaning a counterexample trace has been found and IC3 terminates (lines 35, 23 and 9). If no such counterexample is found, eventually all obligations are worked off (line 31 reduces the number of obligations) and the blocking procedure finishes. It returns to the strengthenClauses procedure, which checks for further predecessors of error states to be blocked in order to eventually create the next frame.

One of the most important aspects of IC3, however, is the fact that it does not exclude a single state s. Instead, it generalizes the state s into a set of states that are to be blocked (line 32).

To this end, it employs a *Generalization* procedure depicted in Listing 2.4.

Listing 2.4: Generalization algorithm in IC3

```
44 generalizeAndBlock(s,n){
45     find minimal subclause ‖c‖ of ¬‖s‖, s.t.
46         ‖F0‖ ∧ ¬‖c‖ is unsatisfiable
47         ‖Fn−1‖ ∧ ‖c‖ ∧ ‖T‖ ∧ ¬‖c‖′ is unsatisfiable
48     for(int i=1; i≤n; i++)
49         ‖Fi‖ := ‖Fi‖ ∧ ‖c‖;
50 }
```

Employed prior to blocking the state s from F_n, it searches a minimal subclause of $\neg\|s\|$ that is inductive relative to F_{n-1}. Such a minimal subclause always exists, since $\neg s$ itself is inductive relative to F_{n-1} (line 30). However, the search is non-trivial since monotonicity is not given for the consecution query of relative inductiveness of the subclauses. The removal of a single literal in the subclause might destroy the consecution property although it held before. In addition, the removal might also

re-establish consecution although it did not hold for the subclause before removing the literal. The reason is c and $\neg c'$ being altered at the same time, where the removal of a literal strengthens c, but weakens $\neg c$. Thus, the search for such a minimal subclause is hard. Most implementations rely on two or more combined approaches. On the one hand, the manual dropping of single literals one by one can be checked with a short query to the solver. If the query returns unsatisfiable, the subclause is still inductive relative, otherwise, the dropped literal is put back in the clause. This manual removal relies on heuristics to determine the order in which literals are tried to be dropped. While this is a valid approach, it is only capable of testing the removal of one literal at a time. Using extended concepts of SAT-solving, this drawback can be reduced. In case the subclause is still inductive relative after the removal of a literal, the query is found to be unsatisfiable, which allows the extraction of an unsatisfiability core (UNSAT-Core). Such a core is used to delete all those literals of the subclause that are not needed in proving the formula unsatisfiable. The removal of these literals is a safe option if the initiation property is not violated by doing so. Using this approach, most implementations do not actually care to find the minimal subset, but in contrast search for a small one in reasonable time. Due to this non-optimality and the removal of several literals determined by the UNSAT-core, the heuristics play an important role in steering of the generalization.

Having found a generalized clause $\|c\|$ of $\neg\|s\|$, it is conjoined to every formula $\|F_0\|$ up to $\|F_n\|$ (lines 48 and 49). This conjunction excludes the state s from the frames, as well as many more states (in case c is a strict subclause).

The benefit of this generalization procedure is a fast refinement and small clauses computed in a fairly efficient way.

In addition, a second procedure exists that provides for a fast refinement. The pseudocode depicting this *Propagation* step is shown in Listing 2.5.

Listing 2.5: Propagation algorithm in IC3

```
51 propagateClauses(int k){
52     for(int i=1; i<k; i++)
53         for each clause ||c|| in ||F_i||:
54             if(||F_i|| ∧ ||T|| ∧ ¬||c||' unsatisfiable)
55                 ||F_{i+1}|| := ||F_{i+1}|| ∧ ||c||;
56 }
```

The procedure pushes learned clauses to subsequent frames whenever possible, providing strengthened frames for the next cycle and, thus, a better guided refinement. To this end, IC3 checks all clauses of each frame for relative induction. If a clause c is inductive relative to a frame F_i, it can be propagated to F_{i+1}, refining F_{i+1} via conjunction as depicted in line 55. Since no state outside of c can be reached from F_i within one transition, this refinement does not destroy the 4 properties.

The propagation procedure is subject to several tricks in efficient implementations. By storing clauses in a data structure that associates them only with the largest frame with which they are conjoined, each clause is tested only once for propagation.

Furthermore, subsumption checks are often applied in this phase in order to filter out obsolete clauses that are not needed any longer.

Correctness All these different procedures and phases maintain the four properties of the frames, such that in the end IC3 is a successful algorithm for computing inductive strengthenings. We will shortly give an intuition why these properties hold during all the phases and why the algorithm terminates for finite state transition systems. These can be found in more detail in the work of Bradley et al. [Bra11].

During the initial phase of the algorithm (lines 1 to 6), IC3 ensures that there exist no error states that are initial or reachable from an initial state by a single transition step. Thus, the first two frames $F_0 = I$ and $F_1 = \rho$ can be created and adhere to the four given properties. In particular, Property 1 is ensured by F_0 being equal to the initial states. Property 2 is ensured in line 2 since ρ includes all initial states, while Property 3 is ensured in line 4. The last property is obviously true due to Property 2 and F_1 being equal to ρ.

These properties are maintained during each cycle of the main algorithm (lines 7-15). They are not violated by the call of function *strengthenClauses* and remain intact during the call of *propagateClauses* as we will see below. Finally, the creation of a new frontier $F_{k+1} = \rho$ adheres to the properties. Since it does not alter the other frames, it remains to show that $\|F_k\| \wedge \neg\|F_{k+1}\|$, $\|F_k\| \wedge \|T\| \wedge \neg\|F_{k+1}\|'$ and $\|F_{k+1}\| \wedge \neg\|\rho\|$ are unsatisfiable before the incrementation of k (line 14) and starting a new loop. The latter is true by definition of F_{k+1} and the first one is true by Property 4 for all smaller frames. The fact that $\|F_k\| \wedge \|T\| \wedge \neg\|F_{k+1}\|'$ is unsatisfiable has been established by *strengthenClauses()*, as it ensured that no direct predecessor of an error state is included in F_k (line 18).

We now consider the method *strengthenClauses()*. During its run, the 4 properties are preserved and when returning *true*, there exists no state in the frame F_k that is a direct predecessor of an error state. The method itself does not alter the frames, but however calls function *blockCTIs({(s,k)})* to block the direct predecessor s of an error state in the frame F_k. This is done until no such predecessors are left. What remains to be shown is that blockCTIs successfully blocks the found CTIs while preserving the 4 properties.

The function *blockCTIs()* works off a list of obligations until none is left. The frames are only altered when calling the function *generalizeAndBlock(s, n)*. This is only done when $\neg s$ was found to be inductive relative to F_{n-1} in which case it is also inductive relative to all smaller frames. The function *generalizeAndBlock(s, n)* searches for a minimal subclause c of $\neg s$ that is still inductive relative. This clause is added to F_1 up to F_n altering the frames and successfully blocking s from F_n. The 4 properties, however, are preserved. In particular, Property 1 still holds, as F_0 is untouched and Property 4 holds due to frames being strengthened. Property 2 still holds since $\|F_0\| \wedge \neg\|c\|$ is unsatisfiable (line 46) and c is added to all F_1 up to F_n. Since Property 3 previously held true and relative consecution (line 47) of clause c holds, the refined frames still satisfy Property 3. Thus, the methods *generalizeAndBlock, blockCTIs* and *strengthenClauses* preserve the 4 properties for the frames.

In addition, the function *propagateClauses* also preserves these properties due to the same reasoning over relative inductiveness. Hence, in total, the 4 properties are maintained for the frames after and during the main loop, allowing the algorithm to find an inductive strengthening, if one exists.

Termination The algorithm will always terminate for a finite state transition system. With the frames being a non-decreasing sequence of sets of states and the algorithm still running, meaning it did not terminate due to two frames being equal, each frame has to include at least one additional state compared to the next smaller frame. That means there can only exist $|S|$ frames at the same time for a finite state transition system as defined in Definition 2.3.1. Thus, the main loop of the algorithm will eventually terminate, provided that the functions *strengthenClauses* and *propagateClauses* terminate.

StrengthenClauses will terminate when no direct predecessor of an error state remains in frame F_k. Assuming the function *blockCTIs* successfully blocks each found predecessor, strengthen must terminate after at most $|S|$ runs of its while loop. The function *blockCTIs* will always terminate. The list of obligations can only contain $|S| \times k$ obligations. Provided that the call to *generalizeAndBlock* terminates and indeed blocks the state s from frames F_0 to F_n, the list of obligations will eventually be worked off or a counterexample is found. Due to the blocking, an obligation can not be found again in the same run of *blockCTIs* after being worked off. Thus, the method will terminate.

Each call of function *generalizeAndBlock* terminates and successfully blocks state s in frames F_0 to F_n. The reason is that only a finite number of subclauses exist and, thus, the minimal subclause can be found. Such a minimal subclause always exists, since $\neg s$ itself is inductive relative to F_{n-1}. In summary, each call of *strengthenClauses* will terminate.

In addition, each call of the function *propagateClauses* terminates since k is finite and the number of clauses in a frame is finite.

Summing up, all functions called by the main loop terminate and the main loop itself also terminates due to adding a new frame each cycle, of which there are only finitely many.

The argumentations about termination and correctness are carried out formally and in more detail in the first paper on IC3 by Aaron Bradley [Bra11].

2.4.2 Optimizations

As mentioned above, there exists a large community of researchers trying to improve and optimize IC3 or transfer it to different domains. Some of the optimizations and improvements are specifically tailored to the domain of application (for example ternary simulation in hardware verification), while others are more general. In the following, we present and discuss some of them.

Many optimizations target the method of generalization (Listing 2.4). Hassan et al. [HBS13] proposed to utilize *Counterexamples to Generalization*, meaning predecessor

states found during the generalization procedure. These are used in order to infer additional clauses that adhere to the relative inductiveness property. Thus, the approach utilizes unsuccessful tries for finding subclauses, however, it is not guaranteed that the inferred clauses are indeed helpful. In general, the optimization is evaluated as successful.

Chockler et al. [Cho+11] utilize an additional SAT-query as a preliminary step to generalization. Designed for hardware verification they search for partial assignments of a CTI by encoding the transition with which the CTI is found, but negating the successor state. Due to being deterministic for the specific set of input values, no distinct successor state exists, rendering the SAT-query unsatisfiable. Chockler extracts a partial assignment of the CTI from the UNSAT-Core, thereby potentially generalizing the CTI.

A similar effect is the result of ternary simulation, as proposed for use in IC3 by Een et al. [EMB11]. Using three valued logic, the effect of removing literals is propagated through the hardware circuit and in the end defines whether the removed literal is of use or not. The same paper also proposed several other optimizations, including a variation of the four properties for the frames and a variant of the algorithm that does not need the two basic checks (lines 2-5). In addition, they propose clever ways of organizing obligations, as well as the clauses in the frames. The resulting implementation, called *Property Directed Reachability* (PDR) proved to be superior to the original implementation and, thus, got adapted as a synonym for IC3.

Other improvements use the propagation phase as starting point. Suda [Sud13] proposed to trigger the propagation of clauses only if possible. To this end, he keeps track of witnesses that hinder the propagation of a clause. Only if such a witness is not valid any more, the propagation of the clause is tried. The approach seems to work well.

Other variants, however, did not prove to be of substantial value, as can for example be seen in the original paper by Bradley [Bra11]. He proposed a variation of the blocking procedure that searches a subclause inductive relative to a larger frame than F_{n-1} (cf. Listings 2.3 and 2.4). Its performance, however, was worse for specific hardware designs due to being unable to use the UNSAT-Core.

In summary, a lot of research has been successful to various extend in improving and optimizing IC3 itself.

There also exists work on the reusage of verification results as is of interest in this thesis. With hardware models changing often during the design process, Chockler et al. [Cho+11] utilize a previously computed inductive strengthening for the re-verification of the changed model. They apply an optimized SAT-encoding in order to find a subset of clauses of the inductive strengthening that are still inductive in the changed model. The found (inductive) clauses are then injected in the re-verification in order to speed it up.

In addition to this incremental way of using IC3, the algorithm has also been applied in completely different contexts and domains. Some work was done on

transferring IC3 to domains distinct from hardware verification and on combinations of IC3 with other techniques.

Hassan et al. [HBS12] utilize the idea of IC3 being incremental and inductive for CTL-model checking. They successfully build a property-directed abstracting model checker for CTL with fairness.

Baumgartner et al. [Bau+12] apply IC3 in the hardware verification domain in order to compute abstractions. They analyze the frames in an incomplete run of IC3 using heuristics in order to obtain priorities for state variables. On the basis of these priorities, a localization abstraction refinement is guided.

Another approach directly combines IC3 with abstraction and utilizes CTIs for refinement. In contrast to the original CEGAR approach, the technique of Birgmeier et al. [BBW14] utilizes single states (the CTIs) for refinement of the abstraction, while the original approach uses counterexample traces. The focus on CTIs enables two distinct points for triggering the refinement of the abstraction and, in addition, enables a choice of delaying the refinement.

More general work examines IC3 from a broader perspective. Bayless et al. [Bay+13] introduce the concept of *SAT modulo SAT*, which works similar to lazy SMT-solvers. They utilize the concept for their implementation of IC3, which proves to be successful.

IC3 is also applied as coverability check of well structured transition systems, in particular petri nets. Kloos et al. [Klo+13] propose an IC3-based algorithm for this task. Their implementation is SAT-based with predecessor and relative inductiveness computations being directly applied on the petri net.

The concepts of IC3 have also been transferred to domains that require other concepts than SAT-solving. For example, the application of IC3 for game solving (Morgenstern et al. [MGS13]) requires a QBF-solver. As a second example, the usage of IC3 for planning problems (Suda [Sud14]) sticks out with the absence of a solver. Instead, Suda delegates the tasks of the solver to a specific planning procedure.

In general, most approaches that do not rely on SAT-solving apply the concept of SMT-solving.

2.4.3 IC3 with SMT

Cimatti et al. were the first to combine IC3 with the concept of SMT-solving in order to model check software [CG12]. Apart from changes to the algorithm in order to support software model checking based on the control-flow graph, they apply a quantifier elimination procedure in linear real arithmetic to cope with the potential infinity of CTIs in a single blocking phase.

Other approaches try to be less theory-specific. A combination of IC3 using SMT with abstraction also proposed by Cimatti et al. [Cim+14] allows to handle a wide range of background theories. During the run of the algorithm an implicit abstraction is computed that is refined, whenever a spurious counterexample is found. The work can, thus, be seen as an instance of CEGAR.

Cimatti also applied IC3 with SMT in order to synthesize parameters [Cim+13a]. Relying on a quantifier elimination procedure, the algorithm searches for parameter combinations that allow counterexample traces. Gradually excluding these combinations, the tool in the end returns a set of safe parameter combinations.

With timed systems in mind, we are specifically interested in applications of IC3 using linear real arithmetic, of which some exist. Hoder et al. [HB12] generalized the PDR algorithm for usage with nonlinear transformers (used, e.g., for modeling software with procedures), in which counterexamples unfold to trees instead of paths. In the same work they apply PDR for linear real arithmetic. Their work applies interpolation in the generalization procedure in order to ensure decidability of the reachability of timed push-down systems. Bjørner et al. extend the previous concept [BG14]. They propose three variations of the generalized PDR algorithm for linear real arithmetic. The first one is basically a redefinition of the pure generalized PDR algorithm applying quantifier-elimination-based projection to extract CTIs, while the other two restrict themselves to find specific inductive invariants. In the second algorithm, the frames are defined as convex polyhedra, meaning conjunctions of linear inequalities. Thus, the algorithm is able to compute a convex polyhedron as inductive invariant. Contrary to this approach, the authors also present a variant of generalized PDR that is able to compute co-convex invariants, denoting the complement of a convex polyhedron.

The latter two techniques are, however, not suited for timed automata verification as they do not reason about discrete parts, e.g., locations of a timed automaton. The approaches mentioned prior to these two include such capabilities. However, their usage of interpolation and quantifier elimination is a generic mechanism. In contrast, we expect more specific techniques designed for timed automata to be better suited. The combination of discrete and continuous components in a timed automaton permits adjusted mechanisms, which is done in the approach presented below.

2.4.4 IC3 for TA

Kindermann et al. [KJN12b] applied IC3 specifically tailored to timed systems. As explained before, this design decision allows several general optimizations of IC3. His approach, however, was not competitive.

The infinity of clock valuations inherent in continuously timed systems poses the main challenge to IC3. When lifting the algorithm to SMT by encoding the system via SMT-formulae, the strengthen phase is no longer guaranteed to terminate. The blocking phase blocks the CTI, which is a single concrete state of the infinite transition systems as defined in Definition 2.1.10. Since there possibly exist infinitely many states all satisfying the query in line 18, the strengthen phase might never terminate or only if the generalization blocks all of them by chance. The same arguments can be used in order to explain a possibly infinite run of the blocking phase, where also infinitely many CTIs might exist.

Thus, modifications to the algorithm are needed to ensure termination. To this

end, the decidability result of reachability properties for timed automata might have led the way for the approach of Kindermann et al. [KJN12b]. With decidability being proven via a finite, reachability equivalent abstraction, the idea of composing all clock valuations that are indistinguishable for the timed automaton seemed like a good candidate. Thus, the region abstraction was employed due to being finite and easily computable for a single state.

Kindermann uses the original IC3 approach, but encodes the timed system via SMT-formulae. The algorithm is run as usual, but whenever a CTI is extracted, a specific technique is applied. As in the original approach, he extracts the concrete state from the satisfying SMT-interpretation. Afterwards he applies an abstraction procedure that replaces the concrete clock valuation of the concrete state with its unique, surrounding region in order to form an abstract state used as CTI. With only finitely many regions, the abstraction eliminates the risk of possibly having infinitely many CTIs, ensuring the termination of the algorithm.

Although the surrounding region of a concrete clock valuation is efficiently computable, the region abstraction is not well suited to be used in practical application, as has been explained before. The abstraction is very fine grained and the number of regions heavily depends on the constants used in the timed automaton. The combination of these two characteristics ensures a huge number of regions, rendering the algorithm ineffective, in particular for large constants. Kindermann's evaluation of his approach showed a good scalability in terms of model size, but does not deal well with large constants. While being a nice theoretical concept, the combination of IC3 with the region abstraction for timed systems proved to be of no practical value. Kindermann proposed some specific optimizations generalizing the CTI upfront before the generalization procedure of IC3 was applied. However, these suggestions could not establish practicality.

Yet, the basic idea has shown to be very interesting since it allows to benefit from future, general IC3 improvements and optimizations. For this reason, our approach presented in Chapter 3 builds on it.

Summing up, the work closest to ours is the one of Kindermann et al. [KJN12b] proposing to use IC3 with the region abstraction for timed system verification. In addition, the work of Chockler et al. [Cho+11] is closely related to our objective of handling reconfigurations. In Chapters 4 to 6 we will pick up this concept in order to improve the verification of safety properties for reconfigured timed models.

3 Timed Automata Verification via IC3 with Zones

Many of today's systems are safety critical, which expresses the need for their correct functioning at all times. Otherwise, undesired behavior could result in a loss in value, e.g., when a production is stopped, or even be a thread to life and physical condition. As a countermeasure, model-based design processes introduce a structured procedure in which models of the systems are built. Then, formal methods are employed to check whether specified properties are fulfilled. With an increasing number of systems relying on real-time communication, or being real-time operating system, there exists a demand for timed models and verification. As detailed in the previous chapter, networks of timed automata are amongst the most important modeling formalisms that incorporate time. With roughly 25 years of research, this modeling formalism is well understood and there exist many sophisticated algorithm for the verification of safety properties for these models. However, these algorithms reach their limits for large models, in particular due to an enormous need of memory during verification.

In the following, we will employ the IC3 algorithm for the verification of safety properties in the domain of timed automata. IC3 originates from hardware verification, in which it has proven to be extremely efficient, both regarding time and memory requirements. We aim for it to be of value also in the domain of timed automata. Its huge success for hardware verification has led to a large amount of optimizations and, in addition, it has been successfully transfered and applied in many other, very diverse domains. Its first usage in the domain of timed automata by Kindermann et al. [KJN12b], however, remained unsatisfying, as it was not competitive.

In this chapter, we introduce our concept combining the IC3 algorithm with the zone abstraction in order to verify safety properties for networks of timed automata [IW14]. We first present the basic components, i.e., the SMT-encoding and zone computation, and afterwards explain their integration in the IC3 algorithm. Lastly, we present the promising results of numerous experiments. We discuss for which

models our approach is able to outperform state of the art tools and why it is inferior for others. Additionally, we experiment with several distinct characteristics of timed automata, e.g., the number of locations, to observe the effects on the performance of our approach. We start with our encoding of a network of timed automata using SMT.

3.1 SMT-Encoding

As detailed in Section 2.4, the IC3 algorithm is entirely based on SAT-solving, which is why we need an encoding of our models. With networks of timed automata including real valued clocks, shared integer variables and locations, we encode them in SMT using linear real arithmetic (LRA), linear integer arithmetic (LIA), as well as propositional logic. A purely boolean encoding as SAT-formulae is possible using booleanization (translation into an untimed system, which can be encoded via propositional logic), but has proven to be non-competitive [KJN12b].

There exist at least three encodings for timed systems, of which the one presented by Kindermann [KJN12b; KJN12a] is closest to ours, but differs, e.g., in the encoding of the locations. While Kindermann uses a variable ranging over the number of locations as has been done by Maria Sorea [Sor03], our approach is closer to the one by Audemard et al. [Aud+02] using boolean variables to represent locations. This usage of several variables to encode a location enables a stronger generalization within IC3. Our SMT-formulae encode the concrete transition system presented in Definition 2.1.10, handling the zone abstraction separately without the use of SMT. Note, that our encoding combines an edge transition ($\rightarrow_e$) with a subsequent time delay transition ($\rightarrow_d$) into a single transition step. This combination is a feasible option for reachability analysis as each path in the transition system can be seen as an alternating sequence of delay and edge transitions. To this end, time delay steps with a delay of 0 time are inserted between two adjacent edge steps, and adjacent time delay steps are combined into a single one.

This combination allows an efficient encoding of the concrete semantics, using the theories mentioned above (LRA, LIA) for their specific parts. Each continuous real-valued clock is encoded as a real-valued variable and each integer variable in the network of timed automata is encoded using an integer-valued variable. The discrete set of locations is encoded via boolean variables.

We call these variables clock, integer and location variables, respectively. As in the previous chapter, we denote the encoding of a part x of the model as $\|x\|$. Again, we use two distinct sets of variables to encode a current and a next state, where the formulae reasoning about the next state are marked as primed. Given a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ defined over $C^g, \mathcal{IV}$ and Σ, we encode it as follows.

3.1.1 Variables

Our encoding employs variables for clocks, locations and integer variables in the model. We use uniquely renamed variables in order to avoid ambiguity and dealing with characters that are forbidden in SMT. In the following, we give a detailed explanation.

Variables encoding the Global Clocks Each global clock $c \in C^g$ is assigned a unique identifier $id(c) \in \{0, \dots, |C^g| - 1\}$. We encode each such clock c as a real valued variable $\|c\| := c^0_{id(c)}$. The subscript shows the unique identifier, while the superscript zero denotes the clock to be global.

Variables encoding the Local Clocks In contrast to global clocks, local clocks are part of a specific timed automaton in the network. Let the automaton A_j ($j \in \{1, \dots, n\}$) be given. Each local clock $c \in C^l_j$ is assigned an identifier $id(c) \in \{0, \dots, |C^l_j| - 1\}$ that is unique in its timed automaton. Given these two identifiers j and $id(c)$, we encode the clock via real-valued variable $\|c\| := c^j_{id(c)}$.

Variables encoding the Integer Variables The encoding of integer variables only uses a single identifier, as these variables are not associated with a specific timed automaton, but shared within the entire network of timed automata. We employ a unique identifier $id(iv) \in \{0, \dots, |\mathcal{IV}| - 1\}$ for the encoding of variable $iv \in \mathcal{IV}$. For each integer variable $iv \in \mathcal{IV}$ in the network, we create an integer variable $\|iv\| := int_{id(iv)}$ for use in the SMT-formulae according to its assigned identifier.

Variables encoding the Locations The locations of each timed automaton are encoded using boolean variables. Consider the timed automaton A_j ($j \in \{1, \dots, n\}$). There exist $|L_j|$ different locations. We encode each location using a unique identifier $i \in \{0, \dots, |L_j| - 1\}$, which is encoded using its binary representation. To this end, we create mx boolean variables, where mx is a function over the timed automata denoting the number of boolean variables needed to encode the largest identifier $|L_j| - 1$. Formally, mx is defined as $mx(A_j) = \begin{cases} 1 & \text{if } |L_j| = 1, \\ \lceil log_2(|L_j|) \rceil & else. \end{cases}$
The created variables are named l^j_0 to l^j_{mx-1}, where l^j_0 encodes the least significant bit. The encoding of a location is done via its binary representation as a conjunction of the above boolean variables, s.t. a true value (1) in the binary representation is encoded as a positive literal and a false value (0) as negated literal. We denote the encoding of location l of A_j as $\|l\|_j$. We exemplify such an encoding in the following.

Example 3.1.1. Consider a timed automaton A_j with $|L_j| = 4$ different locations assigned identifiers 0 to 3. The location l with identifier 0 is encoded using its binary representation 00. Having created $mx = \lceil log_2(|L_j|) \rceil = 2$ boolean variables, the SMT-formula describing location l of A_j can be written as $\|l\|_j := \neg l^j_1 \wedge \neg l^j_0$, since location l has identifier 0.

All the above clock, integer and location variables are used to describe a state in the concrete transition system. For the purpose of encoding a transition step, we additionally need variables encoding the next state after having taken a transition step. To this end, all the above variables are copied as a primed version, respectively $c_i^{0\prime}$, $c_i^{j\prime}$, int_i' and $l_i^{j\prime}$. Using these variables, we create SMT-formulae encoding the initial states, the transition relation and the safety property.

3.1.2 Encoding of the Initial States

The formula encoding the initial state must adhere to Definition 2.1.10. Initially, each timed automaton A_j ($j \in \{1, \ldots, n\}$) is in its initial location $l_{0j} \in L_j$, encoded via a conjunction of the locations' encodings. Furthermore, all integer variables are required to equal their initial values. For each integer variable $iv \in \mathcal{IV}$ with initial value $v_0^i(iv)$, we encode this requirement as $\|iv\| = v_0^i(iv)$. As explained above, we encode a combination of an edge transition ($\rightarrow_e$) with a delay transition ($\rightarrow_d$). To this end, a first time delay transition is encoded directly into the initial states, and each edge in the transition relation is also followed directly by a time delay. Thus, initially all clocks have exactly the same value (≥ 0) that includes a first time elapse for all clocks. The requirement that all invariants have to be met is encoded separately for a better reusability of the SMT-formula, which is also needed when encoding a transition step or error states. This separation allows a modular combination of the individual formulae. Hence, the encoding of the initial clock valuations in the formula at hand must only ensure all clocks to be of the same value. To this end, we employ an auxiliary real valued variable called *initClockValue*. Each clock $c \in C$ is assigned the same value as said variable via conjunct $\|c\| = initClockValue$, regardless if the clock is local or global. Additionally, the auxiliary variable is constrained to be larger or equal to 0 by conjunct $initClockValue \geq 0$. Formally, the formula encoding the initial states is defined as follows.

Definition 3.1.2. Let a network of timed automata be given as in Def. 2.1.11. The SMT-formula encoding the initial states is defined as

$$
\begin{aligned}
\|Init\| := & \bigwedge_{j \in \{1,\ldots,n\}} \|l_{0j}\|_j \\
& \bigwedge_{iv \in \mathcal{IV}} \|iv\| = v_0^i(iv) \wedge initClockValue \geq 0 \\
& \bigwedge_{c \in C^g} \|c\| = initClockValue \quad \bigwedge_{j \in \{1,\ldots,n\}} \bigwedge_{c \in C_j^l} \|c\| = initClockValue.
\end{aligned}
$$

The following example explains the encoding of the initial states.

Example 3.1.3. Let the network of timed automata be given as depicted in Figure 3.1, modeling two processes using the Fischer mutual exclusion algorithm. Each of the automata has a local clock c and l_0 is its initial location assigned identifier 0. The global integer variables *cnt* and *id* have initial values $v_0^i(cnt) = 0$ and $v_0^i(id) = 0$.

Thus, the encoding of the initial state looks as follows:

$$\begin{aligned}\|Init\| :=& \neg l_1^1 \wedge \neg l_0^1 \wedge \neg l_1^2 \wedge \neg l_0^2 \\ & \wedge int_0 = 0 \wedge int_1 = 0 \wedge initClockValue \geq 0 \\ & \wedge c_0^1 = initClockValue \wedge c_0^2 = initClockValue.\end{aligned}$$

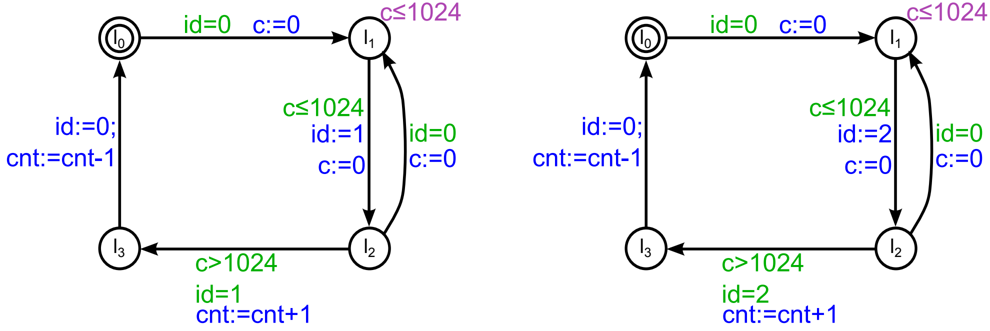


Figure 3.1: Network of two timed automata modeling the Fischer Mutual Exclusion algorithm for two processes

As explained this formula is satisfiable by interpretations that represent an initial state. However, there might exist satisfying interpretations that do not represent a feasible initial state, since we do not check that the values adhere to the locations' invariants here. Instead, as explained before, we encode the invariants separately. This additional encoding includes clock and integer constraints. In the following, we show how these components are encoded.

3.1.3 Encoding of Clock Constraints

Clock constraints play an important role in restricting the models behavior as they are used as invariants and guards within edges. Using SMT, the encoding of a clock constraint is straightforward. Each clock is encoded as defined above, while the encoding of comparison operators is done using the symbols of the employed theory of linear real arithmetic. In detail, the encoding is defined as follows.

Definition 3.1.4. Let a clock constraint ϕ be given as defined in Def. 2.1.2. The SMT-formula $\|\phi\|$ is inductively defined on the structure of ϕ. If ϕ equals

- $x \bowtie n$ for some clock x, $\bowtie \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$, then $\|\phi\| := \|x\| \bowtie n$,
- $(x - y) \bowtie n$ for some clocks x, y, $\bowtie \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$, then $\|\phi\| := (\|x\| - \|y\|) \bowtie n$,
- $\phi_1 \wedge \phi_2$ for some clock constraints ϕ_1 and ϕ_2, then $\|\phi\| := \|\phi_1\| \wedge \|\phi_2\|$.

If the clock constraint equals true, this can naturally be encoded as the boolean constant true. However, this special constraint represents a location not having a clock constraint invariant or an edge having no guard. Thus, it can be left out as it is not restricting the behavior.

The following example illustrates the encoding.

Example 3.1.5. Consider the clock constraint $c \leq 1024$ in the second timed automaton in Figure 3.1. It is encoded as $c_0^2 \leq 1024$.

Similar to clock constraints, we also have to encode integer constraints. In the following, we will see that they are encoded analogously.

3.1.4 Encoding of Integer Constraints

Integer constraints are used in addition to clock constraints to restrict the model's behavior. Their encoding using SMT is straight-forward, employing the theory of linear integer arithmetic. In detail, the encoding is defined as follows.

Definition 3.1.6. Let an integer constraint ψ be given as defined in Def. 2.1.4. The SMT-formula $\|\psi\|$ encoding it depends on the constraint. If ψ equals

- $iv \bowtie n$ for integer variable $iv \in \mathcal{IV}$, $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$ and $n \in \mathbb{Z}$, then $\|\psi\| := \|iv\| \bowtie n$,
- $\psi_1 \wedge \psi_2$ for some integer constraints ψ_1 and ψ_2, then $\|\psi\| := \|\psi_1\| \wedge \|\psi_2\|$.

We exemplify this encoding in the following.

Example 3.1.7. Consider the integer constraint $id = 2$ in the second timed automaton in Figure 3.1. With integer variable id having identifier 0, the constraint is encoded as $int_0 = 2$.

Given the above mechanisms to encode locations and constraints, we can now define the encoding of invariants.

3.1.5 Encoding of Invariants

Invariants are defined as functions over the locations of the timed automata in the network. If a reachable state includes a specific location, then the respective invariant has to be met. This relationship can be encoded as an implication. In particular, each invariant constraint is only dependent on its respective location being met. Thus, we do not need to encode the product locations of the network of automata, but are able to encode each location separately.

In addition to the encoding of the locations' invariants, we need additional restrictions due to locations being encoded using their binary representation. By creating mx boolean variables to encode the locations of an automaton, 2^{mx} different locations can be encoded. However, if less than 2^{mx} locations exist, some interpretations of

the variables are invalid as they represent nonexistent locations. We exclude these interpretations upfront accompanying the invariants. Furthermore, our encoding includes restrictions on the real valued variables representing the clocks. We add a conjunct for each clock ensuring its value to be nonnegative.

The combined encoding is described below.

Definition 3.1.8. Let a network of timed automata be given as in Def. 2.1.11. The SMT-formula encoding the locations' invariants, the exclusion of invalid location identifiers, and clocks being of nonnegative value is defined as follows.

$$
\begin{aligned}
\|Invar\| := & \bigwedge_{j\in\{1,\dots,n\}} \bigwedge_{l\in L_j} \|l\|_j \Rightarrow (\|Inv^c(l)\| \wedge \|Inv^i(l)\|) \\
& \bigwedge_{j\in\{1,\dots,n\}} \bigwedge_{\textit{nonexistent location } l \textit{ in } A_j} \neg\|l\| \\
& \bigwedge_{c\in C} \|c\| \geq 0
\end{aligned}
$$

We illustrate this encoding again with the help of the Fischer model.

Example 3.1.9. Consider the network of timed automata as depicted in Figure 3.1. Both automata have 4 locations and, thus, no invalid location identifiers have to be excluded. However, both automata include a clock invariant for location l_1 with identifier 1. Hence, the resulting formula encoding the invariant looks as follows.

$$
\begin{aligned}
\|Invar\| := \quad & ((\neg l_1^1 \wedge l_0^1) \Rightarrow (c_0^1 \leq 1024)) \\
& \wedge((\neg l_1^2 \wedge l_0^2) \Rightarrow (c_0^2 \leq 1024)) \\
& \wedge(c_0^1 \geq 0) \wedge (c_0^2 \geq 0)
\end{aligned}
$$

Whenever a single state is queried, the above formula $\|Invar\|$ is conjoined to the query in order to ensure the invariants are met and only valid locations and clock valuations are considered. The same holds true for successor states, being enforced by the same formula defined over the respective primed variables. Thus, whenever a transition step is queried, both the primed and unprimed invariant formulae enforce their restrictions on states by conjunction with the transition formula, which is introduced below.

The transition relation contains synchronized and unsynchronized edges. The encoding of the former edges is done straight-forward, while the encoding of the latter ones requires some preprocessing. We compute all combinations of sender and receiver edges upfront and afterwards encode these combinations. This process allows a straight-forward encoding of all edges and a simple extraction of the taken edge (or edge combination) from a satisfying interpretation. The overhead due to the computation of combinations is negligible in most models.

A modular encoding separating the sender and receiver edges might be possible, perhaps using implications and auxiliary variables for signaling purpose. However, the main characteristic of synchronized edges requiring the synchronization of exactly one sender and one receiver edge renders a modular encoding rather complex.

In addition, the extraction of the taken edges needed for the computation of the CTIs might be more difficult.

Thus, we compute the combinations of edges, as defined in Def. 2.1.11. As described in the semantics, only those edges of the product automaton can be taken that are not synchronized (marked by the synchronization symbol ϵ). We encode these edges by enforcing the source and target locations of those timed automata, in which the edges are taken. Furthermore, we encode the constraints and integer assignments and resets of clocks. In addition, we ensure all unused clock, integer and location variables to keep their values, meaning they have the same value in their unprimed and primed versions.

We first describe how integer assignments and resets of clocks are encoded.

3.1.6 Encoding of Clock and Integer Updates

Encoding updates on the clocks and integer variables inherently makes use of both the unprimed and primed versions of the variables, representing the current and the next state values, respectively. First, we define the encoding of integer assignments.

Encoding of Integer Assignments As mentioned before, we can combine each sequence of assignments (Definition 2.1.5) into a sequence, in which the order does not matter, since each integer variable exists at most once in a left hand side of an assignment. This fact results in a simple encoding using conjunction to combine the assignments without the need for intermediate steps. Note, however, that the encoding of a sequence, in which the order is essential, can easily be done using auxiliary variables and intermediate steps. The following definition shows the simple encoding.

Definition 3.1.10. Let an integer assignment ω be given as defined in Def. 2.1.5. The SMT-formula $\|\omega\|$ encoding it depends on the assignment. If ω equals

- $iv := n$ for some integer variable $iv \in \mathcal{IV}$ and $n \in \mathbb{Z}$, then
  $\|\omega\| := \|iv\|' = n$,
- $iv := iv + n$ for some integer variable $iv \in \mathcal{IV}$ and $n \in \mathbb{Z}$, then
  $\|\omega\| := \|iv\|' = \|iv\| + n$,
- $\omega_1; \omega_2$ for some integer assignments ω_1 and ω_2, then
  $\|\omega\| := \|\omega_1\| \wedge \|\omega_2\|$,
- *true*, then
  $\|\omega\| := true$.

For an assignment *true*, the same arguments applies as for an invariant *true*. It is ignored and, thus, all integer variables are to keep their value. This process of keeping the value is encoded separately (as $\|iv\|' = \|iv\|$) in each of the edges for all integer variables $iv \in \mathcal{IV}$ that are not changed by the assignment.

We illustrate the encoding with the following example.

Example 3.1.11. Consider again the Fischer model as depicted in Figure 3.1. The integer assignment $id := 0; cnt := cnt - 1$ can be seen on the edge between location l_3 and l_0 in each of the automata. With integer variable *id* having identifier 0 and *cnt* having identifier 1, the integer assignment is encoded as
$\|\omega\| := (int'_0 = 0) \wedge (int'_1 = (int_1 - 1))$.

The encoding is straight-forward for the integer assignments. However, clock updates can not be encoded this way due to always considering the combination of an edge-transition with a subsequent time elapse step. To this end, we create an additional real-valued variable δ denoting the time elapse after the edge has been taken.

Encoding of Clock Resets Given the set $R \subseteq C$ of clocks to be reset, the update of each clock is encoded dependent on it being an element of R or not. All members are reset to value 0, while the remaining clocks keep their value. Afterwards all clock values are increased by a time delay step, encoded as the addition of a real valued variable δ. Formally, the update for each clock $c \in C$ is encoded as

$$\begin{aligned} \|c\|' &= \delta && \text{if } c \in R, \\ \|c\|' &= (\|c\| + \delta) && \textit{else}. \end{aligned}$$

Using the above encodings, we can finally formalize the encoding of edges.

3.1.7 Encoding of Edges

When encoding the edges of an NTA, we distinguish between the encoding of edges that originate from synchronization and those that don't. We start with the latter, denoting such edges that can be found in single timed automata in the network, marked by symbol ϵ, meaning no synchronization takes place.

Unsynchronized Edges These edges are related only to a single timed automaton A_i ($i \in \{1, ..., n\}$) in the network $NTA = \langle A_1, ..., A_n \rangle$. Thus, they do not consider the locations of the other timed automata, as well as their local clocks. These values do not matter, as they are unchanged when the edge is taken (not taking into account successive time elapse). Hence, our encoding does not need to compute the product automaton with all its combinations of locations. Instead, we demand A_i to be in the source location before taking the edge and to be in the target location afterwards. In addition, the clock and integer constraints have to be met, as well as the reset of clocks and the integer assignments. The encoding is presented in detail below.

Definition 3.1.12. Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11. Each unsynchronized edge $e = (l_1 \xrightarrow{\epsilon, \phi, \psi, \omega, R} l_2) \in E_i$ of each timed automaton A_i ($i \in \{1, \ldots, n\}$) is encoded as follows.

$$
\begin{aligned}
\|e\| := & \|l_1\|_i \wedge \|l_2\|_i{}' \\
& \wedge \|\phi\| \wedge \|\psi\| \wedge \|\omega\| \\
& \bigwedge_{c \in R} \|c\|' = \delta \bigwedge_{c \in C \setminus R} \|c\|' = \|c\| + \delta \\
& \bigwedge_{\substack{j \in \{1,\dots,n\} \\ j \neq i}} \bigwedge_{m=0}^{mx(A_j)} l_m^{j\,\prime} = l_m^j \\
& \bigwedge_{iv \in \mathcal{IV}, iv \notin \omega} \|iv\|' = \|iv\|
\end{aligned}
$$

We explain the encoding in the following. The first line encodes the automaton A_i being in location l_1 before the edge is taken and in l_2 afterwards, which is encoded via the primed variables. The next line demands the constraints ψ and ϕ to hold before the edge is taken and the assignment ω being applied to the integer variables. The third line demands the clocks in R to be reset, while all others keep their value. In addition, a time elapse of δ is encoded, where the requirement of the delay being nonnegative ($\delta \geq 0$) is encoded separately, together for all edges. Finally, the locations of the timed automata other than A_i are preserved, as well as the values of the integer variables that are not updated in the assignment ω.

We illustrate the encoding with the following example.

Example 3.1.13. Consider the Fischer model as depicted in Figure 3.1. We show the encoding of the edge e leading from l_2 to l_3 (with identifiers 2 and 3, respectively) in the first automaton. It requires A_1 to be in the respective locations before and after the transition, encoded as $\|l_2\|_1 := l_1^1 \wedge \neg l_0^1$ and $\|l_3\|_1' := l_1^{1\prime} \wedge l_0^{1\prime}$. Furthermore, it requires integer variable id, encoded as int_0, to have value 1 ($\|\psi\| := int_0 = 1$) and local clock c of A_1, encoded as c_0^1, to be of value larger than 1024 ($\|\phi\| := c_0^1 > 1024$). The integer assignment ω increments the variable cnt by one, encoded as $\|\omega\| := int_1' = (int_1 + 1)$. All other variables keep their value during the transition step. The entire encoding looks as follows.

$$
\begin{aligned}
\|e\| := & l_1^1 \wedge \neg l_0^1 \wedge l_1^{1\prime} \wedge l_0^{1\prime} \\
& \wedge (c_0^1 > 1024) \wedge (int_0 = 1) \wedge (int_1' = (int_1 + 1)) \\
& \wedge (c_0^{1\prime} = (c_0^1 + \delta)) \wedge (c_0^{2\prime} = (c_0^2 + \delta)) \\
& \wedge (l_0^{2\prime} = l_0^2) \wedge (l_1^{2\prime} = l_1^2) \\
& \wedge int_0' = int_0
\end{aligned}
$$

As explained before, our encoding does not need to compute any product of locations and is, thus, very scalable. The same holds true for the encoding of synchronized edges, which we describe below. However, we need to compute all combinations of synchronized sender and receiver edges, which results in a bad scalability for models with many sender and receiver edges synchronized via the

same channel. As noted above, a different kind of encoding without the need for such a computation might be feasible, but results in an encoding way more complex with some sort of signaling between sender and receiver that ensures exactly one edge of each kind.

Thus, we compute the combination of synchronized edges upfront end encode them similar to the previous encoding. The main differences are the usage of two source and target locations, as well as two constraints and assignments due to the sender and receiver edges belonging to two distinct timed automata. In the following, the encoding is formally defined.

Definition 3.1.14. Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11. Each synchronized edge e combined from sender edge $e_s = (l_1 \xrightarrow{a!, \phi_1, \psi_1, \omega_1, R_2} l_2) \in E_i$ and receiver edge $e_r = (l_3 \xrightarrow{a?, \phi_2, \psi_2, \omega_2, R_2} l_4) \in E_j$ of timed automata A_i and A_j ($i \neq j \in \{1, \ldots, n\}$) is encoded as follows

$$
\begin{aligned}
\|e\| :=& \|l_1\|_i \wedge \|l_3\|_j \wedge \|l_2\|_i{}' \wedge \|l_4\|_j{}' \\
& \wedge \|\phi_1\| \wedge \|\phi_2\| \wedge \|\psi_1\| \wedge \|\psi_2\| \wedge \|\omega_1; \omega_2\| \\
& \bigwedge_{c \in R_1 \cup R_2} \|c\|' = \delta \quad \bigwedge_{c \in C \setminus (R_1 \cup R_2)} \|c\|' = \|c\| + \delta \\
& \bigwedge_{\substack{h \in \{1,\ldots,n\} \\ h \notin \{i,j\}}} \bigwedge_{m=0}^{mx(A_h)} {l_m^h}' = l_m^h \\
& \bigwedge_{iv \in \mathcal{IV}, iv \notin \omega_1;\omega_2} \|iv\|' = \|iv\|
\end{aligned}
$$

Using these encodings of the edges, we can encode the transition relation. It is explained below.

3.1.8 Encoding of Transition Relation

The transition relation consists of all unsynchronized edges and the combined synchronized ones. When the transition relation is applied, the current state in the concrete transition system is changed according to the taken edge. Thus, an edge must be taken, which is encoded as a disjunction of the formulae representing the edges. The variable δ modeling the elapsed time is used within all the encodings of the edges and, thus, the requirement that a nonnegative amount of time has passed is encoded together once for all edges.

Definition 3.1.15. Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11. Let $\{e_1, \ldots, e_m\}$ be the set of unsynchronized edges and combined synchronized edges as in the previous definitions. The transition relation describing the application of one of them is encoded as

$$\|Trans\| := (\|e_1\| \vee \cdots \vee \|e_m\|) \wedge \delta \geq 0.$$

All the above definitions are employed to encode the concrete semantics of a network of timed automata as explained. For our verification approach, however, the safety property must also be encoded as SMT-formula. This encoding is defined below.

3.1.9 Encoding of Safety Property

A safety property is defined as a conjunction of negations of error state specifications. Each error state specification is defined as a triple of partial location vector, clock constraint and integer constraint. We have described above, how all these parts are encoded. They are simply combined via conjunction to encode an error state specification.

Definition 3.1.16. Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11. Let $err = (\bar{l}, \phi, \psi)$ be an error state specification as given in Definition 2.1.13. It is encoded using the above definitions as

$$\|err\| := \bigwedge_{\substack{j \in \{1,...,n\} \\ \bar{l}[j] \neq *}} \|\bar{l}[j]\|_j \wedge \|\phi\| \wedge \|\psi\|,$$

where the encoding of the partial location vector encodes only those locations that are specified. In addition, constraints ϕ and ψ are discarded, if they specify *true*.

As stated before, the encoding of the safety property is simply the conjunction of negations of the error state specifications. We formally define it in the following.

Definition 3.1.17. Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11. Let the safety property $\rho = G(\neg err_1 \wedge \cdots \wedge \neg err_x)$ be given specifying that no reachable state should satisfy one of the error state specifications err_1 to err_x. It is encoded as $\|\rho\|$ as follows.

$$\|\rho\| := \neg \|err_1\| \wedge \cdots \wedge \neg \|err_x\|$$

We illustrate the encoding with the following example.

Example 3.1.18. Consider again the Fischer model as depicted in Figure 3.1. It models the Fischer mutual exclusion algorithm that ensures only a single process to be in a critical section. The critical section is modeled as location l_3 and the number of processes in the critical section is counted via integer variable cnt. Let $cnt > 1$ be the error state specification considered here. It specifies that more than a single automaton is in its critical section at the same time. The safety property of interest denotes that no such error state can occur, formally $\rho := G(\neg(cnt > 1))$. It is encoded as $\|\rho\| = \neg(int_1 > 1)$.

Finally, all formulae needed are explained and defined. We briefly summarize their usage within IC3 below. To this end, we give an overview over the queries issued to the SMT-solver.

3.1.10 Usage of the Encodings in the Queries

As stated before, the SMT-formula encoding the transition relation, as well as those for the initial states or the safety property rely on the combination with the unprimed and primed invariant formulae. In consequence, we create these formulae modularly in order to combine them when needed.

We briefly recall the queries issued by IC3 (upper formula) and show how they are build using our encoding (lower formula).

- (An error state is initial.)
  $\|I\| \wedge \neg\|\rho\|$:
  $\|Init\| \wedge \|Invar\| \wedge \neg\|\rho\|$
- (A predecessor of an error state is initial.)
  $\|I\| \wedge \|T\| \wedge \neg\|\rho\|'$:
  $\|Init\| \wedge \|Invar\| \wedge \|Trans\| \wedge \|Invar\|' \wedge \neg\|\rho\|'$
- (A predecessor of an error state is a member of F_k.)
  $\|F_k\| \wedge \|T\| \wedge \neg\|\rho\|'$:
  $\|F_k\| \wedge \|Invar\| \wedge \|Trans\| \wedge \|Invar\|' \wedge \neg\|\rho\|'$
- (A predecessor of state s that is distinct from s is a member of F_{n-1}.)
  $\|F_{n-1}\| \wedge \neg\|s\| \wedge \|T\| \wedge \|s\|'$:
  $\|F_{n-1}\| \wedge \neg\|s\| \wedge \|Invar\| \wedge \|Trans\| \wedge \|Invar\|' \wedge \|s\|'$
- (An initial state is a member of $\neg c$.)
  $\|F_0\| \wedge \neg c$:
  $\|F_0\| \wedge \|Invar\| \wedge \neg c$
- (A predecessor of a state in $\neg c$ is a member of c and of F_{n-1}.)
  $\|F_{n-1}\| \wedge c \wedge \|T\| \wedge \neg c'$:
  $\|F_{n-1}\| \wedge c \wedge \|Invar\| \wedge \|Trans\| \wedge \|Invar\|' \wedge \neg c'$
- (A predecessor of a state in $\neg c$ is a member of F_i.)
  $\|F_i\| \wedge \|T\| \wedge \neg c'$:
  $\|F_i\| \wedge \|Invar\| \wedge \|Trans\| \wedge \|Invar\|' \wedge \neg c'$

The above formulae encode states and transitions in the concrete transition system, as explained in Definition 2.1.10. It might be feasible to encode an abstraction of the clock valuation space directly, e.g., by introducing variables for bounds of clock valuations. However, the encodings would most probably grow far more complex. In particular, the transition relation encoding would do so and, thus, slow down the generalization step. Our encoding refrains from introducing this additional complexity and, instead, favors an auxiliary step outside the scope of IC3's generalization procedure.

This additional step is vital for our approach. Since our encoding represents a concrete transition system with clocks having values from $\mathbb{R}_{\geq 0}$, there might be infinitely many satisfying interpretations to a satisfiable query. This infinity poses

a problem, especially when considering the queries asking for counterexamples to induction (CTIs). There might exist infinitely many satisfying interpretations and, in consequence, infinitely many CTIs, e.g., within one run of *strengthenClauses*. Hence, the algorithm would have to exclude infinitely many CTIs for termination. Even when taking into account the generalization procedure, termination can not be ensured. This is due to the fact, that the generalization algorithm is only capable of deleting literals (for example $c = 1.771$). Tackling this problem by altering the generalization algorithm to shift values is hardly feasible.

Thus, we employ an abstraction in order to ensure termination. First, we need to obtain the concrete states that the solver has computed in form of a satisfying interpretation. This step, called *state extraction*, is detailed below.

3.2 State Extraction

Executing the IC3 algorithm using an SMT-solver with the above encoding results in CTIs being concrete states. Each satisfying interpretation is inspected to extract values for the location variables, clock variables and integer variables. These represent the clock valuation, integer valuation and locations of a state in the concrete transition system (or two states - predecessor and successor - in case of an encoded transition step). The clock and integer valuations can be extracted directly, however, the locations need to be computed from the boolean values of the location variables. The boolean values represent the identifier of the location encoded as integer value in binary form. The extraction of the concrete state is exemplified in the following.

Example 3.2.1. Let the network of timed automata be given as depicted in Figure 3.1. The query $\|F_k\| \wedge \|Invar\| \wedge \|Trans\| \wedge \|Invar\|' \wedge \neg\|\rho\|'$ asks the solver for a predecessor state (in F_k) of an error state. Consider the following part of a satisfying interpretation returned by the solver.

$$
\begin{aligned}
&(define - fun\ int_0\ ()\ Int\ 1)\\
&(define - fun\ int_1\ ()\ Int\ 1)\\
&(define - fun\ l_1^1\ ()\ Bool\ true)\\
&(define - fun\ l_0^1\ ()\ Bool\ false)\\
&(define - fun\ l_1^2\ ()\ Bool\ true)\\
&(define - fun\ l_0^2\ ()\ Bool\ false)\\
&(define - fun\ c_0^1\ ()\ Real\ 1025.0)\\
&(define - fun\ c_0^2\ ()\ Real\ 1.0)\\
&\ldots
\end{aligned}
$$

Note, that the format of the satisfying interpretation depends on the used solver. The presented interpretation represents only values for the predecessor state. We extract the following state: Both integer variables int_0 and int_1 have value 1 and, thus, the integer valuation v^i for the integer variables of the model is set accordingly ($v^i(id) = 1, v^i(cnt) = 1$). The extraction of the clock valuation v^c is done in exactly the

same way ($v^c(c_1) = 1025.0, v^c(c_2) = 1.0$), where c_1 and c_2 denote clock c in automata A_1 and A_2, respectively. The extraction of the locations is a bit more involved. Taking the values for the boolean variables, we build the binary representations of the identifiers. In this example, both binary representations are 10 representing location l_2 for A_1 and A_2.

As explained above, the concrete clock valuation poses a potential thread to the termination of the algorithm. In the following, we show an example having infinitely many such valuations for CTIs all taking the same edge to an error state.

Example 3.2.2. We consider the same example as above with query $\|F_k\| \wedge \|Invar\| \wedge \|Trans\| \wedge \|Invar\|' \wedge \neg\|\rho\|'$ asking for a predecessor state (in F_k) of an error state. The satisfying interpretation represents an edge transition from location l_2 to l_3 in the first timed automaton, while the second one keeps its location l_2. The integer valuation changes from $v^i(id) = 1, v^i(cnt) = 1$ to $v^{i'}(id) = 1, v^{i'}(cnt) = 2$. The respective concrete clock valuation found by the solver is $v^c(c_1) = 1025.0, v^c(c_2) = 1.0$ and does not change ($v^c = v^{c'}$) with $\delta = 0$. It is easy to see, that the locations and valuations enable the given edge with the given successor state being an error state, i.e., satisfying $cnt > 1$. Clearly, there exist infinitely many other satisfying interpretations for the query: Keeping all the above values, but changing $v^c(c_1)$ to any arbitrary value strictly larger than 1024.0. Figure 3.2 shows the computed concrete clock valuation (marked as x), as well as the other valuations that could be used in a satisfying interpretation (marked gray).

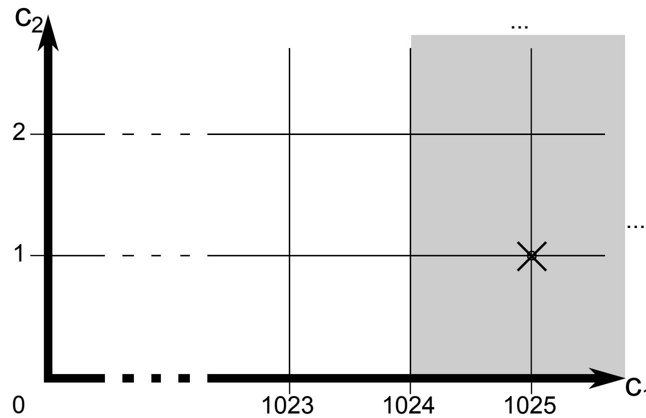


Figure 3.2: An infinite number of satisfying interpretations can exist: When changing the clock valuation found by the solver for Example 3.2.2 (marked as x) to any other one of those marked gray, the respective interpretation still satisfies the issued query

In 2012, Kindermann et al. [KJN12b] proposed a solution to circumvent infinitely many CTIs by using the region abstraction. They extract the concrete state, as shown above, and afterwards compute the region surrounding the found clock valuation. The computation is straight-forward, in particular, due to each concrete clock valuation belonging only to a single, unique region. This region is used in lieu

of the concrete clock valuation forming an abstract CTI. The essence of his approach is that termination is ensured due to the fact that there exist only a finite number of distinct regions. Thus, when blocking an abstract CTI, none of the concrete clock valuations included in the entire region can occur again as CTI (for a specific frame) in the current cycle of the *strengthenClauses* procedure. Accordingly, with abstract CTIs being blocked, only a finite number of distinct frames can be created until two of them are found to be equal (or a counterexample trace is found). As a result, termination is guaranteed.

Kindermann examines his approach thoroughly leading to the conclusion that it scales similar to the state of the art tool Uppaal, but is not competitive to it due to larger runtimes in general. The main reason for the performance issues is the use of the region abstraction. Although being finite, the number of CTIs may still be enormously large due to exponential growth of the employed abstraction regarding time constants and clocks. Thus, the approach is irrelevant in practice.

In general, however, Kindermann's approach points out a valuable direction. By applying finite abstraction, IC3 can be utilized for the verification of timed automata. Such an abstraction should be coarse and efficiently computable in order to be better suited for practical purposes.

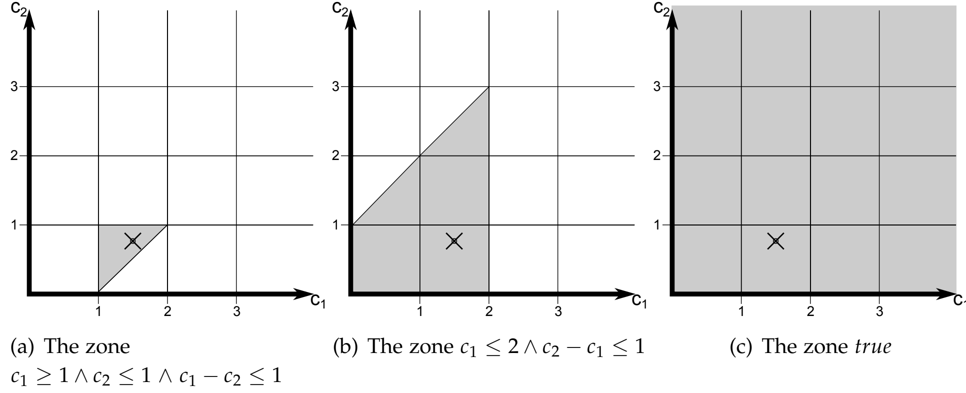
In our approach, detailed below, we apply the zone abstraction. We believe that this abstraction is particularly suitable for application in IC3 due to its properties. It is coarser than the region abstraction, it is finite since a zone is the union of regions and there exist sophisticated algorithms for computation and manipulation of zones. Even though it is coarse, it is exact enough to describe exactly all those clock valuations that reach a specific zone of a successor state via the same edge. Note, that none of the problems of storing disjunctions of zones occurring in other approaches (see the related work in Chapter 2) are present in the IC3 approach.

However, applying the zone abstraction is not as straight-forward as the region abstraction, since there is no unique surrounding zone for each clock valuation. We explain this issue and our solution in the following.

3.3 Zone Abstraction

Our SMT-formulae as shown above encode the concrete semantics of a timed automaton. The states extracted from satisfying interpretations are, thus, concrete states with a single clock valuation. Unlike in Kindermann's approach, one can not simply compute a surrounding zone for such a clock valuation as there might exist several surrounding zones. We exemplify the ambiguity in the following example.

Example 3.3.1. Let the set of clocks be given as $C = \{c_1, c_2\}$. A concrete clock valuation v^c extracted from a satisfying interpretation of a query might contain the values $v^c(c_1) = 1.5, v^c(c_2) = 0.75$. It is depicted in Figure 3.3 marked as x, together with several surrounding zones. Each such zone is a convex union of clock valuations (defined as a clock constraint) including the found clock valuation. Some of these zones are very fine grained and some are extremely coarse. The difficulty here is to chose the right one.



(a) The zone $c_1 \geq 1 \wedge c_2 \leq 1 \wedge c_1 - c_2 \leq 1$ (b) The zone $c_1 \leq 2 \wedge c_2 - c_1 \leq 1$ (c) The zone *true*

Figure 3.3: A concrete clock valuation, as extracted from a satisfying interpretation of an SMT-query, might be included in numerous surrounding zones with different size

We search for a zone that includes the concrete valuation found by the solver, and is as large as possible, while permitting only relevant behavior. The last two properties are opposed, but reasonable. A zone that is too small may result in impracticality of the approach, while a zone that is too large would break the IC3 algorithm. For illustration consider computing a zone that equals the region surrounding the concrete clock valuation. Clearly, this choice results in impracticality due to being too fine grained. It depicts exactly the approach proposed by Kindermann et al. [KJN12b]. On the other hand, a zone *true* allowing all valuations would be too coarse and disallow the selective blocking of specific clock valuations and, in consequence, break the IC3 algorithm.

We denote behavior as relevant, if is analog to the one found via the SMT-query, meaning the same edge is taken with the same error state or CTI as successor. Thus, the zones we are interested in contain exactly those clock valuations that enable the same edge and reach the same (abstract) successor state as the clock valuation found by the query.

Due to the abstraction of the found concrete clock valuation into a zone, the IC3 algorithm no longer deals with concrete states as CTIs, but with abstract states. We denote these states as *abstract CTIs*, as opposed to the previously used *concrete CTIs*. Abstract CTIs are abstract states as defined below.

Definition 3.3.2 (Abstract State)**.** Let there be given a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ as in Def. 2.1.11 with concrete semantics $TS = (S, s_0, \rightarrow)$. The set of abstract states is defined as $S_a = L \times \Phi(C) \times \Psi(\mathcal{IV})$. An *abstract state* $s_a = (l, \phi, \psi) \in S_a$ denotes the set of concrete states $\{(l, v^c, v^i) \in S | v^c \models \phi \wedge v^i \models \psi\}$.

By taking into account the definition of safety properties and abstract CTIs, it is easy to formalize the above intuition of the largest zone with relevant behavior.

3.3.1 Zone Computation

IC3's search for CTIs always includes an SMT-query involving a single transition step (lines 18 and 30). Assuming the zone of the successor state to be known, we can apply traditional backwards reachability computation to compute the zone that abstracts the found clock valuation of the concrete CTI. It requires the extraction of the concrete predecessor and successor states from the satisfying interpretation and based thereon the backwards computation. In detail, the following steps are performed.

1. - Extract the concrete predecessor state (concrete CTI) s from the satisfying interpretation for the unprimed location, clock and integer variables.
   - Extract the concrete successor state t from the satisfying interpretation for the primed location, clock and integer variables.
   - Extract the used edge e from the satisfying interpretation of $\|Trans\|$.
2. Get the successor's zone Z' depending on the query.
3. Compute the predecessor zone Z by backwards computation from Z' using s,t and e. It abstracts the predecessor's concrete clock valuation and is used in the abstract CTI.

Extracting the concrete states is done as described previously in Section 3.2. However, getting the successor's zone Z' is a bit more involved depending on the employed query. Either, the successor state is an abstract CTI (see Query in line 30) or it is an error state (see Query in line 18). In the first case, an abstract CTI is encoded as successor state and, thus, its zone must have been previously computed and can simply be used. In the latter case, the negation of the safety property is encoded. Hence, the successor state satisfies one of the error state specifications in the safety property. In order to identify which one, we issue a series of small queries to the SMT-solver that check compatibility of the extracted concrete successor state t with each error state specification in the safety property. Definition 3.3.3 denotes how these queries are build.

Definition 3.3.3. Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11. Given an error state specification $err = (\bar{l}, \phi, \psi)$ as defined in Definition 2.1.13, we check whether a concrete state $t = ((l_1, \ldots, l_n), v^i, v^c)$ is included in err by issuing the following SMT-query to the solver.

$$\|t\| \wedge \|err\| := \bigwedge_{i=1}^{n} \|l_i\|_i \bigwedge_{c \in C} (\|c\| = v^c(c)) \bigwedge_{iv \in \mathcal{IV}} (\|iv\| = v^i(iv)) \wedge \|err\|$$

At least one of these queries has to be satisfied since the extracted state t clearly is an error state. In case of satisfiability, t is included in the error state's specification. Thus, we extract the respective zone ϕ from the error state specification and use it as successor zone Z'.

Despite the difference in step 2 (getting the successor zone Z'), both methods have the same idea in mind. Not the found concrete clock valuation of the successor state t is of interest, but all the clock valuations included in Z'. They are useful in combination with the knowledge of the predecessor's and successor's locations and integer valuations and the taken edge. Taking into account all the information, we compute the largest set of clock valuations that enables the given edge and leads to a successor clock valuation in Z' under the given assumptions of successor's and predecessor's locations and integer valuations. In the literature, the procedure is usually called backward analysis (c.f. [Bou09]). The entire handling of zones within our approach is based on the DBM structure as it is trivial to compute the DBM representation of a clock constraint and vice versa. Hence, we do not strictly separate the notion of zones and their storage and manipulation in form of DBMs here. The employed backwards computation of the predecessor zone is, thus, presented as a sequence of algorithms based on DBMs. It refers to procedures defined by Bengtsson [Ben02].

Definition 3.3.4. Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11. In addition, let the concrete states $s = (l, v^i, v^c)$ and $t = (l', v^{i'}, v^{c'})$ be the states extracted from the satisfying interpretation with taken edge $l \xrightarrow{\epsilon, \phi, \psi, \omega, R} l'$. Given the successor zone Z' represented as a DBM, the following sequence of operations is executed to *compute the predecessor zone* Z represented as DBM. As explained above, the operations refer to algorithms from Bengtsson [Ben02]. The algorithm $backwards(l, l', e, Z')$ is defined by the following sequence of procedures.

1. $Z = down(Z')$
   The past of the DBM Z' is computed to include all those valuations that may have led to a valuation in Z' via time elapse.

2. $Z = and(Z, Inv^c(l'))$
   Only those clock valuations are valid that adhere to the successor locations' invariant. To this end, the conjunction of Z with the DBM representing the invariant is computed.

3. $Z = reset(Z, R)$
   After a reset all clocks in R are of value 0. Thus, only valuations with these clocks set to zero must be considered valid.

4. $Z = free(Z, R)$
   Prior to the reset all clocks in R may have been of any value. Thus, they are set free to allow any value.

5. $Z = and(Z, \phi)$
   The taken edge is only enabled, if the clock constraint ϕ is satisfied by the clock valuation. Thus, all invalid clock valuations are removed via a conjunction with the DBM representing ϕ.

The general problem when employing zone-based computation is that some of the algorithms need the DBM to be in canonical form, i.e., all its bounds are as tight as possible. This form is established using the all-pairs-shortest-paths algorithm of Floyd Warshall [Flo62]. With its cubic runtime, this process is the bottleneck of all basic zone computation approaches. There exist approaches to reduce the need to call the Floyd Warshall algorithm, e.g., the above algorithms from Bengtsson [Ben02], which we employ as explained. The bottleneck is, thus, reduced, but not eliminated completely.

By employing this sequence of algorithms, we compute a zone Z that includes all clock valuations that are able to reach a clock valuation in Z' for the given successor location and integer valuation under the given predecessor location, integer valuation and edge. This abstraction of the found concrete clock valuation allows the blocking of a large number of concrete states at once and, thus, is crucial in respect of performance and termination. As explained above, all zones are stored as difference bound matrices, allowing an efficient manipulation. Furthermore, their encoding via SMT-formulae is straight-forward as the clock difference constraints stored as bounds in the cells of the matrix are encoded easily as explained in Subsection 3.1.3.

Additionally, we employ another backwards computation for the integer valuations as detailed below.

3.3.2 Integer Constraint Computation

As explained above, our abstract CTIs include a set of integer valuations instead of a single valuation. These are stored as integer constraints. Analog to the zone, we extract the successor's integer constraint from the error state specification or the abstract CTI. The computation of the predecessor's integer constraint employs standard weakest precondition computation [Dij78] in order to include all integer valuations that enable the edge with a resulting valuation in the successor's integer constraint. This procedure is denoted $wp(\omega, \psi)$ below, for a sequence ω of integer assignments and a successor integer constraint ψ. It is defined as follows.

Definition 3.3.5. Let an integer constraint ψ be given, as well as an integer assignment ω. The *weakest precondition computation* computes the weakest integer constraint $wp(\omega, \psi)$ that includes all integer valuations, which result in a valuation in ψ via the application of ω. Technically, the left-hand side of the assignment is replaced in ψ by the right-hand side.

In the following, we show how the original IC3 algorithm changes when employing our method.

3.4 Algorithm

In general, the structure of the algorithm is kept the same. It is now based on the computation and blocking of abstract CTIs instead of concrete ones. Their encoding is straight-forward as explained above. In particular, the combination

with the encoding of the concrete semantics permits the usage of the generalization procedure without any modifications. The most complex changes to the algorithm happen in those parts that are concerned with computing the CTIs.

Two lines are affected, namely lines 20 and 38 in the original algorithm, which are located in the procedures *strengthenClauses* (Listing 2.2) and *blockCTIs* (Listing 2.3). The two occurrences represent the two distinct cases of accessing the successor's zone Z', in particular, the successor is an error state in method *strengthenClauses*, while it is an abstract CTI in *blockCTIs*. Thus, different modifications are needed in these separate spots. We present the altered procedures in the following, which both employ the distinct procedures for the weakest precondition computation of integer constraints (*wpIntegers*) and zones (*wpClocks*) as presented in Listings 3.3 and 3.4. Listing 3.1 shows the adapted procedure *strengthenClauses*.

Listing 3.1: Algorithm: Strengthen the frames in IC3 for Timed Automata

```
57 strengthenClauses(){
58     while(‖F_k‖ ∧ ‖T‖ ∧ ¬‖ρ‖' sat){
59         //extract from satisfying interpretation
60         concrete predecessor state s=(l,v^i,v^c);
61         //extract from satisfying interpretation
62         concrete successor state t=(l',v^i',v^c');
63         //extract from satisfying interpretation
64         taken edge e with s →_e t;
65         error state err=(l̄',φ',ψ') = fetchErrorState(t);
66         integer constraint ψ = wpIntegers(l,e,l',ψ');
67         clock constraint φ = wpClocks(l,e,l',φ');
68         combine into abstract CTI s_a=(l,φ,ψ);
69         if(!blockCTIs({(s_a,k)}))
70             //found counterexample trace
71             return false;
72     }
73     return true;
74 }
```

Starting in line 59, we extract the concrete predecessor (lines 59 and 60) and successor state (lines 61 and 62) and the taken edge (lines 63 and 64) from the satisfying interpretation of the issued query. This information is successively used to find an error state specified within the safety property, s.t. the concrete successor state t is included in the error state specification as explained in Definition 3.3.3. To this end, the procedure *fetchErrorState* is applied (line 65), which is described below in Listing 3.2.

Listing 3.2: Algorithm: Finding an error state specified in the safety property that includes a concrete state

```
75 fetchErrorState(Concrete State t){
76     //let ρ = ¬err_1 ∧ ··· ∧ ¬err_x
77     for(int i=1;i≤x;i++)
78         if(‖err_i‖ ∧ ‖t‖ sat)
79             return err_i;
80 }
```

Each error state specification is checked whether it includes the concrete successor state t, in which case the search is terminated and the found error state specification is returned. Since the concrete successor state has been extracted from the satisfying interpretation of the query in line 58, it is clearly included in at least one of the error state specifications. To this end, inclusion checks are performed using short SMT-queries in which both the error state specification in question, as well as the concrete state are encoded (line 78). The error state specification contains a partial location vector, an integer and a clock constraint. Its encoding in conjunction with the encoding of the concrete state is only satisfiable if the concrete state is included in the specification. Otherwise at least two literals encode contradicting information, rendering the formula unsatisfiable.

For illustration, consider the following example.

Example 3.4.1. Let the network of timed automata be given as depicted in Figure 3.1. It includes two local clocks, one for each automaton, denoted c_1 and c_2. Line 58 queries the solver for a CTI as predecessor of an error state. The concrete successor state extracted from the satisfying interpretation might be $t = ((l_3, l_2), v^{c\prime}, v^{i\prime})$ with $v^{i\prime}(id) = 1, v^{i\prime}(cnt) = 2$ and $v^{c\prime}(c_1) = 1025.0, v^{c\prime}(c_2) = 1.0$ as previously explained in Example 3.2.2. The taken edge is the one between locations l_2 and l_3 in the first automaton. As explained previously, the safety property specifies mutual exclusion as $\rho := \neg(cnt > 1)$. The search for an error state specification including the found concrete successor state will, thus, issue the following query

$$\|l_3\|_1 \wedge \|l_2\|_2 \wedge (\|id\| = 1) \wedge (\|cnt\| = 2)$$
$$\wedge(\|c_1\| = 1025.0) \wedge (\|c_2\| = 1.0) \wedge (\|cnt\| > 1).$$

The last conjunct encodes the error state specification and the other parts of the formula encode the concrete successor state t. With the satisfiability of the query, the IC3 algorithm knows which error state specification has to be used. In particular, $cnt > 1$ is used as successor integer constraint, while the used successor zone includes all valuations, since the constraint *true* was specified.

The found error state is exploited afterwards in order to compute the maximum sets of integer and clock valuations that enable the given edge e using the extracted locations l and l', leading to a concrete state satisfying the found integer and clock constraints of the error state. For both sets of integer and clock valuations, we employ

backwards computation algorithms computing the weakest precondition. They refer to the previously presented backwards computation and weakest precondition for zones and integers, respectively, but additionally ensure the invariants to be met. They are shown below.

Listing 3.3: Algorithm: Computing the weakest precondition of integer constraints

```
81  wpIntegers(l,e,l',ψ'){
82      //let e = l --ε,φe,ψe,ωe,Re--> l'
83      //conjunct integer invariant of l'
84      ψ = ψ' ∧ Inv^i(l');
85      //wp using assignments in edge e
86      ψ = wp(ωe,ψ);
87      //conjunct integer constraint (guard) of edge e
88      ψ = ψ ∧ ψe;
89      //conjunct integer invariant of l
90      ψ = ψ ∧ Inv^i(l);
91      return ψ;
92  }
```

The algorithm starts from the successor's integer constraint specifying the allowed valuations. Since the integer invariants have to be met by all valuations, the integer invariant constraints for successor location l' are added via conjunction (line 84). These combined constraints denote all integer valuations allowed in l' for the considered error state. In order to compute all possible predecessor valuations regarding edge e, the algorithm considers all integer assignments ω_e of e. These are used to compute the set of integer valuations that result in one of the above valuations after applying the assignments. To this end, standard weakest precondition as first defined by Dijkstra [Dij78] is employed. Thereafter, these constraints are conjoined with the integer guard of edge e and the integer invariant of l (lines 88 and 90). Thus, the resulting integer constraint ψ specifies exactly those integer valuations that are allowed in location l and enable the edge e, whose application results in a valuation included in ψ' and allowed in l'.

For illustration, consider the following example.

Example 3.4.2. Consider the same situation as in the previous example. The found error specification includes the integer constraint $cnt > 1$ with extracted successor location (l_3, l_2) and predecessor location (l_2, l_2). The extracted taken edge has integer constraint $id = 1$ and assignment $cnt := cnt + 1$. Applying the procedure $wpIntegers$ results in the following computation. First, the successor's integer constraint $cnt > 1$ is conjoined with the integer invariant constraint of the successor location, which does not exist here. Then, weakest precondition wp is applied resulting in the integer constraint $cnt + 1 > 1$, simplified to $cnt > 0$. Afterwards, the integer constraint of the edge is conjoined resulting in integer constraint $cnt > 0 \land id = 1$. Since no

integer invariant exists for the predecessor location, the above constraint is returned as depicted.

Similarly, the computation of clock constraints includes clock invariants, guards and resets. It entirely relies on manipulations of difference bound matrices representing the involved zones.

Listing 3.4: Algorithm: Computing the weakest precondition of integer constraints

```
93  wpClocks(l,e,l',φ'){
94      //intersection with clock invariant of l'
95      φ := and(φ', Inv^c(l'));
96      //backwards computation using edge e
97      φ := backwards(l,l',e,φ);
98      //intersection with clock invariant of l
99      φ := and(φ, Inv^c(l));
100     return φ;
101 }
```

Before using the backwards computation algorithm as explained in Subsection 3.3.1, the algorithm limits the allowed clock valuations according to the clock invariants of location l'. To this end, the intersection of the DBMs representing both zones is build. Subsequently, backwards computation is employed as defined in Definition 3.3.4 based on edge e. The result is the set of clock valuations that enables the clock guard of edge e, whose application leads to a valuation, which respects the clock invariant of l' and is included in the successor's zone. Finally, this set is intersected with the clock invariants of location l in order to include only valuations respecting the predecessor's invariants.

For illustration, we again extend the above example.

Example 3.4.3. Consider the same situation as in the previous example. The found error specification includes no specified clock constraints and, in consequence, the successor zone includes all valuations. The extracted successor locations are (l_3, l_2) and the predecessor locations are (l_2, l_2) with the clock constraint $c_1 > 1024$ restricting the application of the taken edge. Applying the procedure *wpClocks* results in the following computation. The intersection with the successor's clock invariant does not alter the set of valuations, as no such invariant exists. Then, the past is computed, which still includes all clock valuations. The application of *free* and *reset* is also without any effect, since no clocks are reset. Afterwards, the zone is restricted to only include clock valuations respecting the predecessor's clock invariant and the clock constraint of the edge ($c_1 > 1024$). Thus, after the application of procedure *wpClocks*, the resulting set of clock valuations can be specified by constraint $c_1 > 1024$.

In summary, both procedures computing the sets of predecessor valuations are designed to include the maximal sets of valuations that enable edge e while reaching

a successor valuation as specified by ϕ' and ψ'. Thus, they are extremely important, both related to efficiency and also termination of our approach. They are likewise used in the *blockCTIs* procedure whenever the contained query is satisfiable and a predecessor of a CTI needs to be computed. In contrast to the above case using the error state specifications, no search of a successor state is necessary here, since it is known in form of an abstract CTI that already includes a zone and integer constraint. These informations are, thus, exploited to compute the predecessor's constraint and zone. Listing 3.5 shows the changed pseudocode.

Listing 3.5: Algorithm: Excluding states from the frames in IC3 for timed automata

```
102 blockCTIs(Set Q){
103     while(Q≠ ∅){
104         //let abstract CTI s_a=(l',φ',ψ')
105         get (s_a,n) ∈ Q with smallest n;
106         if(‖F_{n−1}‖ ∧ ¬‖s_a‖ ∧ ‖T‖ ∧ ‖s_a‖' unsatisfiable)
107             Q := Q\{(s_a,n)};
108             generalizeAndBlock(s_a,n);
109         else if (n−1==0)
110             //found counterexample
111             return false;
112         else
113             //extract from satisfying interpretation
114             concrete predecessor state t=(l,v^i,v^c);
115             //extract from satisfying interpretation
116             taken edge e;
117             integer constraint ψ = wpIntegers(l,e,l',ψ');
118             clock constraint φ = wpClocks(l,e,l',φ');
119             combine into abstract CTI t_a=(l,φ,ψ);
120             Q := Q ∪ {(t_a,n−1)};
121     }
122     return true;
123 }
```

As can be seen, the same mechanisms are employed in *blockCTIs* in order to compute the predecessor state. With the successor state being an abstract CTI, its zone and integer constraints must have been previously computed and we can simply reuse them. This is the only difference to the above case located in the procedure *strengthenClauses*, in which we previously had to make an effort to find an abstract successor state.

The computation and usage of our abstract states serves as means able to ensure termination of the algorithm. However, there exist models for which even this concept can not guarantee it. We will see reasons for this in the following subsection, where we analyze and prove termination under certain restrictions to the models.

3.4.1 Termination

In Chapter 2, we illustrated the termination guarantee for the original IC3 algorithm. Given that the model is finite, there exist only finitely many states. Since the frames are sets of states and are required to be distinct, there can only be a finite number of frames. The rest of the argumentation is about each procedure call terminating eventually and about progress, meaning at least one of the frames is refined after the blocking phase.

In the case of our modified IC3 algorithm for timed automata using Zones, a similar reasoning can be applied, but encounters a couple of problems. The most obvious challenge concerning the infinite state transition system of a timed automata is solved using the Zone abstraction. Each abstract CTI includes a zone instead of a single, concrete clock valuation. When using backwards computation, only finitely many such zones can occur for a given timed automaton as they are unions of regions [Bou09]. Thus, the CTIs in our algorithm can only include one of these finitely many zones. In addition, they can only contain one of finitely many locations. In combination with a finite number of sets of integer valuations encoded in the integer constraints, our CTIs are of finite quantity. The generalization procedure taking place after CTI computation can not destroy finiteness, since it can only delete some of the literals. The result would be a set of locations with an enlarged zone and enlarged set of integer constraints. Hence, the same argumentation holds true as for the original algorithm, stating that there can only be finitely many different frames and, thus, the algorithm will terminate eventually.

Theorem 3.4.1. *Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given as in Def. 2.1.11. If the set of backwards reachable valuations for the integer variables $\mathcal{IV}$ is finite, then the presented algorithm $IC3$ with $Zones$ terminates.*

However, the presumption that there exists only a finite number of sets of integer valuations in a CTI may be incorrect. This difficulty comes from integer assignments $iv := iv + n$ for some $iv \in \mathcal{IV}$ and $n \in \mathbb{Z} \setminus \{0\}$. Whenever a cycle is present in the timed automaton that is able to repeatedly increase or decrease the valuation of an integer variable, there might exist an unbounded number of CTIs. It might occur that these are all ruled out by generalization and the algorithm terminates, but it can not be guaranteed. Thus, the presence of such cycles destroys the guarantee of termination of our modified IC3 algorithm. Contrary, the absence of such constructs inducing infinitely many sets of integer valuations guarantees termination. The following theorem formalizes the above illustration.

Theorem 3.4.2. *In general, the verification of safety properties for networks of timed automata as presented in this thesis is undecidable.*

This theorem has been proven for a closely related formalism in 2004 [Bou+04], which relies on updates on the clocks of a timed automaton. It involves a trivial reduction of the halting problem for deterministic 2-counter machines. Our formalism allows a similar reduction, which does not rely on updates on clocks, but instead

relies on our integer variables being unbounded and assignments being able to increment and decrement these values. We explain the details below. The reachability problem for deterministic 2-counter machines, which was shown to be undecidable in 1967 [Min67], can trivially be reduced to a safety property verification for networks of timed automata using our formalism. A 2-counter machine $(Q, q_o, \triangle)$ consists of a finite set of states Q with initial state $q_0 \in Q$, where $\triangle \subseteq Q \times Op' \times Q$ and $Op' = \{inc(c_1), inc(c_2), testAndJump(c_1), testAndJump(c_2), condDec(c_1), condDec(c_2)\}$ for two counters c_1, c_2 that can be incremented, tested for zero or decremented in dependence of the current value. These operations can be described as

- $inc(c_i)$: Increment counter c_i,
- $testAndJump(c_i)$: Require the value of counter c_i to be zero,
- $condDec(c_i)$: Require the value of counter c_i to be strictly larger than zero and decrement it.

Note, that the latter two are sometimes combined into a single instruction with two distinct target states in dependence whether the counter can be decremented or not. The reachability problem asks whether an accepting state $q_f \in Q$ is reachable in a finite number of steps starting from a given initial state with predetermined counter values. The reduction to a timed automaton is straight-forward.

Given any 2-counter machine $M = (Q, q_o, \triangle)$, we build a timed automaton $A = (L, l_0, C, \mathcal{IV}, \Sigma, Inv^c, Inv^i, E)$ and safety property ρ that is not invariant if and only if q_f is reachable in M. The two counters are mapped to two integer variables with respective initial values, i.e., $\mathcal{IV} = \{c_1, c_2\}$. There does not exist a clock and the locations represent the states of the 2-counter machine, meaning $C = \emptyset$ and $L = Q$ with $l_0 = q_0$. No synchronization takes place and no invariants are given on the locations, i.e., $\Sigma = \emptyset$, $\forall l \in L : Inv^c(l) = true$ and $\forall l \in L : Inv^i(l) = true$. Furthermore, the instructions that modify the counter values are mapped to updates on the integer variables on the edges between the locations. Formally, for every $(q_1, op, q_2) \in \triangle$, there exists an edge $e \in E$. If op equals

- $inc(c_i)$: $e = (q_1 \xrightarrow{\epsilon, true, true, c_i := c_i + 1, \emptyset} q_2)$,
- $testAndJump(c_i)$: $e = (q_1 \xrightarrow{\epsilon, true, c_i = 0, true, \emptyset} q_2)$,
- $condDec(c_i)$: $e = (q_1 \xrightarrow{\epsilon, true, c_i > 0, c_i := c_i - 1, \emptyset} q_2)$.

The safety property specifies the location representing the accepting state q_f to be an error state, formally $\rho := G\neg((q_f), true, true)$. Checking the reachability of the accepting state in the 2-counter machine is, thus, reduced to checking the violation of the safety property. Trivially, any path of configurations in the 2-counter machine exists as well in the timed automaton and, hence, the reachability problem could be solved if the safety property could be checked for violation. As a consequence, our problem of checking safety properties for timed automata as defined in Chapter 2

is undecidable in general. Note, that the undecidability is due to the unbounded usage of integer variables and assignments.

3.5 Evaluation

The presented approach verifies safety properties for networks of timed automata. One of its design goals is efficiency regarding memory consumption and runtime. The IC3 algorithm underlying our approach is one of the key components to achieve this goal. In addition, the abstraction of concrete clock valuations into zones is equally important, as it efficiently handles the infinity of the clock valuation space. In order to evaluate whether the design goal is met in practice, we implemented the presented technique using Java. Several standard models are employed as benchmarks in order to compare our method with the state of the art tool Uppaal and the previous approach by Kindermann [KJN12b]. These experiments provide a basis to estimate the scalability of the presented approach. The results are very promising, in particular regarding the scalability, but also point out some drawbacks.

In the following, we present the models employed in our experiments followed by some details about our implementation. Afterwards, the experiments are presented and discussed.

3.5.1 Benchmark Models

We employed several standard models from literature, as well as a smaller one that we developed ourselves considering the context of Industry 4.0. In total, the following benchmarks are subsequently presented and afterwards employed in numerous experiments.

- Models obtained from Uppaal website [UPP]
  - Fischer Mutual Exclusion algorithm
  - Carrier Sense Multiple Access with Collision Detection protocol
  - FDDI token ring protocol
- FDDI token ring protocol with counting variable
- Models obtained from Bruttomesso et al. [Bru+12]
  - Fischer Mutual Exclusion algorithm
  - Lamport Mutual Exclusion algorithm
  - Shavit-Lynch Mutual Exclusion algorithm
- A shrunk model of the Lamport Mutual Exclusion algorithm
- Model of the Shavit-Lynch Mutual Exclusion algorithm from PAT [PAT]

Fischer Mutual Exclusion Algorithm (Uppaal) One of the most important models for estimating the scalability of our technique is the network of timed automata, which we have shown previously in the examples. It models the *Fischer Mutual Exclusion algorithm* [Lam87] that ensures mutually exclusive access to a critical section for several processes. Given its dependency on time, it is a perfect fit to be modeled as a network of timed automata. There exist several models of the Fischer algorithm, most of which only differ in small details. The model we used in the examples and our experiments is taken from Uppaal, found at their website [UPP]. We denote it as *Fischer_U_x*, where x is the number of timed automata. Mutual exclusion is handled by each process writing its unique identifier to a shared variable when requesting access to the critical section. After waiting a certain amount of time, the process is allowed to proceed only if the shared variable still matches its identifier. To this end, each process is modeled as a timed automaton, each having its own unique identifier.

Figure 3.1 shows the model for two processes. The first process has identifier *1*, while the second one has identifier *2*. In consequence, the edges between locations l_1 and l_2, as well as the ones between l_2 and l_3 are different. Each process has to write its own identifier to the shared variable and accordingly check for its own identifier. All other parts of the model are equal for every timed automaton. The critical section is modeled as location l_3 and the integer variable *cnt* counts the number of processes currently in the critical section.

The safety property of interest asks whether at most one of the processes is in the critical section at every point in time. Using the integer variable *cnt* mentioned above, we can express this safety property as $\rho := G\neg((*,*), true, cnt > 1)$, abbreviated as $\rho := G\neg(cnt > 1)$.

With the Fischer algorithm being independent of the number of processes, it provides a perfect opportunity for scalability experiments. To this end, the number of processes, that is, timed automata in the network, is increased while the performance is examined.

Furthermore, modifications to the used time constants allow inspections to performance changes caused by time constants. These options are also present in the other models used within our experiments.

Carrier Sense Multiple Access with Collision Detection Protocol We adopted a model of the *Carrier Sense Multiple Access with Collision Detection Protocol (CSMA/CD)* also found at the website mentioned above [UPP]. We denote it as $CSMA/CD_x$, where x is the number of timed automata modeling stations. It models a broadcasting communication protocol in which several stations are trying to send data over a bus. In this scenario, only one station may send data at a time as otherwise two or more transmissions would be simultaneous and collide, s.t. no data can be received by a station listening to the bus. In this protocol, a station, willing to broadcast data, first senses whether the bus is busy. If so, it waits a random amount of time and then starts over. Otherwise, it starts sending data while listening to the bus for collisions, which may occur due to a propagation delay of the signal, denoting the time until it

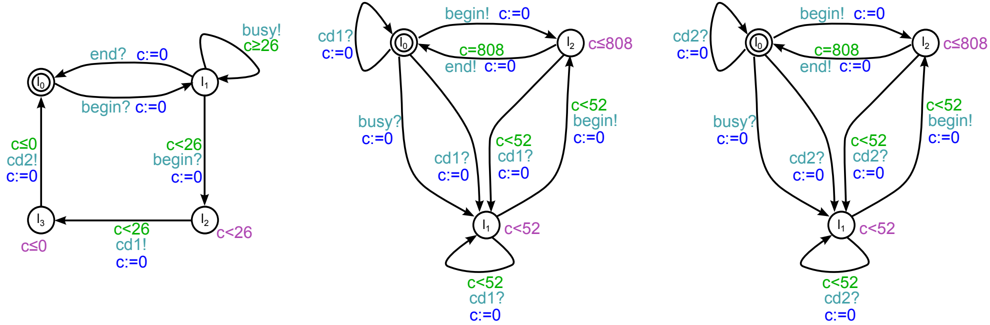


Figure 3.4: Network of three timed automata modeling the CSMA/CD protocol with a bus (left timed automaton) and two communicating stations (other two automata)

can be sensed by every station. In case a collision occurs, all stations stop sending and start over again. The model again denotes a mutual exclusion problem, where only one sender should be sending after the propagation delay time has passed.

Figure 3.4 shows the model for two stations. This model does not require a shared variable, but makes heavy use of synchronization channels. In particular, all stations are informed of a collision by the bus consecutively firing synchronized edges (using $cd1!$, $cd2!$, ...), one for each station.

The strict sequence of edges fired after a collision reduces the reachable portion of the state space. Our experiments show that the ratio of this reachable portion compared to the entire state space is an interesting characteristic with a wide influence to the utility of our algorithm. The safety property used in this CSMA/CD model specifies that the second station (automaton A_3) is not allowed to transmit (modeled as location l_2), when the first station (automaton A_2) is transmitting since more than 52 time units (the propagation delay). It is formally described as $\rho := G\neg((*, l_2, l_2), c_2 \geq 52)$, where c_2 denotes clock c in automaton A_2.

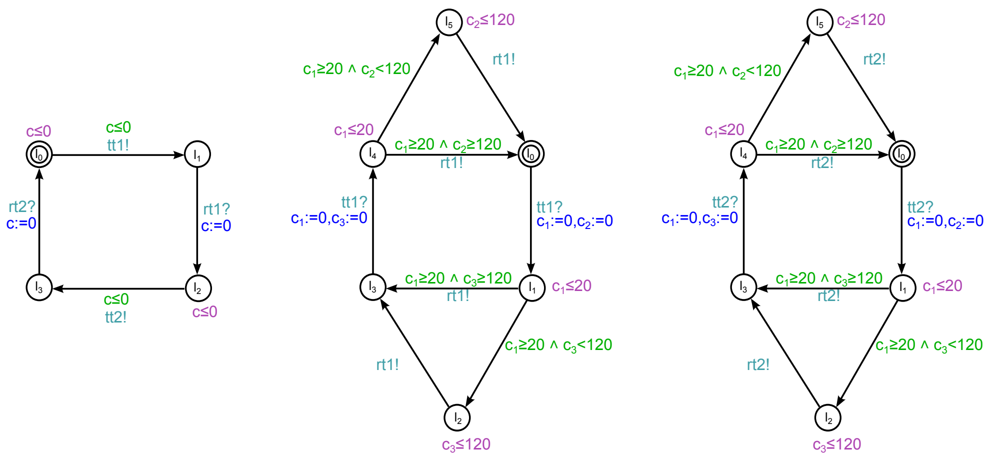


Figure 3.5: Network of three timed automata modeling the FDDI Token ring protocol with a model of the ring (left timed automaton) and two communicating stations

FDDI Token Ring Protocol The last benchmark adopted from the above mentioned Uppaal website [UPP] is the *Token Ring FDDI Protocol*. We denote it as $FDDI_x$, where x is the number of timed automata modeling stations. It models a situation in which several symmetric stations are ordered as a ring and a token is passed between the stations along the ring. In this scenario, there exist two options for the token passing, one is fast and the other is slow. The ring structure is modeled as a timed automaton that determines the sequence in which the token is passed from station to station, each modeled as an additional timed automaton. Due to the two different passing modes, this model includes many clocks and time constants, one of which is linearly dependent on the number of stations. Thus, the model is specifically suitable to examine the effect of large time constants. Furthermore, the reachable portion of the state space is an extremely small fraction of the entire state space due to the ring structure. This characteristic allows for interesting insight in the applicability of our technique.

Figure 3.5 shows the model for two stations. This model does not require a shared variable, but makes use of synchronization channels to enforce the stations only communicating in order of the ring.

The safety property specifies that the token is not at two stations at the same time. Thus, there must not exist a reachable state in which two stations are both in one of the following locations: l_1, l_2, l_4, l_5. Considering the mutual exclusion of only the first two stations, this safety property is formalized as

$$\begin{aligned}\rho := G\neg((*, l_1, l_1)) &\wedge \neg((*, l_1, l_2)) \wedge \neg((*, l_1, l_4)) \wedge \neg((*, l_1, l_5)) \\ \wedge\neg((*, l_2, l_1)) &\wedge \neg((*, l_2, l_2)) \wedge \neg((*, l_2, l_4)) \wedge \neg((*, l_2, l_5)) \\ \wedge\neg((*, l_4, l_1)) &\wedge \neg((*, l_4, l_2)) \wedge \neg((*, l_4, l_4)) \wedge \neg((*, l_4, l_5)) \\ \wedge\neg((*, l_5, l_1)) &\wedge \neg((*, l_5, l_2)) \wedge \neg((*, l_5, l_4)) \wedge \neg((*, l_5, l_5),\end{aligned}$$

where the integer constraint and zone *true* are each omitted for readability.

FDDI Token Ring Protocol with Counting Variable As can be seen, this way of specifying mutual exclusion is lengthy. Thus, we adapted the FDDI model by addition of an integer variable *cnt* that counts the number of automata currently in any of the mentioned locations. We denote it as $FDDIcount_x$, where x is the number of timed automata modeling stations. Figure 3.6 shows this adapted model for two stations. The safety property specifying mutual exclusion can, thus, be reduced to $\rho := G\neg(cnt > 1)$. This adaptation allows for an interesting study of the effect of an additional integer variable (with simultaneous reduction of the number of error state specifications).

Fischer Mutual Exclusion Algorithm (Bruttomesso) Additional mutual exclusion algorithms have been employed in our experiments. We adopted a different version of the Fischer Mutual Exclusion algorithm as presented by Bruttomesso et al. [Bru+12], denoted as *Fischer_B_x*, where x is the number of timed automata. It is depicted in Figure 3.7.

Figure 3.6: Adapted network of three timed automata modeling the FDDI Token ring protocol with two stations as in Figure 3.5 that includes an additional integer variable *cnt* used to count the number of stations that are in their respective transmitting locations

Figure 3.7: Network of two timed automata modeling the Fischer Mutual Exclusion algorithm for two processes as presented by Bruttomesso et al. [Bru+12]

Lamport and Shavit-Lynch Mutual Exclusion Algorithm Bruttomesso's publication also includes models formalizing two other mutual exclusion algorithms, namely the Lamport and Shavit-Lynch algorithms. The former solely relies on shared variables, while the latter additionally depends on timing constraints. We adopted a model for each of these two algorithms, denoted as $Lamport_B_x$ and $ShavitLynch_B_x$, where x is the number of timed automata. They are depicted in Figures 3.8 and 3.9.

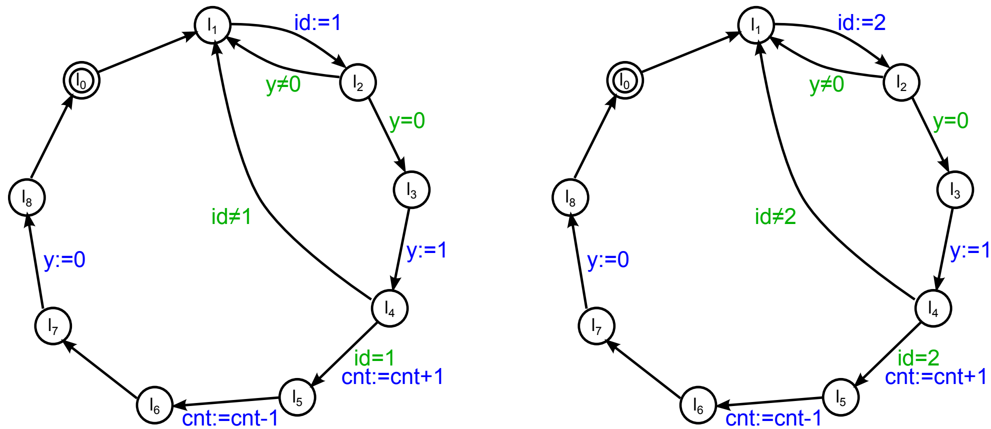


Figure 3.8: Network of two timed automata modeling the Lamport Mutual Exclusion algorithm for two processes as presented by Bruttomesso et al. [Bru+12]

Shrunk Model of Lamport Mutual Exclusion Algorithm Furthermore, we constructed a shrunk version of the Lamport model since some of the locations are not necessary. It is denoted as $Lamport_S_x$. The comparison of this shrunk version with the original one allows us to draw conclusions about the impact of a large number of locations in a model. The shrunk model is presented in Figure 3.10.

Shavit-Lynch Mutual Exclusion Algorithm (PAT) Additionally, we employ a version of the Shavit-Lynch algorithm in our experiments that includes fewer locations. The model is taken from the PAT website [PAT], denoted as $ShavitLynch_P_x$. It is depicted in Figure 3.11. We augmented all these models with a counting variable, s.t. the safety property specifying mutual exclusion can be formalized as $\rho := G\neg(cnt > 1)$.

Lemgo Model The applicability of our technique in real world scenarios is checked with an additional model. We developed it to represent a part of the *Lemgo Smart Factory* [inI] of the Hochschule Ostwestfalen-Lippe [Hoc]. It is similar in terms of size and structure to those timed automata models that are automatically learned from existing real world systems [Mai14]. It might, thus, be suitable to estimate the real world practicality of our approach.

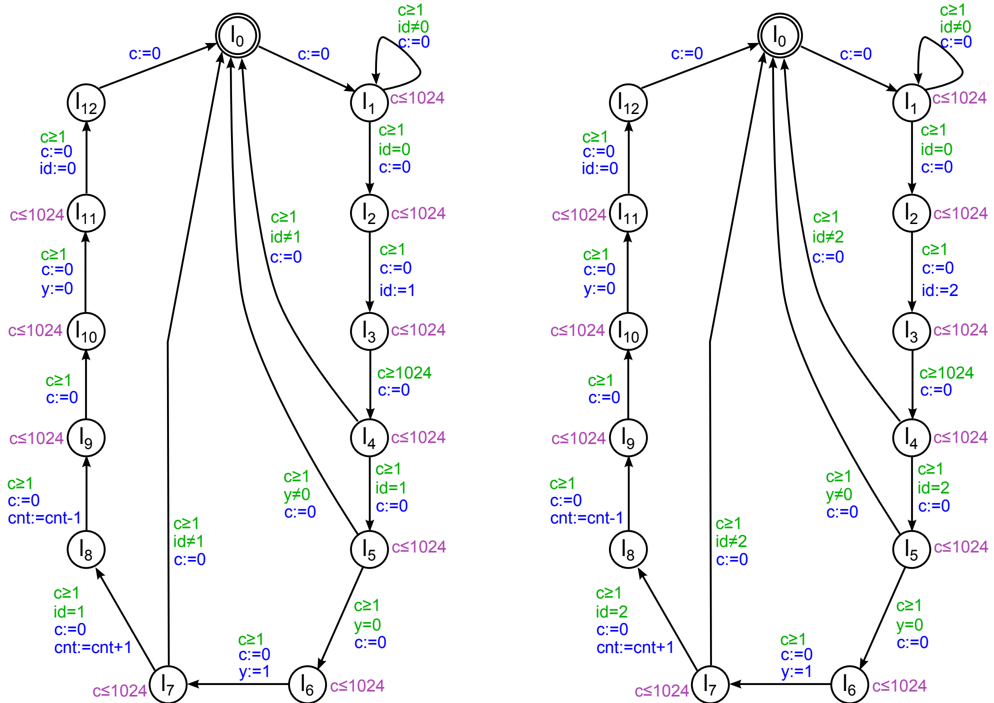


Figure 3.9: Network of two timed automata modeling the Shavit-Lynch Mutual Exclusion algorithm for two processes as presented by Bruttomesso et al. [Bru+12]

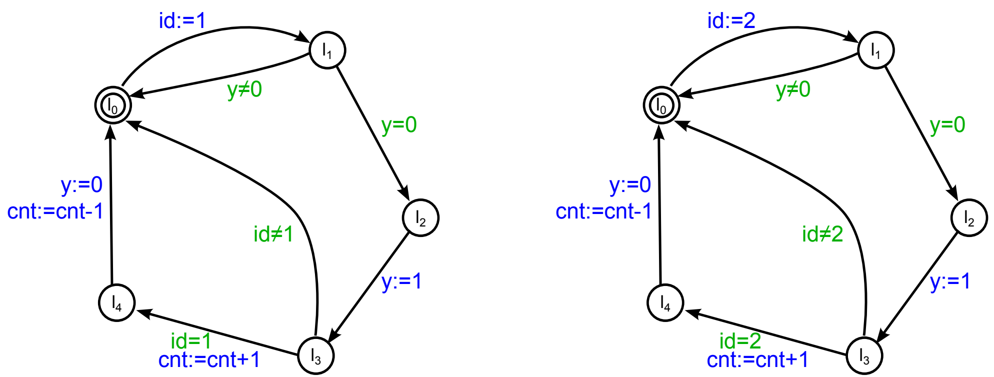


Figure 3.10: Shrunk Network of two timed automata modeling the Lamport Mutual Exclusion algorithm for two processes

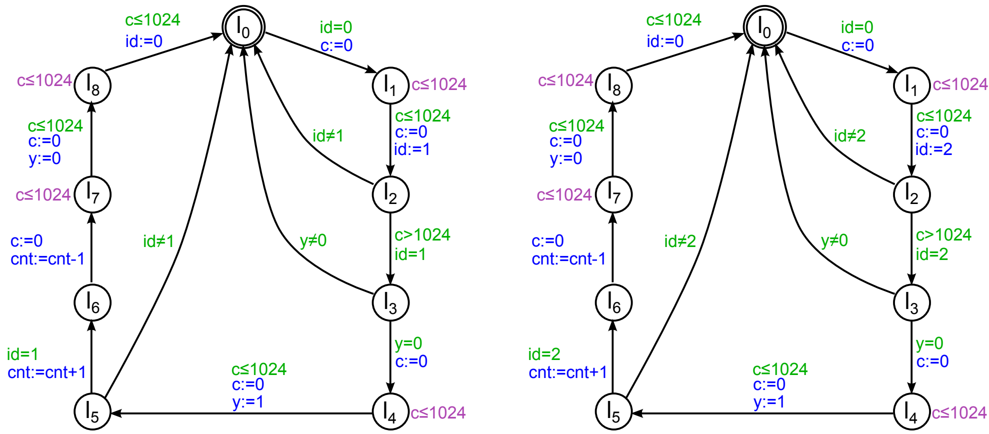


Figure 3.11: Network of two timed automata modeling the Shavit-Lynch Mutual Exclusion algorithm for two processes as found on the PAT website [PAT]

Figure 3.12 shows the model that depicts the interaction between a conveyor belt transporting bottles and a picker arm picking them up in order to store them elsewhere. The model is interesting as it can be used to show the effects of different time constants, as we will see in subsequent chapters.

A safety property of interest in this scenario requires the picking arm to be ready (location l_0) for a new bottle before the conveyor belt brings the next one as otherwise a bottle might be missed and confuse the rest of the system. Formally, this is expressed as $\rho := G\neg((l_0, l_1), true, true)$.

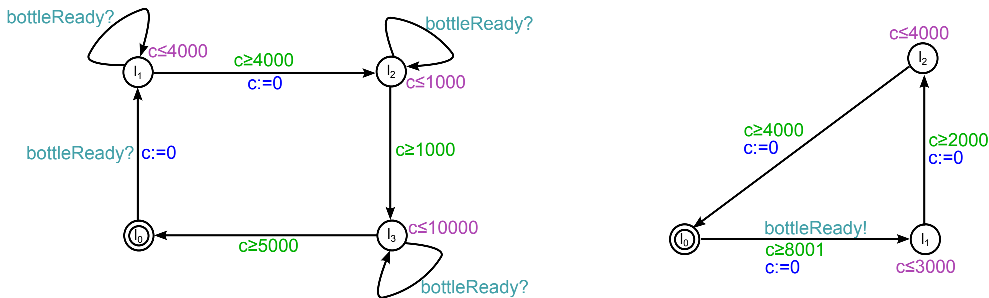


Figure 3.12: Network of two timed automata modeling the interaction between a conveyor belt (right) and a picker arm (left)

Using all the above models, we conducted numerous experiments in order to evaluate the performance and scalability of our technique. The used implementation is presented below.

3.5.2 Implementation

We implemented the algorithm presented above. The tool is written in Java version 7. It parses models and safety properties given in extended markup language (XML). After successful parsing, the verification of the safety property for the model is started. The algorithm starts with the two basic queries (lines 2 and 4) and afterwards runs its main loop (lines 7 to 15) constructing and refining the frames. It terminates whenever a counterexample is found, the property has been verified, a timeout occurred or the system ran out of memory. The implementation is tightly integrated with the used SMT-solver. We employ Z3 [DB08], version 4.3.2.0, using its API *Z3 for Java* [Leo12]. The experiments are performed on a pc with 3,2 GHz (AMD Phenom II X4 955). The timeout is set to 5 hours (18000 seconds).

Most parts of the algorithm are implemented straight-forward following the pseudo code given above. All parts concerning the usage and modification of zones entirely rely on the DBM data structure, i.e., they are stored and altered as matrices of clock differences. Whenever a zone needs to be encoded within an SMT-formula, the clock differences stored in the DBM cells are extracted and encoded such that those entries that do not contain information, e.g., clocks being unrestricted, are not used.

The implementation of the generalization procedure is similar to the one in the reference implementation of IC3. It sorts the literals according to a heuristic and checks whether the deletion of each literal changes the satisfiability of the query. To this end, it makes heavy use of the UNSAT-core: In case the query is still unsatisfiable after the deletion of a literal, the UNSAT-core denotes all literals that are used to prove unsatisfiability. Thus, all other literals can be discarded. This mechanism efficiently reduces the number of literals. A second element accountable for the efficiency of the generalization procedure is the order in which the literals are tried for deletion. For distinct orders the generalization procedure might result in very different clauses excluding varying states. Hence, in the successive cycles of the IC3 main loop, different CTIs may be found and the entire runs of IC3 might very much diverse. As a consequence, the runtime and memory consumption of a verification run are tremendously depending on the generalization procedure in general and its literal ordering in particular.

We have examined various heuristics for the ordering of the literals in the following subsection.

3.5.3 Heuristics for ordering Literals

In the IC3 reference implementation for hardware verification, the heuristic arranges the literals according to their history of use. Literals rarely occurring within the blocking clauses are preferred for deletion. This strategy seems to work fine in hardware verification, especially considering that few additional information is available that distinguishes these literals from one another.

For verification of timed automata, however, additional information is available, especially in the form of different types of literals. Our approach uses three distinct

	Fischer_U_15 (switched)	CSMA/CD_15	FDDI_15
Loc<Int<Clock	642,0	408	544
Loc<Clock<Int	481,3	408	575
Int<Loc<Clock	8546,7	405	514
Int<Clock<Loc	1743,0	323	836
Clock<Int<Loc	1742,6	321	376
Clock<Loc<Int	737,1	322	455
reference Heuristics	1780,0	344	501
random Order Range	602-2674	185-303	43-OOM

Table 3.1: Runtime (seconds) of benchmarks for different heuristics ordering the literals for *Generalization*, including the heuristic $Loc < Clock < Int$ that is used below

types of literals, namely location, clock and integer literals. Treating these categories differently, e.g., preferring location over clock literals for deletion, gives rise to six different orders. Additionally, the literals within each category can be sorted differently, e.g., using the heuristic from the reference implementation that prefers literals that are often used within blocking clauses.

We have explored these heuristics for some of the presented models. The results are depicted in Table 3.1.

It shows the runtime of our approach for some of the example models under different heuristics. The first column denotes the ordering of the heuristic, where $Loc < Int < Clock$ means that first location literals are tried for removal, then integer literals and at the end clock literals. The penultimate heuristic is the standard one from the IC3 reference implementation for hardware verification, ordering the literals according to their history of use as explained above. The last row shows the interval of runtimes for 10 runs with random literal ordering. The models are the Fischer, CSMA/CD and FDDI token ring models taken from the Uppaal website consisting of 15 timed automata each.

As can clearly be seen, none of the above heuristics is superior for all models. There might exist some characteristics, e.g., number of clocks or synchronized edges, in the model that might be exploited to indicate the preference of some of the heuristics. These might be extracted via syntactic analysis and used to decide which heuristic should be employed. In our evaluation, however, we stick to a fixed heuristic that works well for all the examined models, namely $Loc < Clock < Int$ (LCI). Thus, we leave the search for better heuristics, possibly in dependence of the models characteristic, as future work.

The results of the above experiments point out the tremendous impact of various heuristics for the performance of our approach. A badly chosen variable ordering is able to prevent a successful verification since the approach becomes inefficient and runs out of memory, as can be seen for the $FDDI$ model with random ordering. But even non-randomly chosen orderings can differ in efficiency very much, as can be seen for the heuristics $Loc < Clock < Int$ and $Int < Loc < Clock$ when verifying the

Fischer model. The verification using the latter heuristic is almost 18 times slower. Thus, the choice, which heuristic should be used, is of enormous importance.

In the following, we employ the $Loc < Clock < Int$ heuristic within the evaluation section, abbreviated as LCI. Note, however, that we also performed all experiments with the $Clock < Loc < Int$ heuristic (CLI) as can be seen in the resulting tables in the Appendix. The difference between both heuristics regarding scalability was marginal and, thus, only the LCI heuristic is shown in this section.

3.5.4 Experiments

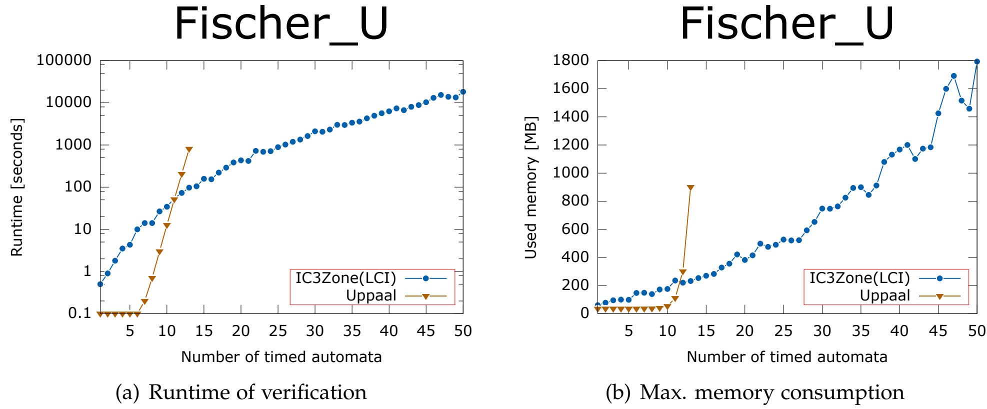
We conducted experiments for the benchmark models mentioned above. Aside from the Lemgo model, all of them can be scaled up to an arbitrary large network of timed automata. We use this characteristic in order to check the scalability of our approach. The runtime and memory consumption are taken during each of the experiments. The results of all experiments are given in Tables A.1 to A.10 in the Appendix. They are promising and speak in favor of the practicality of our method.

In the following, however, we will focus on analyzing specific aspects of the performance of our approach. Note, that all figures display the entire performance, i.e., if a runtime or memory consumption is not displayed, then the verification ran out of time or memory, except for the model $FDDIcount$.

General Scalability Experiments The general scalability of our approach can be summarized when considering Figures 3.13, 3.14 and 3.15. They show the performance of our approach in terms of runtime and memory consumption compared to the state of the art tool Uppaal [Beh+11], version 4.0.13, with default parameters. Clearly, Uppaal is superior for small models. For example, consider the experiments using the *Fischer_U* model with up to ten timed automata. The verification time for these models is significantly smaller using the Uppaal tool compared to our approach. This advantage is due to Uppaal's forward exploration algorithm in combination with smart data structures and optimizations. Our approach, however, includes the overhead of creating and using SMT-formulae and is, thus, not competitive for small models. When verifying safety properties for large models, however, this downside is reduced and our technique shows promising results.

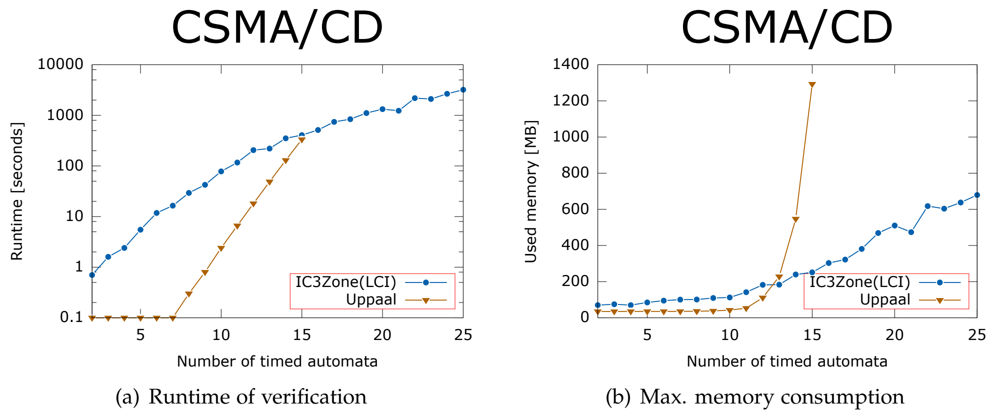
Uppaal needs enormous amounts of memory when verifying models with a large set of reachable states. This demand is observable in Figures 3.13 and 3.14, where the memory consumption of Uppaal increases rapidly for larger models until running out of memory for models with 14 and 16 timed automata, respectively. Our algorithm, however, is capable of verifying models with up to 50 timed automata, respectively. In addition to this capabilities of verifying large models that Uppaal can't, it is faster for some of the instances, e.g., the *Fischer_U* models with twelve and thirteen automata. It has, thus, proven to be competitive for some models with a large set of reachable states as Uppaal has difficulties with these models.

When considering Figure 3.15, we can clarify the performance for large models as the most basic distinction of our approach with Uppaal. While our technique



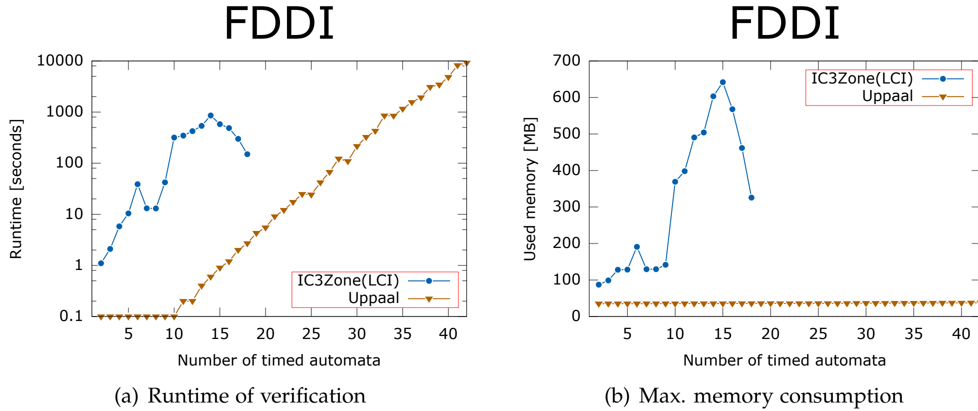
(a) Runtime of verification (b) Max. memory consumption

Figure 3.13: Results of the experiments using the *Fischer_U* model: Runtime and memory consumption of our presented approach *IC3 with Zones* are compared with state of the art tool Uppaal for models with different number of timed automata in the NTA



(a) Runtime of verification (b) Max. memory consumption

Figure 3.14: Results of the experiments using the *CSMA/CD* model: Runtime and memory consumption of our presented approach *IC3 with Zones* are compared with state of the art tool Uppaal for models with different number of timed automata in the NTA



(a) Runtime of verification

(b) Max. memory consumption

Figure 3.15: Results of the experiments using the *FDDI* model: Runtime and memory consumption of our presented approach *IC3 with Zones* are compared with state of the art tool Uppaal for models with different number of timed automata

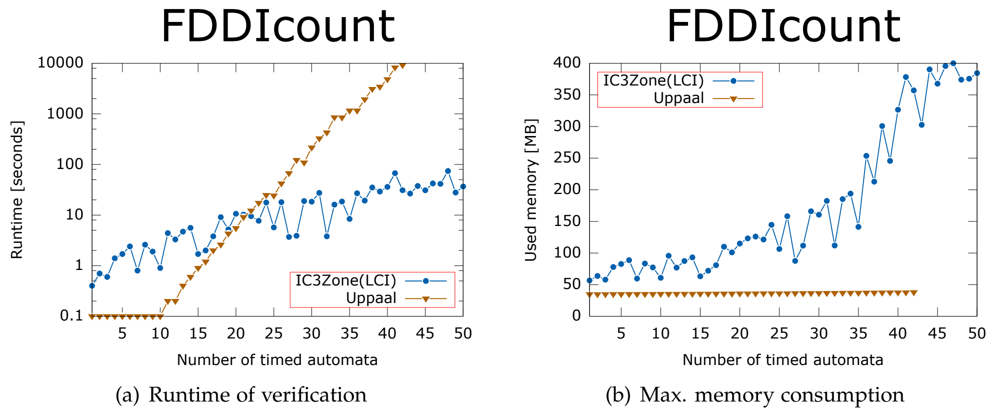
always examines the entire state space encoded via the used SMT-formulae, Uppaal only examines the reachable states via (forward) exploration. Thus, it has a huge advantage, if the ratio of reachable states to the entire number of states is small, as is the case in the FDDI model. Due to the ring structure, the order in which the automata's edges may be taken is very strict, reducing the number of reachable states. As an example, consider the FDDI model with 20 stations. During the entire verification of the safety property Uppaal explores only 8061 states, while the entire state space includes $4 * 6^{20}$ possible location combinations, not taking into account the additional clock space. It has, thus, an enormous advantage over our approach exactly for those models with a small number of reachable state, where the entire state space is large.

The two distinct ways of exploring the model's states result in different suitabilities of the approaches for different models. In summary, we would recommend using Uppaal for smaller models and those with a small set of reachable states (hard to estimate upfront). Nevertheless, this characteristic might be guessed by means of the structure of the model, e.g., when the model allows only very specific sequences of edges being taken. On the other hand, we recommend the approach presented in this chapter to be used for larger models. It has shown to scale well and is competitive to Uppaal for some of the benchmark models. We emphasize the fact that it was capable of verifying safety properties for models three times larger than Uppaal was capable of for the *Fischer_U* model.

Next, we examine the impact of integer variables on our technique.

Integer Variable Experiments Scaling up the number of integer variables may have various effects depending on their usage. It may entirely change the state space, or may only be of little effect. Thus, we can only exemplify possible impacts of integer variables here. To this end, we will examine the change introduced by the addition of the integer variable *cnt* in the *FDDI* model.

Our adapted *FDDI* model includes an additional integer variable *cnt* that counts the number of automata in transmitting locations. Six out of eight edges are extended by an integer update in each of the automata modeling a station. The introduction of this additional variable has an interesting effect. It simplifies the found inductive strengthening for every number of automata to include only two clauses. The first one is the safety property $\neg(int_0 > 1)$ with int_0 encoding the integer variable *cnt*. The second one $\neg(\neg l_0^1 \wedge (int_0 > 0))$ specifies (via the least significant bit of the location identifiers) that no location of the first automaton (the ring structure) with an even identifier $(l_0, l_2, \dots)$ is reachable when $cnt > 0$. Since these are the only locations with outgoing edges incrementing *cnt*, the counter can not be larger than 1. The conjunction of these two clauses is inductive. This verification heavily profits from the SMT-encoding of locations via boolean variables, as well as the ring structure of the model including two locations for each station. It allows a found CTI to be generalized, s.t. it reasons about all locations of A_1 with *even* location identifier. Figure 3.16 shows the impact on the runtime, which is extremely improved, actually able to outperform Uppaal.

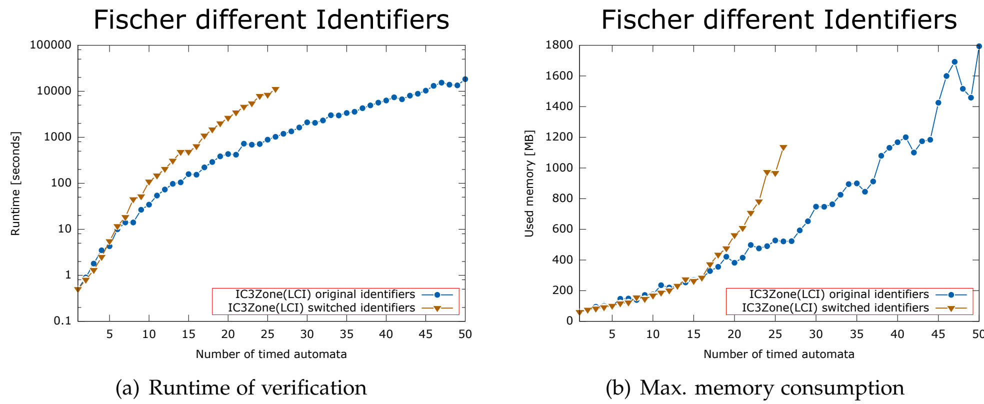


(a) Runtime of verification (b) Max. memory consumption

Figure 3.16: Results of the experiments using the *FDDIcount* model: Runtime and memory consumption of our presented approach *IC3 with Zones* are compared with state of the art tool Uppaal for models with different number of timed automata in the NTA

The positive impact of the location encoding in combination with the generalization procedure becomes apparent by this example. Yet, the presented encoding introduces additional interesting effects.

Location Identifier Experiments With our SMT-encoding relying on boolean variables to encode the identifiers of locations, we were interested whether the actual assignment of identifiers to locations affects the performance of our approach. To this end, we examined the runtimes for the verification of the mutual exclusion property for the *Fischer_U* models as above, where each location l_i ($i \in \{0,1,2,3\}$) has identifier i. Additionally, we performed experiments with a model *Fischer_U*$(switched)$, where location l_0 and l_2 switched identifiers, meaning l_0 has identifier 2 and l_2 has identifier 0. The results are shown in Figure 3.17.



(a) Runtime of verification

(b) Max. memory consumption

Figure 3.17: Results of the experiments using the *Fischer_U* models (Figure 3.1) with distinct assignment of identifiers to the locations: The verification has been executed using the assignment of location identifiers as used before, where l_i ($i \in \{0,1,2,3\}$) is assigned identifier i, and also with switched assignments of identifiers for l_0 and l_2, s.t. l_0 is assigned identifier 2 and l_2 is assigned 0

Clearly, the assignment of identifiers is of importance, as the runs with switched identifiers exhibit worse performance. Comparing and analyzing these runs did not result in a definite answer to why the runs differ that much.

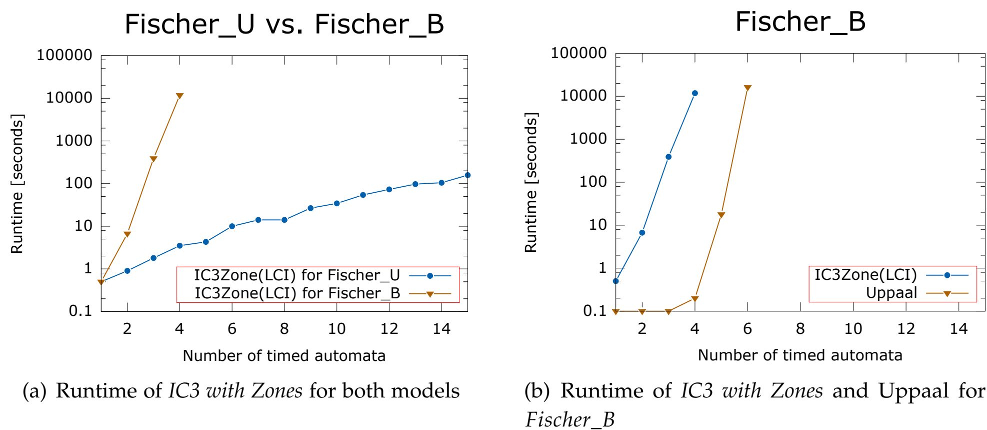
In these experiments, switching the identifiers reduced the number of cycles of the main loop with fewer CTIs found. Thus, the loss of performance must have different reasons. We noticed that the instances with switched identifiers show a significantly larger number of SMT-queries. This increase might be due to the generalization procedure needing more attempts to discard literals, or due to frames and CTIs being different resulting in the procedures blockCTIs and strengthenClauses issuing more queries.

In general, it is not feasible to chose the assignment of identifiers upfront such that the performance is optimized. This drawback of the location encoding via boolean variables is opposed to the wider generalization capabilities as presented before. Thus, the encoding can be summarized as ambivalent.

Below, we will examine the effects of the number of locations in a model.

Location Experiments Our encoding introduces many variables by relying on the locations being encoded via boolean variables. Thus, more literals have to be tried to be discarded during generalization, leading to diverse effects. On the one hand, the increased number of variables introduces additional overhead in the SMT-formulae and in the generalization procedure. However, as seen above it can be extremely beneficial in reasoning about several locations in a single automaton during generalization. In the following, we examine models with additional locations to get a more general view on the impact of locations. To this end, we compare the *Fischer_B* model with the *Fischer_U* model since they both model the same algorithm. They use the same number of clocks and integer variables, which are used in the same way, only differing in smaller details, e.g., the invariants. The most important distinction, however, is that the larger model includes five additional location, which is more than twice as much as the smaller model.

Figure 3.18(a) shows the comparison of both models for up to 15 automata in the NTA. Note, that the safety property could be proved for *Fischer_U* models up to 50 automata as shown above. In order to relate the performance change of our algorithm for the larger model *Fischer_B*, Figure 3.18(b) shows the comparison with Uppaal's performance.



(a) Runtime of *IC3 with Zones* for both models

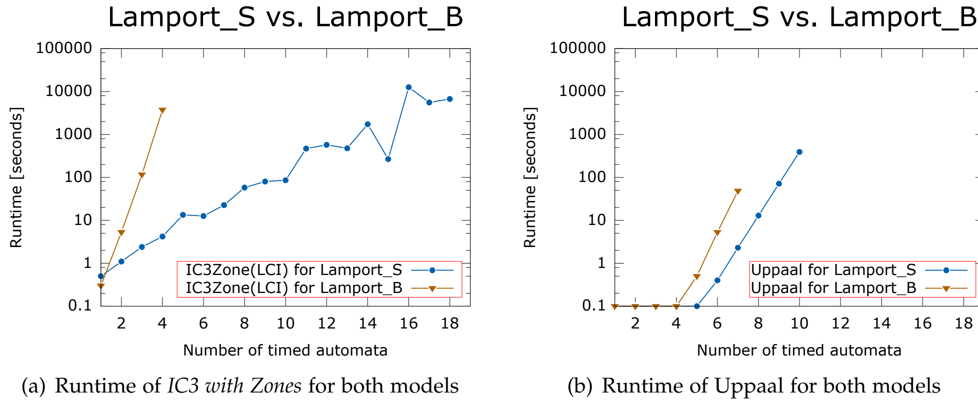
(b) Runtime of *IC3 with Zones* and Uppaal for *Fischer_B*

Figure 3.18: Results of the experiments comparing the performance of the verifications using the two *Fischer* models: *IC3 with Zones* performs significantly worse for the *Fischer_B* model

The experiments show that our approach performs significantly worse in the presence of many locations. The decline is more intense than the one observed for Uppaal, possibly due to our location encoding being based on boolean variables, which introduces overhead if the number of locations is large. The deterioration might, however, also be due to an unsuitable location identifier assignment as seen in the previous paragraph or due to the general size of the model.

We extend this insight with the comparison of runtimes for the verification experiments using the *Lamport_B* and *Lamport_S* models, which only differ by the

number of locations. The runtimes of our approach for both models are compared in Figure 3.19, as well as the runtimes of Uppaal for both models.



(a) Runtime of *IC3 with Zones* for both models

(b) Runtime of Uppaal for both models

Figure 3.19: Results of the experiments comparing the performance of the verifications using the two *Lamport* models: *IC3 with Zones* performs significantly better for the shrunk *Lamport* model

Clearly, our presented approach reacts worse to the additional locations in the larger model. The increase of runtime is worse than the one for Uppaal. The conclusion drawn before is, thus, supported by this experiment. Our algorithm handles models with many locations (in the single automata) worse than Uppaal. Experiments with the two distinct Shavit Lynch models reach the same conclusion (see Tables A.7 and A.8 in the Appendix).

We checked whether the bad scalability in terms of locations is caused by the location encoding via boolean variables. To this end, we replaced this encoding via several boolean variables with an alternative encoding of the locations via a single integer variable per automaton. The resulting performance was better, but the increase was not as significant as expected. However, the loss of the ability to generalize the locations within a single automaton (as possible with boolean variables) led to a significantly worsened performance with the *FDDIcount* model. In summary, the usage of an adapted encoding is possible and might even yield a better performance for some models, but we recommend the usage of the encoding presented in this chapter due to the improved generalization ability.

In the following, we compare our approach with the previous attempt to utilize IC3 for timed automata verification in order to point out the advancement of our approach. We compare it with Kindermann's IC3 approach using the region abstraction [KJN12b]. For a fair comparison regarding the used programming language, SMT-solver, encoding and heuristic, we reimplemented his approach.

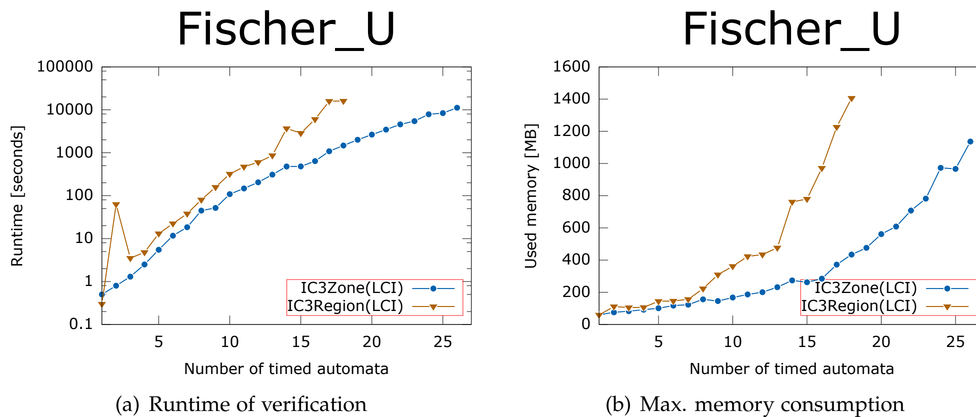
Used Constant	IC3Regions(LCI)		IC3 with Zones(LCI)	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
1	1186,2	669,6	480,2	264,4
4	2174,3	740,2	483,2	262,6
16	2179,7	741,3	482,6	263,9
64	2860,0	779,7	481,5	263,1
256	2628,4	742,3	483,0	263,7
1024	2860,5	778,3	481,9	264,6

Table 3.2: Scalability experiments with distinct time constants in the *Fischer_U_15(switched)* model with 15 processes

Experiments with the used Abstraction The employed region abstraction renders Kindermann's approach non-practical due to the enormous number the regions. This insignificance of the region abstraction for practical purposes has been explained in the previous chapter. In particular, its exponential growth in dependence on the size of time constants is a crucial argument.

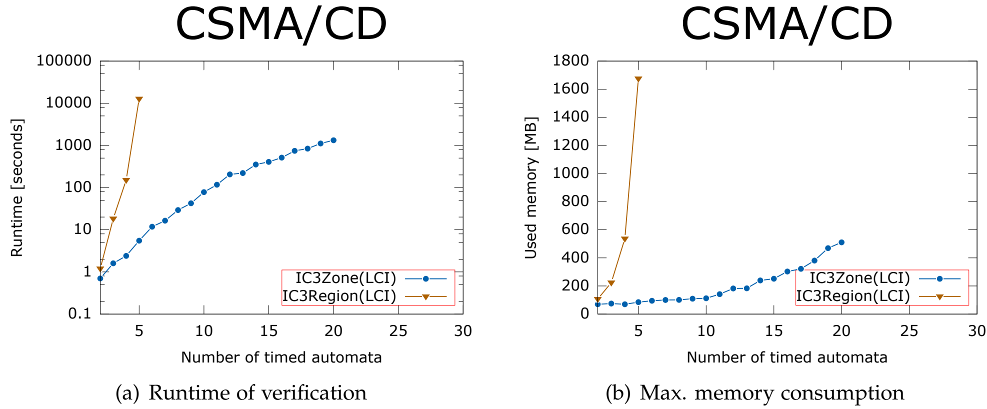
Table 3.2 illustrates this significant drawback. The safety property is verified for *Fischer_U(switched)* models with 15 processes and different time constants. This experiment perfectly shows the lack of the region-based technique to cope with large constants, while our approach is unaffected.

For illustration of this advantage, we present the runtimes of Kindermann's approach for the *Fischer_U(switched)*, *CSMA/CD* and *FDDI* model in Figures 3.20, 3.21 and 3.22. These experiments clearly show the improvement accomplished by using the zone abstraction.



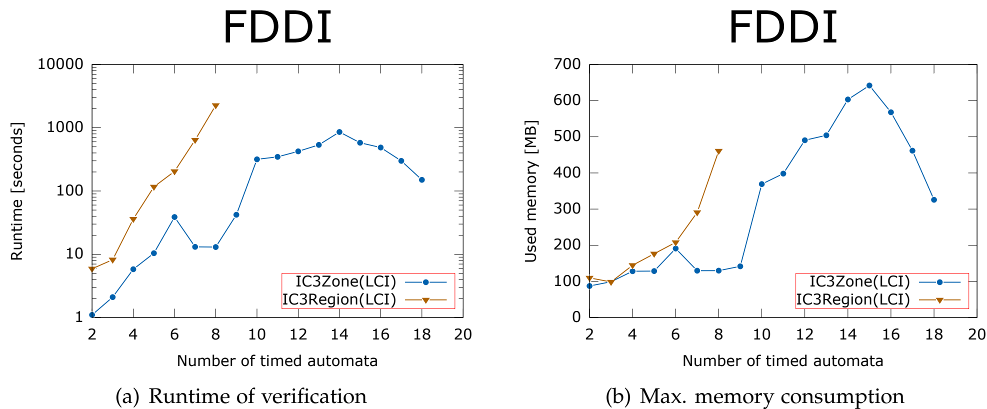
(a) Runtime of verification (b) Max. memory consumption

Figure 3.20: Results of the experiments using the *Fischer_U(switched)* model: Runtime and memory consumption of our presented approach *IC3 with Zones* are compared with the previous approach using the region abstraction for models with different number of timed automata in the NTA



(a) Runtime of verification (b) Max. memory consumption

Figure 3.21: Results of the experiments using the *CSMA/CD* model: Runtime and memory consumption of our presented approach *IC3 with Zones* are compared with the previous approach using the region abstraction for models with different number of timed automata in the NTA



(a) Runtime of verification (b) Max. memory consumption

Figure 3.22: Results of the experiments using the *FDDI* model: Runtime and memory consumption of our presented approach *IC3 with Zones* are compared with the previous approach using the region abstraction for models with different number of timed automata in the NTA

In order to test the practicality one last time, we verified the mentioned safety property for the *Lemgo* model presented above. Table 3.3 shows the runtime and memory of our approach when verifying the real world example from Lemgo. It is apparent, that the performance is sufficient and the presented approach is of practical value.

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime	Memory	Runtime	Memory	Runtime	Memory
Lemgo model	0,7	79,1	0,7	75,2	0,1	35,1

Table 3.3: Experiments using our *Lemgo* model (Figure 3.12): The runtimes (seconds) and memory consumption (MB) are depicted for our approach *IC3 with Zones* with two distinct heuristics for variable ordering (LCI and CLI), as well as for the tool Uppaal

3.5.5 Inductive Strengthening Experiments

In addition to the verification results that are "There exists a counterexample trace, i.e., the property is not invariant" or "The property is invariant", our technique yields an extra outcome in the latter case. It computes an inductive strengthening of the safety property, which is a formula encoding an inductive set of states that all satisfy the safety property. Due to its inductiveness, it trivially includes all reachable states (which means it is an overapproximation). It can, thus, be used as an easy means to validate a successful verification. To this end, the three properties of inductive strengthenings (Definition 2.4.1) have to be checked. This check again employs SMT-solving and usually significantly outperforms a complete re-verification. As a result, the additional outcome in form of an inductive strengthening can be of huge importance.

For illustration, consider the following scenario that involves two parties. On the one hand, there exists a client that wants to verify a safety property for a specific model. However, his resources in form of time and memory are usually restricted, s.t. he cannot execute the verification himself. On the other hand, there exists a provider, e.g., a computing center, that possesses the necessary resources and is, thus, capable of executing the desired verification. The important aspect in such scenarios is the question whether the provider is trustworthy and the client can trust the result of the provider. At this point a certificate, here the inductive strengthening, can be employed. To this end, the provider ships the certificate to the client. Since the validation of the certificate is by far easier than the entire verification, the client is able to validate it using his limited resources. If the shipped formula is indeed an inductive strengthening of the desired safety property, the client can be sure that the safety property is invariant. Such approaches are called *proof-carrying* and are widely employed in several domains [Nec02], [DKP09].

We have performed a validation of the inductive strengthenings found during our experiments in the previous chapter. The benefit of validation compared to

verification is obvious when considering the runtimes in Tables A.11 to A.21 with Tables A.1 to A.10. As can be seen, most inductive strengthenings found in our experiments can be validated within seconds. The speed up in comparison with an entire verification run is significantly better, the larger the instance. In particular, there exist smaller instances, where almost no speed up was measurable. In contrast, in some larger instances the required time for validation was as small as roughly 0,02% of the time required for verification. With its small size (max. 2 MB in our experiments) the inductive strengthenings computed by our technique are perfectly suitable to be used in certifying scenarios.

In addition, they are of value in other scenarios, where we speed up the verification of a reconfigured model or reason about an entire family of models. These approaches are presented in the subsequent chapters.

3.6 Summary

In summary, we have presented an approach for the verification of safety properties for networks of timed automata. To this end, we have developed a concept that combines the successful algorithm IC3 with the Zone abstraction to employ the runtime and memory efficiency of it for the verification of timed systems.

We have given an SMT-encoding that is designed with IC3's generalization procedure in mind. Using backwards computation, we have shown how to incorporate the Zone abstraction into the algorithm. The necessary modifications are only punctual and, thus, allow the usage of most of the optimizations proposed for IC3 so far and in the future.

We have implemented the concept in Java and evaluated its strengths and weaknesses in numerous experiments.

The presented technique scales well and shows promising results. It is competitive to state-of-the-art tools in many instances. However, its weakness is the dependence on the size of the entire state space. In contrast, other tools only depend on the size of the reachable portion of the state space.

Taking into account the additional outcome for a successful verification in form of an inductive strengthening of the safety property, we conclude that the presented technique is definitely of value (for example in proof-carrying approaches) and yields promising results. This additional outcome is fundamental for the two approaches presented in the next chapters, in which it is used to speed up the verification of a reconfigured model and to reason about a family of models.

4 Incremental Inductive Verification of Parameterized Timed Systems

In the previous chapter, we have proposed a novel combination of two well-known techniques that can be used to verify safety properties for timed automata. Aiming at model-based design processes, this technique is well suited to be employed for the formal verification of safety properties for real-time systems. With an increasing number of systems being time-based and safety critical, there exists a large demand for such techniques.

In some structured model-based design processes, an iterative procedure triggers a repeated reconfiguration of the model until a final design is found. A similar scenario is the reconfiguration at lifetime of a system, e.g., due to self-adaptation. In these scenarios, there exists a need to verify properties for reconfigured models, which have already been verified before for the original model.

Like other state-of-the-art techniques, the previously presented approach does not cover these scenarios. It verifies a safety property for a fixed model. In order to cope with a reconfigured model, it would restart the verification from scratch like in other approaches. It is, however, likely that the reconfigured model is only slightly changed. Thus, it is desirable and promising to reuse the results of the previous verification. We study this reuse in the following. Chapters 4 and 5 propose a workflow for a restricted subset of models and reconfigurations, while Chapter 6 explores the general case.

Consider the specific setting, where the system consists of an arbitrary, but fixed number of equal processes. Though this setting is specific, it includes a large number of situations, e.g., processes communicating via a peer to peer protocol or executing a mutual exclusion algorithm. Using the technique from the previous chapter enables us to verify safety properties for each such given model with a fixed number of processes. We are, however, interested in the verification of the safety property for every such model as the number of processes is arbitrary. For illustration consider a system $[P_1||\dots||P_n]$ with a fixed number n of processes ensuring mutual exclusion. A reconfiguration that adds an additional process requires an new verification for

the reconfigured model $[P_1||\ldots||P_n||P_{n+1}]$. In contrast, it would be preferable to verify a property ϕ for the system for every arbitrary number n of processes given as parameter, formally $\forall n \in \mathbb{N} : [P_1||\ldots||P_n] \models \phi$. Such a family of models specifying an arbitrary, but fixed number of equal processes is denoted as Parameterized System. In case timed models are employed, we speak of *Parameterized Timed System* [Bru+12]. The term *Parameterized* denotes the dependence of the number of processes from a parameter. In general, these systems can be parameterized with many parameters, s.t. each one of them specifies the number of processes of a certain class, e.g., a system with two classes of processes would require two parameter. However, in this thesis we only consider systems with a single class of processes. Note, that the verification question even for untimed parameterized systems is undecidable in general [AK86]. Yet, there exist numerous works concerned with decidable fragments or semi-algorithms.

The latter is in spirit with our work presented in this chapter. We verify single instances (with a fixed number of processes) of the parameterized timed system and try to generalize the result for all larger instances. In particular, our approach [Ise15] heavily relies on the reuse of inductive strengthenings as detailed below.

Reuse of Inductive Strengthenings Reconfigurations (changes) of models may occur frequently within the design phase and thereafter, in particular in the context of Industry 4.0, where systems are adaptive. They can generally be classified as the addition, deletion or modification of parts of the system, respectively model. As an example, consider the transformation of a programmable logic controller program into timed automata [Wil99]. The addition of a new statement in the program might result in an additional location, while the rest of the model might be analog to the original one. On the other hand, the removal of a statement might result in the removal of a location.

In the present chapter, we only consider specific reconfigurations and models. As explained above, we work with parameterized timed systems that consist of a fixed, but arbitrary number of instantiations of the same process. To this end, our models are networks of symmetric timed automata, which are all equal (up to certain restrictions) and the considered reconfiguration is the addition of a symmetric automaton. The verification question of interest asks whether a safety property is invariant for any number of symmetric automata in the network. So far, our approach presented in Chapter 3 allows the verification of a safety property only for a fixed model. We would, thus, need to run the approach for a model with 1 timed automaton, 2 timed automata, 3 timed automata and so on. Trivially, this can not be done in finite time and, in addition, the verification becomes infeasible for huge models, e.g., with one million timed automata in the network.

As a consequence, we propose a workflow that is able to reason about all models of a parameterized timed system, based on the verification of several of the fixed instances. To this end, we employ the inductive strengthenings that are computed by our algorithm *IC3 with Zones* presented in the previous chapter. They are a perfect means to validate a successful verification as shown in Subsection 3.5.5.

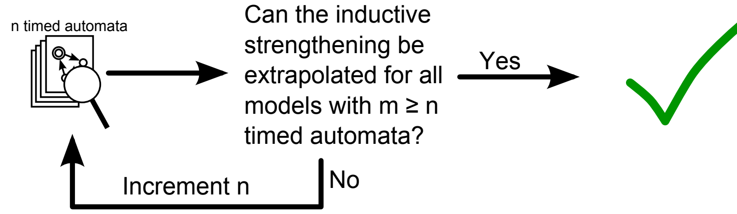


Figure 4.1: Overview of our incremental workflow

Considering a reconfigured model, however, their utility is not obvious. An inductive strengthening of the safety property for the original model will usually not be a valid inductive strengthening for the reconfigured model. The reason is most often that it is not inductive for the new model. Yet, it may still be of value. We propose an iterative procedure specifically designed to adapt and employ a computed inductive strengthening in scenarios with symmetric models. Figure 4.1 conveys the idea. We verify a model with a fixed number n of timed automata and reuse the inductive strengthening to reason about all models with larger number of automata. If the reasoning step can not be applied, we start over with the verification of the next larger model with $n+1$ automata. However, if the reasoning step can be applied, we have guaranteed the safety property to be invariant in each model of the entire parameterized timed system. In the following, we take a closer look at the inductive strengthenings computed by our algorithm *IC3 with Zones* to give an intuition why their reuse is of value in this symmetric setting.

Observations Several similarities can be found between some of the clauses in the inductive strengthenings. As example consider the *Fischer_U* model from the previous chapter for one, two and three timed automata (*Fischer_U_*1, *Fischer_U_*2 and *Fischer_U_*3). Table 4.1 shows the respective inductive strengthenings of the safety property $\rho := G\neg(cnt > 1)$ computed during the experiments.

Fischer_U_1	*Fischer_U_2*	*Fischer_U_3*
$(int_1 \leq 1)$	$(int_1 \leq 1)$	$(int_1 \leq 1)$
$\wedge(l_0^1 \vee (int_1 \leq 0))$		
$\wedge(l_1^1 \vee (int_1 \leq 0))$		
	$\wedge((int_0 \neq 0) \vee (int_1 \leq 0))$	$\wedge((int_0 \neq 0) \vee (int_1 \leq 0))$
	$\wedge(\neg l_0^1 \vee l_1^1 \vee (int_1 \leq 0))$	$\wedge(\neg l_0^1 \vee l_1^1 \vee (int_1 \leq 0))$
	$\wedge(\neg l_0^2 \vee l_1^2 \vee (int_1 \leq 0))$	$\wedge(\neg l_0^2 \vee l_1^2 \vee (int_1 \leq 0))$
		$\wedge(\neg l_0^3 \vee l_1^3 \vee (int_1 \leq 0))$
	$\wedge(l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$	$\wedge(l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$
	$\wedge(l_0^2 \vee (int_0 \neq 2) \vee (int_1 \leq 0))$	$\wedge(l_0^2 \vee (int_0 \neq 2) \vee (int_1 \leq 0))$
		$\wedge(l_0^3 \vee (int_0 \neq 3) \vee (int_1 \leq 0))$
	$\wedge((c_0^1 \leq 1024.0) \vee (int_0 \neq 1) \vee (c_0^2 > 1024.0))$	$\wedge((c_0^1 \leq 1024.0) \vee (int_0 \neq 1) \vee (c_0^2 > 1024.0))$
		$\wedge((c_0^1 \leq 1024.0) \vee (int_0 \neq 1) \vee (c_0^3 > 1024.0))$
	$\wedge((c_0^2 \leq 1024.0) \vee (int_0 \neq 2) \vee (c_0^1 > 1024.0))$	$\wedge((c_0^2 \leq 1024.0) \vee (int_0 \neq 2) \vee (c_0^1 > 1024.0))$
		$\wedge((c_0^2 \leq 1024.0) \vee (int_0 \neq 2) \vee (c_0^3 > 1024.0))$
		$\wedge((c_0^3 \leq 1024.0) \vee (int_0 \neq 3) \vee (c_0^2 > 1024.0))$
		$\wedge((c_0^3 \leq 1024.0) \vee (int_0 \neq 3) \vee (c_0^1 > 1024.0))$

Table 4.1: Inductive strengthenings computed for the Fischer models with distinct number of automata during the experiments in the previous chapter

Within each of the inductive strengthenings, there exist some clauses that are extremely similar, meaning they have the exact same structure of literals. They differ only by the timed automata that are referred. A perfect example are the fifth and sixth clauses $((l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$ and $(l_0^2 \vee (int_0 \neq 2) \vee (int_1 \leq 0)))$ in the inductive strengthening found for the model *Fischer_U_2* (second column). Their structure is the same, but the former clause refers to timed automaton A_1 and the latter one refers to timed automaton A_2. The reason is located in the symmetry of the Fischer model. It results in different CTIs being equal up to symmetry including their generalized clauses.

In addition, there exist clauses that can be related across distinct inductive strengthenings. These clauses are written on the same row (but distinct columns) in Table 4.1, e.g., the clause $(l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$ that can be found in the inductive strengthening of the safety property for models *Fischer_U_2* and *Fischer_U_3*.

We propose to utilize these symmetry-related clauses in a workflow that incrementally verifies the safety property for symmetric networks of timed automata with an increasing number of automata. This workflow reuses inductive strengthenings of the safety property found in smaller models for the verification with larger networks of timed automata by exploiting the symmetric nature of the models.

We start with the definition of the subclass of models that can be verified using the technique presented in this chapter. To this end, we carefully restrict the allowed operations. We formalize the intended notion of symmetry and ensure it by definition of our models as templates. Finally, we present the workflow including optimizations and evaluate the practicality and performance of it.

4.1 Restrictions

The definitions in Chapter 2 allow for rather general networks of timed automata. In order to be able to profit from symmetry when reusing inductive strengthenings, we have to restrict the allowed models. To this end, a model is only permitted to consist of symmetric timed automata, which all contain the same locations and edges.

All constraints and updates in these automata are exactly the same (but refer to their respective local clocks). There are, however, specific restrictions. For simplicity reasons, we disallow synchronization channels in this chapter. In our extension of the formalism in Chapter 5 they are again allowed in a limited way. Furthermore, we introduce a separation of integer variables according to their intended use.

There exist general purpose integer variables employed unaware of the fact that several timed automata exist. Thus, they have to be used in exactly the same way in each of the automata. In addition, there exist those variables that can be used distinctively in each of the automata by usage of the identifiers, e.g., for signaling purposes.

The former set of integer variables is denoted as $\mathcal{IV}_{\not{id}}$, as it is employed without knowledge of the identifiers, while the latter set is denoted $\mathcal{IV}_{id}$. The entire set of

integer variables is the union of these two disjoint sets, formally $\mathcal{IV} = \mathcal{IV}_{\not{id}} \cup \mathcal{IV}_{id}$ with $\mathcal{IV}_{\not{id}} \cap \mathcal{IV}_{id} = \emptyset$. We exemplify both sets in the following.

As an example for identifier unaware integer variables consider the variable *cnt* in the *Fischer_U* model (Figure 3.1). In each of the timed automata, it is used exactly the same way, namely $cnt := cnt + 1$ and $cnt := cnt - 1$ at exactly the same edges.

As example for identifier aware integer variables consider the variable *id* in the *Fischer_U* model (Figure 3.1). It is compared to the neutral value 0 and each timed automaton's identifier, e.g., 1 and 2. In addition, each automaton assigns the neutral value 0 to the variable (see edge between l_3 and l_0) or assigns its own identifier to the variable (see edge between l_1 and l_2). The usage of the identifier can be abbreviated as $id == pid$ and $id := pid$, where *pid* needs to be instantiated with each timed automaton's identifier.

The ability to reason about specific timed automata induces some restrictions on the allowed usage as otherwise the symmetry could be broken. The following definitions redefine the usage of integer variables and overwrite Definitions 2.1.4 and 2.1.5 of Chapter 2.

Definition 4.1.1. Let $\mathcal{IV} = \mathcal{IV}_{\not{id}} \cup \mathcal{IV}_{id}$ be a set of integer variables split into variables unaware/aware of identifiers. Mapping each integer variable $iv \in \mathcal{IV}$ to a value $v^i(iv) \in \mathbb{Z}$ is called an *integer valuation* v^i.

Each of the two distinct sets of variables allows for different constraints.

Definition 4.1.2. $\Psi_{\not{id}}(\mathcal{IV})$ is the set of *integer constraints* ψ for identifier unaware integer variables defined by $\psi := iv \bowtie n | \psi_1 \wedge \psi_2 | true$ with $iv \in \mathcal{IV}_{\not{id}}$, $n \in \mathbb{Z}$, $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$ and $\psi_1, \psi_2 \in \Psi_{\not{id}}(\mathcal{IV})$.

Definition 4.1.3. For a given identifier $pid \in \mathbb{N}_{\geq 1}$, let $\Psi_{id}(\mathcal{IV}, pid)$ be the set of *parameterized integer constraints* $\psi(pid) := iv \bowtie n | \psi_1 \wedge \psi_2 | true$ with $iv \in \mathcal{IV}_{id}$, $n \in \{0, pid\}$, $\bowtie \in \{=, \neq\}$ and $\psi_1, \psi_2 \in \Psi_{id}(\mathcal{IV}, pid)$, defined for identifier aware integer variables.

The variables aware of identifiers are only allowed to be compared with a neutral value 0 or a specific identifier given as parameter *pid*, which will later be instantiated by the identifier of each automaton. These restrictions still allow a usage of the integer variables for signaling, e.g., checking whether no timed automaton is interested ($iv = 0$), or whether an automaton is still the only one interested ($iv = pid$) after a respective assignment. In addition, several comparison operators are prohibited since they destroy symmetry. Figure 4.2 shows an example for the usage of unbalanced comparison operators breaking symmetry.

Example 4.1.4. When considering the model in Figure 4.2 and the safety property $\rho := \neg((l_2, l_2), true, true)$, it is obvious that the model is not symmetric. Starting with integer valuation $v^i_0(i) := 0$ for identifier aware integer variable i, the edges in both automata may be taken. When taking the edge in automaton A_1, the edge in automaton A_2 can be taken afterwards, resulting in a violation of the safety property. However, when first taking the edge in the second automaton, the value of i is set to

Figure 4.2: A Network of Timed Automata that is not symmetric due to the use of identifier aware integer variable i with an unbalanced comparison operator $\leq$

2 and the edge in the first automaton can not be taken. These two distinct sequences of edges do not conform to our intended notion of symmetry. The reason is the integer constraint using the asymmetric comparison operator $\leq$. Thus, we disallow such unbalanced operators for parameterized integer constraints.

We combine both distinct types of integer constraints as follows.

Definition 4.1.5. The set of *integer constraints* for the entire set of integer variables is defined as $\Psi(\mathcal{IV}, pid) = \{\psi_1 \wedge \psi_2(pid) | \psi_1 \in \Psi_{i\!\not{d}}(\mathcal{IV}), \psi_2(pid) \in \Psi_{id}(\mathcal{IV}, pid)\}$.

If an integer valuation v^i satisfies an integer constraint $\psi \in \Psi(\mathcal{IV}, pid)$ for an instantiated parameter pid, we write $v^i \models \psi$. The initial integer valuation v^i_0 maps each integer variable $iv \in \mathcal{IV}$ to its initial value $v^i_0(iv)$ with $v^i_0(iv) \in \mathbb{Z}$ for $iv \in \mathcal{IV}_{i\!\not{d}}$ and $v^i_0(iv) = 0$ for $iv \in \mathcal{IV}_{id}$. Allowing an initial value other than the neutral element zero for identifier aware integer variables would also destroy symmetry, as a contained edge may directly be enabled in one of the automata (for example with constraint $i = pid$), while not being enabled in the others.

We also have to adapt the definitions of assignments. As above, they are more restrictive in case of identifier aware integer variables.

Definition 4.1.6. $\Omega_{i\!\not{d}}(\mathcal{IV})$ is the set of *integer assignments* ω for identifier unaware integer variables defined by

$$true \mid iv := n \mid iv := iv + n \mid \omega_1; \omega_2$$

with $iv \in \mathcal{IV}_{i\!\not{d}}$, $n \in \mathbb{Z}$ and $\omega_1, \omega_2 \in \Omega_{i\!\not{d}}(\mathcal{IV})$. The latter definition creates a sequence of assignments, which are applied from left to right. The resulting integer valuation $v^i[\omega]$ for integer assignment $\omega = \omega_1; \omega_2$ is, thus, defined as $(v^i[\omega_1])[\omega_2]$. For the non-recursive integer assignments, the resulting integer valuation $v^i[\omega]$ is defined as:

$$\forall iv \in \mathcal{IV}_{i\!\not{d}} : v^i[\omega](iv) = \begin{cases} n & \text{if } \omega = iv := n, \\ v^i(iv) + n & \text{if } \omega = iv := iv + n, \\ v^i(iv) & \textit{else}. \end{cases}$$

In contrast, identifier aware integer variables are only allowed to be assigned the neutral value or a specific value given as parameter. It will later be instantiated to each automaton's identifier.

Definition 4.1.7. For a given identifier $pid \in \mathbb{N}_{\geq 1}$, let $\Omega_{id}(\mathcal{IV}, pid)$ be the set of *parameterized integer assignments* $\omega(pid)$ defined by

$$true \mid iv := n \mid \omega_1; \omega_2$$

with $iv \in \mathcal{IV}_{id}$, $n \in \{0, pid\}$ and $\omega_1, \omega_2 \in \Omega_{id}(\mathcal{IV}, pid)$. The latter definition creates a sequence of assignments, which are applied from left to right. The resulting integer valuation $v^i[\omega(pid)]$ for integer assignment $\omega(pid) = \omega_1; \omega_2$ is, thus, defined as $(v^i[\omega_1])[\omega_2]$. For the non-recursive integer assignments, the resulting integer valuation $v^i[\omega(pid)]$ is defined as:

$$\forall iv \in \mathcal{IV}_{id} : v^i[\omega(pid)](iv) = \begin{cases} n & \text{if } \omega(pid) = iv := n, \\ v^i(iv) & else. \end{cases}$$

Taking both adapted definitions into account, we combine the set of allowed integer assignments as follows.

Definition 4.1.8. With both kinds of integer variables we get the following set of *integer assignments* $\Omega(\mathcal{IV}, pid) = \{\omega_1; \omega_2(pid) | \omega_1 \in \Omega_{\not{id}}(\mathcal{IV})$ and $\omega_2(pid) \in \Omega_{id}(\mathcal{IV}, pid)\}$.

Note, that the decision where parameterized assignments $\omega_2(pid)$ are placed in the sequence (in front or behind the assignments ω_1 for identifier unaware variable) does not matter, as the sets of variables do not interfere with each other.

Using the above split of the set of integer variables, we can finally formalize the intended notion of symmetry.

4.2 Symmetry

Our notion of symmetry relies on the permutation of the timed automata in the model. Ip and Dill introduced this technique using permutations to define and employ symmetry in 1996 [ID96] for finite state systems. It has, however, also been applied for timed automata [Hen+04] resulting in a definition of symmetry close to ours. The utilization of the symmetry, however, is very distinct, as the above approach is concerned with symmetry reduction when verifying a single model. The resulting effect of a permutation on the states can be defined as an operation that swaps the locations and clock and integer values related to the swapped automata. We show below, how the values of two timed automata are swapped.

Definition 4.2.1 (Swap)**.** Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given with concrete semantics $TS = (S, s_0, \rightarrow)$. A *swap* $\pi = swap_{i,j}$ of two timed automata A_i and A_j ($i, j \in \{1, \ldots, n\}$) changes the state $s = ((l_1, ..., l_n), v^c, v^i) \in S$ into the state $\pi(s) = ((\pi(l_1), ..., \pi(l_n)), \pi(v^c), \pi(v^i)) \in S$ with

- $\forall k \in \{1, \ldots, n\} : \pi(l_k) = \begin{cases} l_i & if\ k = j, \\ l_j & if\ k = i, \\ l_k & else. \end{cases}$

- $\forall c \in C^g : \pi(v^c)(c) = v^c(c)$.
- $\forall k \in \{1, \ldots, n\} : \forall c \in C^l_k : \pi(v^c)(c) = \begin{cases} v^c(c_i) & if\ k = j, \\ v^c(c_j) & if\ k = i, \\ v^c(c) & else, \end{cases}$
  where c_i and c_j refer to the clock c in automaton A_i ($c \in C^l_i$) and A_j ($c \in C^l_j$), respectively.
- $\forall iv \in \mathcal{IV}_{\not{id}} : \pi(v^i)(iv) = v^i(iv)$.
- $\forall iv \in \mathcal{IV}_{id} : \pi(v^i)(iv) = \begin{cases} i & if\ v^i(iv) = j, \\ j & if\ v^i(iv) = i, \\ v^i(iv) & else. \end{cases}$

Note, that we also use $\pi(m)$ to denote the identifier of the automaton with which m has been swapped, formally $\pi(m) = \begin{cases} i & if\ m = j, \\ j & if\ m = i, \\ m & else. \end{cases}$

The *swap* operation swaps the locations of the two automata. Furthermore, it interchanges the values of their local clocks, as well as values for identifier aware integer variables that refer to one of the two automata. The swap operation is applicable, whenever the two swapped automata have the same set of locations and local clocks. However, its applicability does not ensure symmetry as defined below.

Note, that a permutation involving more than two automata being swapped can easily be realized via several swap operations in sequence. We employ this swap operation to define the notion of symmetry required in our workflow.

Definition 4.2.2 (Symmetry of the State Space)**.** Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given with concrete semantics $TS = (S, s_0, \rightarrow)$. It is *symmetric*, if for any swap π and any states $s_1, s_2 \in S$, it holds that $s_1 \rightarrow s_2$ with time delay δ and taken edge e of timed automaton A_i if and only if $\pi(s_1) \rightarrow \pi(s_2)$ with time delay δ and taken edge e of timed automaton $A_{\pi(i)}$. Furthermore, it must hold that $s = s_0$ if and only if $\pi(s) = s_0$.

Our definition of symmetry ensures the initial state to be symmetric, as well as paths to be symmetric. This implies for a state s that is reachable via a path from an initial state, that all its swapped states $\pi(s)$ are reachable via the respective swapped paths. Note, that the notion of symmetry characterizes π to be an automorphism on the transition system that is the concrete semantics TS (c.f. [Hen+04]).

As mentioned above, the applicability of the swap operation is not sufficient to ensure this notion. It can, however, be ensured via the specific restrictions on integer variables as above. To this end, we formalize our models as templates that are bound to these restrictions. We have proven in the Appendix B.1 that all networks of timed automata created via these templates meet our notion of symmetry.

4.3 Specification via Templates

Symmetric networks of timed automata allowed in our incremental approach are defined as instantiations of a template. A template is the abstract specification of a parameterized network of timed automata using the above redefinitions of integer variables. In order to actually create the network, it is instantiated several times with the distinct identifiers of the automata. The resulting timed automata in the network are, thus, distinct in dependence of their identifiers. In summary, templates are an intuitive way of formalizing symmetric models for our approach. We define this concept below. It is closely related to the templates as defined in Uppaal, but imposes the above restrictions on the automata.

Definition 4.3.1 (Timed Automaton Template)**.** Let the set C^g of global clocks be given, as well as the global set of integer variables $\mathcal{IV} = \mathcal{IV}_{i\not d} \cup \mathcal{IV}_{id}$ as the union of disjoint sets of identifier unaware ($\mathcal{IV}_{i\not d}$) and aware ($\mathcal{IV}_{id}$) integer variables, s.t. $\mathcal{IV}_{i\not d} \cap \mathcal{IV}_{id} = \emptyset$. A *symmetric timed automaton template* $\mathcal{A}(pid)$ defined over globally shared C^g and $\mathcal{IV}$ is a tuple $\mathcal{A}(pid) = (L, l_0, C, \mathcal{IV}, \emptyset, Inv^c, Inv^i(pid), E(pid))$ such that

- $pid \in \mathbb{N}_{\geq 1}$ is a unique parameter, called identifier,
- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- $C = C^l \cup C^g$ is the union of the finite and disjoint sets of local and global clocks with initial valuation v_0^c,
- $\mathcal{IV}$ is the finite set of shared integer variables with initial valuation v_0^i,
- $\emptyset$ is the set of synchronization channels (no synchronization is allowed),
- $Inv^c : L \to \Phi(C)$ is a total function of clock invariants, s.t. $v_0^c \models Inv^c(l_0)$,
- $Inv^i(pid) : L \to \Psi(\mathcal{IV}, pid)$ is a total function of integer invariants, s.t. $v_0^i \models Inv^i(pid)(l_0)$, and
- $E(pid) \subseteq L \times \{\epsilon\} \times \Phi(C) \times \Psi(\mathcal{IV}, pid) \times \Omega(\mathcal{IV}, pid) \times 2^C \times L$ is the set of edges.

Instantiating a template $A(pid)$ with unique identifier $pid = j \in \mathbb{N}_{\geq 1}$ results in timed automaton A_j. The following example illustrates the process of instantiation.

Example Figure 4.3 shows the template representing the *Fischer_U* model shown in the previous chapter (Figure 3.1) modeling the Fischer mutual exclusion algorithm. It can be used to create the symmetric model with any number of automata. There exists only one integer variable (*id*) that is aware of identifiers and the constraints and assignments involving it are parameterized with *pid*. The template can be instantiated, e.g., with the two identifiers 1 and 2 resulting in the automata depicted

in Figure 3.1. When instantiating the template with a unique identifier, e.g., $pid = 2$, a timed automaton A_2 is created in which the parameter *pid* is replaced by the actual identifier 2. For instance, the automaton contains the constraint $id = 2$ instead of the parameterized one $id = pid$.

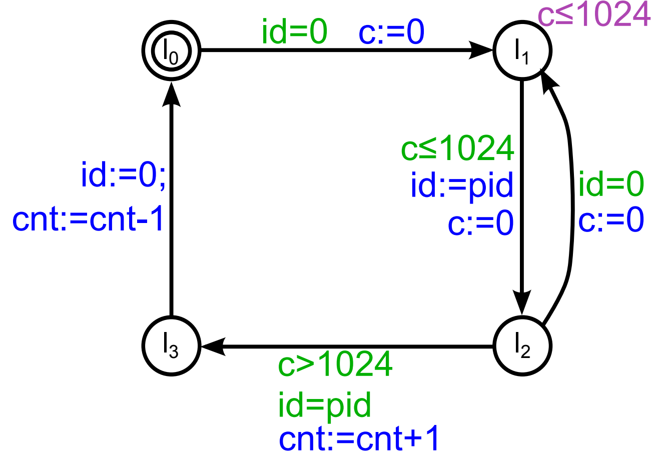


Figure 4.3: Example of a timed automaton template that represents the Fischer mutual exclusion algorithm (*Fischer_U*)

We employ templates as intuitive means to specify a parameterized timed system modeled as symmetric network of timed automata. In order to create a model for a fixed number $n \in \mathbb{N}_{\geq 1}$ of automata, the template is instantiated n timed with the respective identifiers 1 to n. We formalize this creation of a symmetric network of timed automata with fixed number of automata below.

Definition 4.3.2 (Symmetric Network of Timed Automata)**.** Given $n \in \mathbb{N}_{\geq 1}$ and the timed automaton template $\mathcal{A}(pid)$ as defined in Definition 4.3.1. The *symmetric network of timed automata* with n timed automata is defined as $NTA_n = \langle A_1, \dots, A_n \rangle$, where A_i is the instantiation of $\mathcal{A}(pid)$ with $pid = i$ ($i \in \{1, \dots, n\}$).

This definition of networks of timed automata via templates ensures the models to be symmetric. Section B.1 in the Appendix proves our notion of symmetry to hold for the models.

In addition to symmetric models, our technique relies on the safety property to be symmetric, too. It is defined as follows.

Definition 4.3.3 (Symmetry of the Safety Property)**.** Let a network of timed automata $NTA = \langle A_1, ..., A_n \rangle$ be given with concrete semantics $TS = (S, s_0, \rightarrow)$. Given any swap π and state $s \in S$, a *safety property* ρ *is symmetric* if it holds that $s \models \rho$ if and only if $\pi(s) \models \rho$.

We ensure this symmetry via redefinition of the safety property. The adapted definition includes all permutations of timed automata in the network and, thus, is inherently symmetric.

Recall the definition of error state specifications (Definition 2.1.13). An error state specification *err* reasons about the locations, integer and clock valuations of states in a network NTA_m of m timed automata. In order to employ the error state specification also for larger models of size $n \geq m$ (as in a parameterized system), we create a scaled error state specification err^n. To this end, the partial location vector simply needs to be filled with the unspecified location element ($*$) for all additional automata. We obtain a *symmetric error state specification* by considering all permutations of the n timed automata.

Definition 4.3.4 (Symmetric Error State Specification)**.** Given an error state specification $err = (\bar{l_m}, \phi, \psi)$ for NTA_m as in Definition 2.1.13, where $\psi = \psi_0 \wedge \psi_1(1) \wedge \cdots \wedge \psi_m(m)$ contains parameterized constraints for the m automata. The *scaled error state specification* for NTA_n ($n \geq m$) is defined as $err^n = (\bar{l}, \phi, \psi)$ with $\forall i \in \{1, \ldots, m\} : \bar{l}[i] = \bar{l_m}[i]$ and $\forall i \in \{m+1, \ldots, n\} : \bar{l}[i] = *$.

The *symmetric error state specification* for NTA_n is defined as $err^n_{sym} = \exists \pi : err^n_\pi$ for permutation π (cf. Def. 4.2.1) with $err^n_\pi = (\pi(\bar{l}), \pi(\phi), \pi(\psi))$ being defined as

- $\forall i \in \{1, \ldots, n\} : \pi(\bar{l})[i] = \bar{l}[\pi(i)]$,
- $\pi(\phi) = \begin{cases} \pi(\phi_1) \wedge \pi(\phi_2) & \text{if } \phi = \phi_1 \wedge \phi_2, \\ (\pi(x) - \pi(y)) \bowtie n & \text{if } \phi = (x - y) \bowtie n, \\ \pi(x) \bowtie n & \text{if } \phi = x \bowtie n, \\ true & \text{if } \phi = true, \end{cases}$
  where $\pi(x)$ refers to clock x itself if $x \in C^g$ or otherwise refers to the respective clock in $C^l_{\pi(i)}$ if $x \in C^l_i$,
- $\pi(\psi) = \psi_0 \wedge \psi_1(\pi(1)) \wedge \cdots \wedge \psi_m(\pi(m))$.

With the existential quantification, which is replaced by enumeration and disjunction in the SMT-encoding, any combination of timed automata is possible and, thus, the error state is symmetric. For illustration, consider the following example.

Example 4.3.5. Let the model *Fischer_U_3* be given ($n = 3$). Consider an error state specification $err = ((*, l_1), c_1 \leq 0, cnt \leq 1 \wedge id = 1)$ referring to $m = 2$ timed automata, namely A_1 and A_2, and c_1 denoting clock c in A_1. It specifies that any state is an error state that includes location l_1 for automaton A_2, a value for c of A_1 less or equal to 0, and $cnt \leq 1$ and $id = 1$. As can be seen, the timed automaton A_3 is not referred. The scaled version simply states that the location of A_3 is undefined, formally $err^3 = ((*, l_1, *), c_1 \leq 0, cnt \leq 1 \wedge id = 1)$. When considering the swap $\pi_1 = swap_{1,2}$ the resulting permuted error state specification is $err^3_{\pi_1} = ((l_1, *, *), c_2 \leq 0, cnt \leq 1 \wedge id = 2)$, where c_2 denotes clock c in A_2. Another swap might be $\pi_2 = swap_{1,3}$ resulting in the specification $err^3_{\pi_2} = ((*, l_1, *), c_3 \leq 0, cnt \leq 1 \wedge id = 3)$ with c_3 denoting clock c in A_3. In order to achieve symmetry, all permutations have to be considered.

With enumeration, the symmetric error state specification is as follows.

$$\begin{aligned}
err^3_{sym} =& ((*, l_1, *), c_1 \leq 0, cnt \leq 1 \wedge id = 1) \\
& \vee((*, *, l_1), c_1 \leq 0, cnt \leq 1 \wedge id = 1) \\
& \vee((l_1, *, *), c_2 \leq 0, cnt \leq 1 \wedge id = 2) \\
& \vee((*, *, l_1), c_2 \leq 0, cnt \leq 1 \wedge id = 2) \\
& \vee((*, l_1, *), c_3 \leq 0, cnt \leq 1 \wedge id = 3) \\
& \vee((l_1, *, *), c_3 \leq 0, cnt \leq 1 \wedge id = 3)
\end{aligned}$$

If any of them is satisfied by a state s, then s is an error state. Trivially, any swap $\pi(s)$ is also an error state, since all permutations of the error state specification are considered.

The above symmetric error state specifications allow for the definition of symmetric safety properties.

Definition 4.3.6 (Symmetric Safety Property)**.** Let NTA_n be a given network of n timed automata. The *symmetric safety property* is defined as $\rho^n = \neg(err1^n_{sym}) \wedge \neg(err2^n_{sym}) \wedge \ldots$ for symmetric error state specifications $err1^n_{sym}, err2^n_{sym}, \ldots$.

Given the above definitions, we define the actual verification questions we are interested in when considering parameterized timed systems.

> **Verification Question:** Given the timed automaton template $\mathcal{A}(pid)$ and a symmetric safety property ρ, does the safety property ρ^n hold for all networks of timed automata NTA_n consisting of $n \in \mathbb{N}_{\geq 1}$ instantiations of the template?

Since the error states in the safety property are defined for models with more than or exactly m timed automata, we say ρ^n holds for NTA_n with $n < m$, but need to verify the property for all $n \geq m$. In the following, we shortly elaborate on the decidability of this verification question.

4.4 Decidability

In general, the above verification question is undecidable. Trivially, the argumentation of the previous chapter (Subsection 3.4.1) can be applied for our symmetric models as well. One reason for undecidability is the existence of updates on the identifier unaware integer variables as has been shown via reduction in the mentioned subsection. An additional issue in the above verification question is the fixed, but arbitrary large number of timed automata. This allows for a reduction that exploits the number of automata as shown by Abdulla et al. [ADM04] for timed networks. Their reduction can be adapted to fit our formalism by representing the controller via an integer variable and using a global clock in combination with additional integer variables for synchronization of edges.

Thus, the verification question for parameterized timed systems is in general undecidable. However, Abdulla has shown that there exist some decidable instances, e.g., when a process includes at most one clock like in the Fischer models [AJ03]. Our experiments are in line with these insights. Hence, an approach answering the given verification question is reasonable for some models. We show our incremental technique for this problem in the following.

4.5 Basic Approach

We propose to incrementally verify the symmetric models with increasing number n of timed automata. To this end, we harness the symmetry and composition aspect resulting in a Termination Theorem that ensures the safety property to hold for all $n \in \mathbb{N}$ upon successful application.

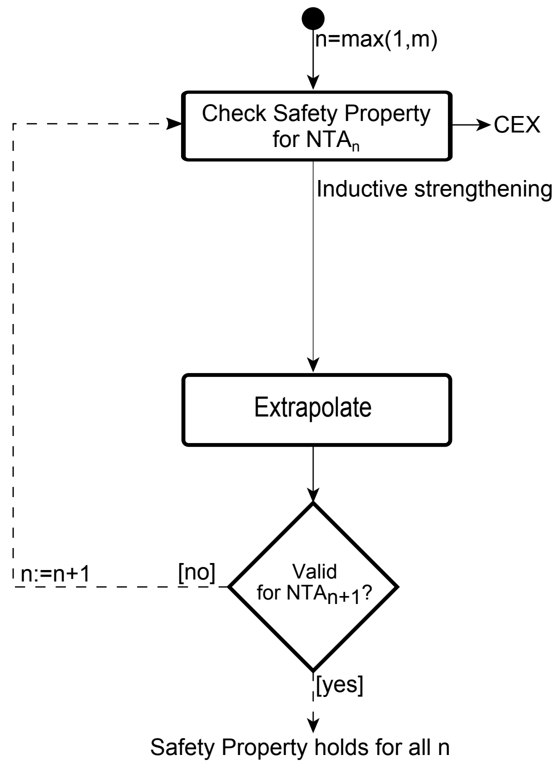


Figure 4.4: Basic workflow

The workflow can be seen in Figure 4.4. It comprises a loop that is indefinitely iterated until the Termination Theorem is applicable, i.e., our extrapolation produces a valid inductive strengthening for NTA_{n+1}. However, since the verification question is undecidable in general, there exist instances that run indefinitely with our theorem never being applicable. The approach is entirely based on SMT-solving and works as follows. Our workflow starts with the smallest model. Taking into account that the safety property is specified over m timed automata (cf. Definition 4.3.4), we start

with a network of timed automata NTA_n, where $n = max(1, m)$. The symmetric safety property ρ^n is verified for NTA_n using our IC3-based technique *IC3 with Zones* presented in the previous chapter providing an inductive strengthening of the safety property in case of success. Otherwise, a counterexample trace is found and the safety property does not hold for all $n \in \mathbb{N}_{\geq 1}$. The inductive strengthening is provided as an SMT-formula $\|F\|$ in conjunctive normal form. It is a conjunction of clauses, which consist of location, clock and integer literals reasoning about the n timed automata in the model NTA_n. In order to use it for the next larger model NTA_{n+1}, we execute a subsequent extrapolation step. Definition 4.5.1 specifies this extrapolation procedure with which we yield an SMT-formula $\|F\|^{exp(n+1)}$ reasoning about all timed automata in NTA_{n+1}.

Definition 4.5.1 (Extrapolation of an Inductive Strengthening)**.** Given the SMT-formula $\|F\|$ representing the inductive strengthening F of safety property ρ^n for the network of timed automata NTA_n. $\|F\|^{exp(x)}$ is the SMT-formula representing the *Extrapolation* of F to a network NTA_x of $x \geq n$ timed automata. $\|F\|^{exp(x)}$ is defined as the SMT-formula representing the conjunction of $\pi(F)$ for all possible permutations π of the x timed automata. To this end, $\|F\|$ is permuted according to the operation *swap*, i.e., location literals are swapped, as well as local clocks and values of identifier aware integer variables different from the neutral element 0.

The implementation of the extrapolation procedure permutes clauses separately. To this end, only clauses that are not identical up to permutation are considered. This allows an efficient handling of clauses and results in a low runtime. The following example illustrates the above extrapolation procedure.

Example 4.5.2. Consider the timed automaton template depicted in Figure 4.3. The instantiation for $n = 2$ results in the symmetric network of timed automata depicted in Figure 3.1. The verification of the symmetric safety property $\rho := \neg(cnt > 1)$ computes the inductive strengthening as depicted in Table 4.1 in the second column. There exist five clauses that are not identical up to permutation. The first column of Table 4.2 depicts representatives for these clauses. The application of the extrapolation procedure computes all permuted clauses of these representatives for all permutations of the $n + 1 = 3$ timed automata. The identifier aware integer variable *id* is represented in the encoding as int_0 and has to be considered in the permutations whenever the compared value in the literal is a specific identifier. The identifier unaware integer variable *cnt* is represented in the encoding as int_1 and must not be considered in the permutations at all. In addition, all location and local clock variables have to be considered for the permutations. The resulting clauses are depicted in the second column of the mentioned table. As can be seen, the extrapolated formula equals the inductive strengthening computed in the previous chapter for *Fischer_U_3*. It shows that it is possible to infer a valid inductive strengthening of the safety property for $n + 1$ automata from an inductive strengthening of the safety property for n automata. Our workflow relies on this potential.

representative clause from $\|F\|$	$\|F\|^{exp(3)}$
$(int_1 \leq 1)$	$(int_1 \leq 1)$
$\wedge((int_0 \neq 0) \vee (int_1 \leq 0))$	$\wedge((int_0 \neq 0) \vee (int_1 \leq 0))$
$\wedge(\neg l_0^1 \vee l_1^1 \vee (int_1 \leq 0))$	$\wedge(\neg l_0^1 \vee l_1^1 \vee (int_1 \leq 0))$
	$\wedge(\neg l_0^2 \vee l_1^2 \vee (int_1 \leq 0))$
	$\wedge(\neg l_0^3 \vee l_1^3 \vee (int_1 \leq 0))$
$\wedge(l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$	$\wedge(l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$
	$\wedge(l_0^2 \vee (int_0 \neq 2) \vee (int_1 \leq 0))$
	$\wedge(l_0^3 \vee (int_0 \neq 3) \vee (int_1 \leq 0))$
$\wedge((c_0^1 \leq 1024.0) \vee (int_0 \neq 1) \vee (c_0^2 > 1024.0))$	$\wedge((c_0^1 \leq 1024.0) \vee (int_0 \neq 1) \vee (c_0^2 > 1024.0))$
	$\wedge((c_0^1 \leq 1024.0) \vee (int_0 \neq 1) \vee (c_0^3 > 1024.0))$
	$\wedge((c_0^2 \leq 1024.0) \vee (int_0 \neq 2) \vee (c_0^1 > 1024.0))$
	$\wedge((c_0^2 \leq 1024.0) \vee (int_0 \neq 2) \vee (c_0^3 > 1024.0))$
	$\wedge((c_0^3 \leq 1024.0) \vee (int_0 \neq 3) \vee (c_0^2 > 1024.0))$
	$\wedge((c_0^3 \leq 1024.0) \vee (int_0 \neq 3) \vee (c_0^1 > 1024.0))$

Table 4.2: Each of the clauses (that are not identical up to permutation) in the computed inductive strengthening for n automata (left) is permuted for all permutations of the $n+1$ automata in order to create the extrapolated formula (right)

The resulting extrapolated formula $\|F\|^{exp(n+1)}$ is denoted as candidate inductive strengthening for NTA_{n+1}. It is then checked for validity, i.e., whether all three characteristics of inductive strengthenings hold true for the extrapolated formula $\|F\|^{exp(n+1)}$ and model NTA_{n+1}. If invalid, n is incremented and the next iteration of the loop is started. Otherwise, the candidate represents a valid inductive strengthening of ρ^{n+1} for NTA_{n+1}, in which case our Termination Theorem (Theorem 4.5.1) applies. It states that the safety property holds $\forall n \in \mathbb{N}_{\geq 1}$, if the candidate is a valid inductive strengthening.

Theorem 4.5.1 (Termination Theorem)**.** *Given the SMT-formula $\|F\|$ representing an inductive strengthening F of ρ^n for NTA_n. If $\|F\|^{exp(n+1)}$ is an inductive strengthening of ρ^{n+1} for NTA_{n+1}, then $\|F\|^{exp(n+2)}$ is an inductive strengthening of ρ^{n+2} for NTA_{n+2}.*

The proof of correctness is given in the subsequent chapter for a more general formalism.

The incremental design of the workflow yields the benefit of starting with small networks of timed automata, which are efficiently verifiable. Thus, a fast overall performance of our approach is likely in case the Termination Theorem can be applied early. Additionally, our theorem is most suitable for an incremental workflow, being able to reason about all $n \in \mathbb{N}_{\geq 1}$ by taking into account the results of a specific instance.

We illustrate the workflow in the following.

Example 4.5.3. As an example consider the *Fischer_U* template as given in Figure 4.3 with the symmetric safety property $\rho := \neg(cnt > 1)$ specifying values for $x = 0$ timed automata. Our workflow starts with the verification of the safety property for the symmetric network of $n = 1$ timed automata NTA_n. After successful verification, the found inductive strengthening $\|F\| = (int_1 \leq 1) \wedge (l_0^1 \vee (int_1 \leq 0)) \wedge (l_1^1 \vee (int_1 \leq 0))$ is returned as can be seen in Table 4.1. The extrapolation procedure produces a candidate formula with five clauses, namely $\|F\|^{exp(n+1)} =$

$(int_1 \leq 1) \wedge (l_0^1 \vee (int_1 \leq 0)) \wedge (l_0^2 \vee (int_1 \leq 0)) \wedge (l_1^1 \vee (int_1 \leq 0)) \wedge (l_1^2 \vee (int_1 \leq 0))$. The candidate is not inductive for NTA_{n+1}, as *Consecution* fails. Thus, the next cycle starts with the verification of the safety property ρ^n for NTA_n with $n = 2$. After successful verification the resulting inductive strengthening is again extrapolated into a candidate, which is checked for validity (which means whether Theorem 4.5.1 is applicable). This cycle is iterated indefinitely until the theorem can be applied, a counterexample trace is found during a run of *IC3 with Zones* or the systems runs out of memory or time.

4.6 Optimizations

Technically, the presented workflow can be realized using any verification technique resulting in an inductive strengthening as defined above. However, we designed some optimizations tailored specifically to our algorithm *IC3 with Zones* shown in Chapter 3. We present these optimizations in the following, namely a *Feedback*-loop and the *Generalization* of the found inductive strengthening. These enhancements have distinct aims within the workflow. The latter is designed to increase the applicability of Theorem 4.5.1, while the former is constructed to speed up the verification of models in the IC3 algorithm itself. We start with a detailed description of this speed-up optimization.

4.6.1 Feedback Loop

During the basic loop of our workflow, we propose a candidate inductive strengthening $\|F\|^{exp(n+1)}$ for the next larger model NTA_{n+1}. In case the Termination Theorem can not be applied, i.e., the candidate is not a valid inductive strengthening, it is discarded so far. It might, however, be of substantial value. Even though it is not a valid inductive strengthening it might be similar to one, as some of its clauses might occur within a valid inductive strengthening. These clauses should be extracted and used to accelerate the verification process. To this end, the proposed candidate is injected into the verification run of *IC3 with Zones* for NTA_{n+1}. We call this a *Feedback*-loop, since the clauses of the candidate are fed back into the cycle instead of simply being discarded. In order to decide, which of the clauses are relevant, i.e., can be reused without breaking the properties of the frames in the IC3 algorithm, we use two distinct methods. On the one hand, we employ the incremental method of Chockler et al. [Cho+11] to inject an inductive subset of the clauses in the candidate formula into each frame of IC3. On the other hand, we employ our own method to inject clauses that are inductive relative to the set of initial states (frame F_0) into the frame F_1. We denote the former one as *Chockler*-Feedback and the latter one as *Frame*1-Feedback. Both methods are shown below.

***Chockler*-Feedback** The following method is taken from Chockler et al. [Cho+11], where it is used for incremental verification of hardware models. It searches for the largest subset of the clauses in the candidate formula $\|F\|^{exp(n+1)}$ that is inductive.

This set of clauses can, thus, be conjoined to all frames in *IC3 with Zones*. As a result, the search space for CTIs in the frames is pruned heavily upfront. Formally, it searches the largest subset $C \subseteq clauses(\|F\|^{exp(n+1)})$ of clauses in the candidate that is inductive, i.e.,

- **Initiation:** $\|I^{n+1}\| \wedge \neg\|C\|$ is unsatisfiable,
- **Consecution:** $\|C\| \wedge \|T^{n+1}\| \wedge \neg\|C\|'$ is unsatisfiable,

with the formulae $\|I^{n+1}\|$ and $\|T^{n+1}\|$ denoting the encoding of the set of initial states and the transition relation of NTA_{n+1} with invariants as shown, e.g., in Subsection 3.1.10.

The conjunction of each of the frames with the set C of clauses clearly preserves the four properties of the frames, as C is inductive. It is extremely powerful in pruning the considered states in the frames. However, we observed that the inductive set of clauses C is rather small for some models, since many of the clauses hinder inductiveness. Despite being of value for the IC3 algorithm, these clauses are discarded. For this reason, we constructed a weaker Feedback method detailed below.

*Frame*1**-Feedback** Instead of searching an inductive subset of clauses that are conjoined to every frame, our *Frame*1-Feedback searches a subset of clauses that is inductive relative to the set of initial states, but can only be conjoined to frame F_1. Formally, given the proposed candidate formula $\|F\|^{exp(n+1)}$, we compute the largest subset $C \subseteq clauses(\|F\|^{exp(n+1)})$ of clauses that are initialized and inductive relative to $F_0 = I^{n+1}$:

- **Initiation:** $\|I^{n+1}\| \wedge \neg\|C\|$ is unsatisfiable,
- **Consecution relative to I^{n+1}:** $\|I^{n+1}\| \wedge \|T^{n+1}\| \wedge \neg\|C\|'$ is unsatisfiable,

with the formulae $\|I^{n+1}\|$ and $\|T^{n+1}\|$ denoting the encoding of the set of initial states and the transition relation of NTA_{n+1} with invariants as shown, e.g., in Subsection 3.1.10.

The result is injected into *IC3 with Zones* as clauses of frame F_1 (cf. Section 2.4), which may prevent the costly rediscovery of some of them. Although the conjunction with F_1 does not prune the larger frames $F_2, \ldots$ the method is of importance. Clearly, the number of clauses in C is not smaller in the *Frame*1-Feedback than in the *Chockler*-Feedback, as an inductive set of clauses is also inductive relative to F_0. Furthermore, the clauses of the latter method can be propagated to larger frames during the run of *IC3 with Zones*. Thus, the restriction of the *Frame*1-Feedback to conjoin clauses only with F_1 is not a serious cutback.

Both these feedback methods can be used separately or in combination. Furthermore, their implementation is simple as the clauses only have to be checked and inserted at the start of *IC3 with Zones* and the rest of the algorithm proceeds entirely as before. Our implementation is based on the one described by Chockler et al. [Cho+11] using auxiliary variables for finding the injected subsets.

Apart from this Feedback technique used to speed up the verification runs of IC3, we use an additional optimization in our workflow. It is devised to improve the applicability of the Termination Theorem.

4.6.2 Generalization

The inductive strengthening $\|F\|$ of the safety property ρ^n computed during the verification run of $IC3$ for NTA_n may not be the only possible inductive strengthening. With the intuition that an inductive strengthening with fewer clauses may more likely be extrapolated into a candidate $\|F\|^{exp(n+1)}$ that is a valid inductive strengthening (since there are less clauses that could hinder inductiveness, especially *Consecution*), we try to alter $\|F\|$. To this end, we generalize it (analog to the *Generalization* process in $IC3$ itself) by deleting clauses while maintaining a valid inductive strengthening for NTA_n. This is done making heavy use of the SMT-solver, including the utilization of Unsat-Cores as is done in $IC3$ itself. Since the subsequent extrapolation procedure will result in a symmetric formula, meaning all permutations of each clause are included, the *Generalization* procedure tries to delete all permutations of a clause at once, not one after another. The order in which clauses (or rather sets of clauses identical up to permutation) are deleted is controlled by the Unsat-Core as well as an ordering of clauses, favoring the ones for deletion that refer to many timed automata over those ones that refer only a few. The underlying intuition is to get rid of very specific clauses (referring to many automata) first.

The *Generalization* step results in an inductive strengthening with potentially fewer clauses, which might be preferable for the application of Theorem 4.5.1. We illustrate the approach in the following.

Example 4.6.1. Consider a verification run of the symmetric safety property $\rho = \neg(cnt > 1)$ for *Fischer_U_2*. It results in the inductive strengthening depicted in Table 4.3. The table also shows the generalized inductive strengthening, which clearly includes fewer clauses.

4.6.3 Combination

Using *Generalization* and a *Feedback*-loop in combination allows for an additional choice, if Theorem 4.5.1 could not be applied.

The next iteration of the loop is started with the injection of clauses into $IC3$. However, either the generalized inductive strengthening or the non-generalized (entire) inductive strengthening can be employed for this task. We will explore the effect of both, as well as the other proposed optimizations in the following section.

Figure 4.5 shows the overall workflow including all proposed optimizations.

clauses of $\|F\|$ found by IC3	*Generalization* of $\|F\|$
$(int_1 \leq 1)$	$(int_1 \leq 1)$
$\wedge(l_0^1 \vee l_0^2 \vee (int_1 \leq 0))$	
$\wedge(l_0^1 \vee l_1^2 \vee (int_1 \leq 0))$	
$\wedge(l_1^1 \vee l_0^2 \vee (int_1 \leq 0))$	
$\wedge(l_1^1 \vee l_1^2 \vee (int_1 \leq 0))$	
$\wedge((int_0 \neq 0) \vee (int_1 \leq 0))$	$\wedge((int_0 \neq 0) \vee (int_1 \leq 0))$
$\wedge(l_1^1 \vee \neg l_0^1 \vee (int_1 \leq 0))$	$\wedge(l_1^1 \vee \neg l_0^1 \vee (int_1 \leq 0))$
$\wedge(l_1^2 \vee \neg l_0^2 \vee (int_1 \leq 0))$	$\wedge(l_1^2 \vee \neg l_0^2 \vee (int_1 \leq 0))$
$\wedge(l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$	$\wedge(l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$
$\wedge(l_0^2 \vee (int_0 \neq 2) \vee (int_1 \leq 0))$	$\wedge(l_0^2 \vee (int_0 \neq 2) \vee (int_1 \leq 0))$
$\wedge(l_0^2 \vee (int_0 \neq 0) \vee (int_1 \leq 0))$	
$\wedge(l_0^1 \vee (int_0 \neq 0) \vee (int_1 \leq 0))$	
$\wedge((c_0^1 > 1024.0) \vee (c_0^2 \leq 1024.0) \vee (int_0 \neq 2))$	$\wedge((c_0^1 > 1024.0) \vee (c_0^2 \leq 1024.0) \vee (int_0 \neq 2))$
$\wedge((c_0^2 > 1024.0) \vee (c_0^1 \leq 1024.0) \vee (int_0 \neq 1))$	$\wedge((c_0^2 > 1024.0) \vee (c_0^1 \leq 1024.0) \vee (int_0 \neq 1))$

Table 4.3: Generalization: Each of the clauses (that are not identical up to permutation) in the computed inductive strengthening $\|F\|$ for NTA_n (left) is tried for deletion resulting in a *Generalization* of $\|F\|$ (right)

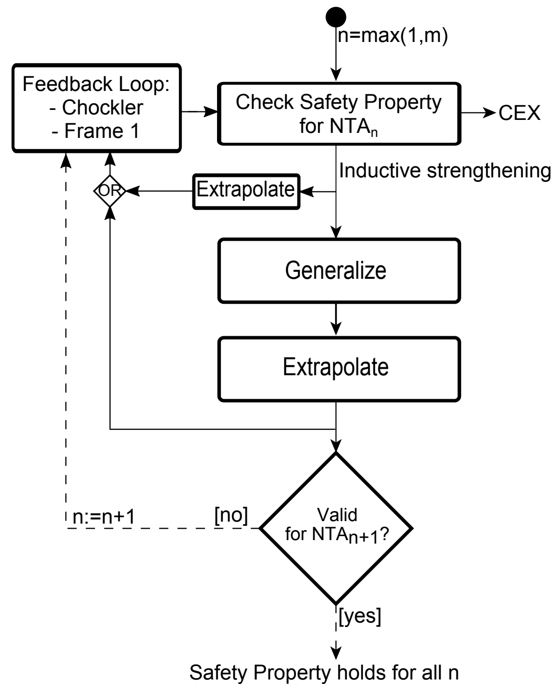
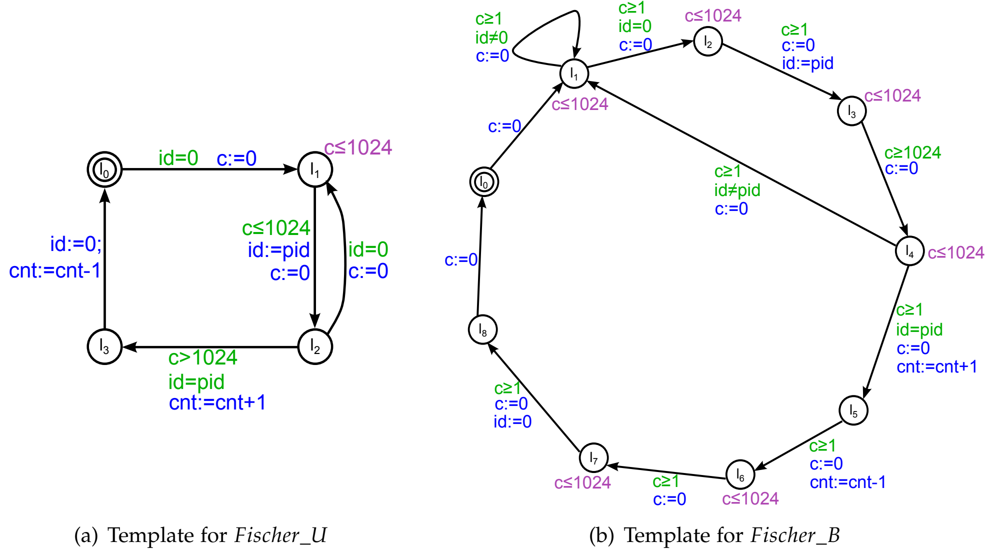


Figure 4.5: Workflow including the optimizations *Feedback* and *Generalization*

4.7 Evaluation

Several of the benchmark models presented for the experiments in Chapter 3 are symmetric and can be modeled as a template. They do not require synchronized edges and their structure, including the integer variables, satisfies the restrictions presented above. We employ these models in experiments to evaluate the practicality and performance of our approach. In particular, we employ both Fischer models (Figure 4.6), both Lamport models (Figure 4.7) and also the two Shavit-Lynch models (Figure 4.8).



(a) Template for *Fischer_U*　　(b) Template for *Fischer_B*

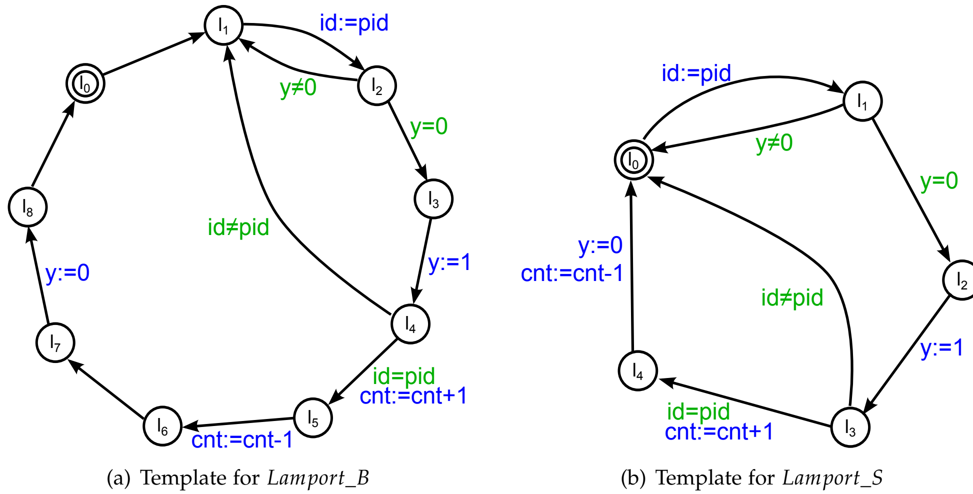
Figure 4.6: Templates for symmetric networks of timed automata modeling the Fischer algorithm (cf. models *Fischer_U* and *Fischer_B* in Figures 3.1 and 3.7)

Starting from the basic workflow, we examine the effects of the proposed optimizations and the overall practicality of our concept. We show promising results and explain the benefits in relation to non-symmetric verification approaches.

The presented workflows have been implemented in Java based on the implementation used in the previous chapter. As before, we ran our experiments on a pc with 3,2 GHz (AMD Phenom II X4 955).

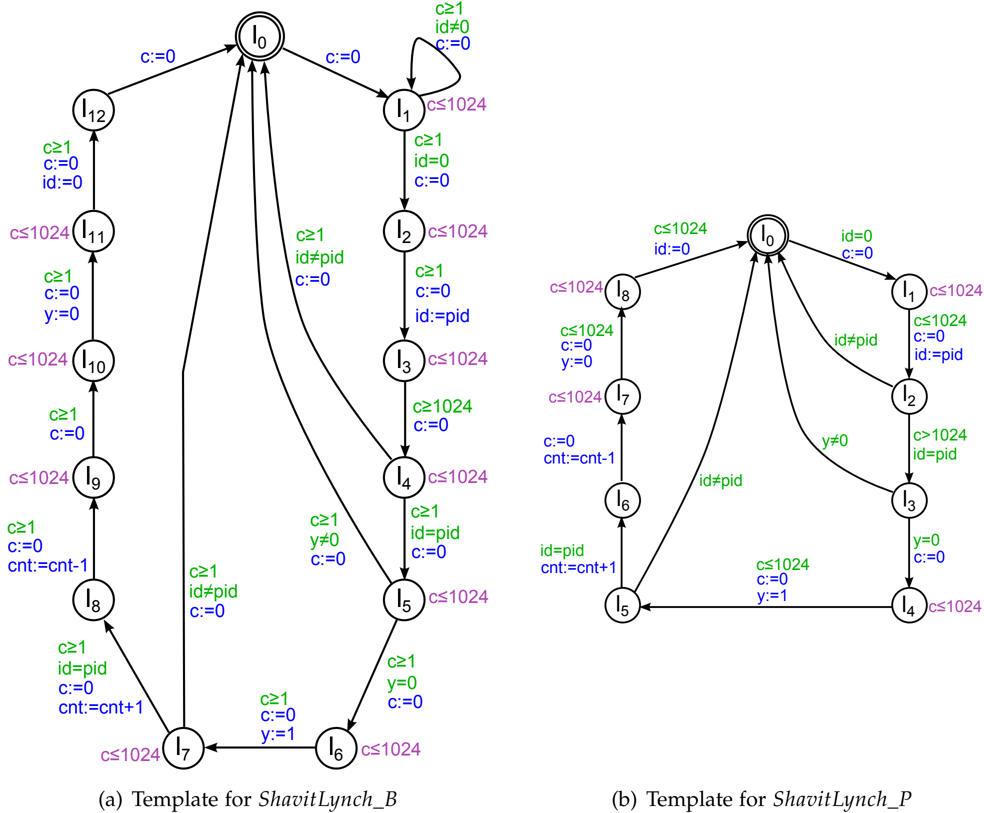
4.7.1 Experiments

In order to evaluate the basic idea of inferring inductive strengthenings for larger models from the smaller ones, we examine the performance of our basic workflow. It illustrates the applicability of our Termination Theorem and the practicality of the pure approach. Table 4.4 shows the runtimes of the basic workflow with a timeout of 5 hours (18000 seconds). The system ran out of memory when using more than 2 GB of memory.



(a) Template for *Lamport_B* (b) Template for *Lamport_S*

Figure 4.7: Templates for symmetric networks of timed automata modeling the Lamport algorithm (cf. models *Lamport_B* and *Lamport_S* in Figures 3.8 and 3.10)



(a) Template for *ShavitLynch_B* (b) Template for *ShavitLynch_P*

Figure 4.8: Templates for symmetric networks of timed automata modeling the Shavit-Lynch algorithm (cf. models *ShavitLynch_B* and *ShavitLynch_P* in Figures 3.9 and 3.11)

	Fischer_U	*Fischer_U* (switched)	*Fischer_B*	*Lamport_B*	*Lamport_S*	*ShavitLynch_B*	*ShavitLynch_P*
Runtime (s)	OOM	35,2	OOM	OOT	267,1	OOM	OOM
Successfully verified models NTA_n for which n?	1-8	$\forall n \in \mathbb{N}$	1-4	1-4	$\forall n \in \mathbb{N}$	1-3	1-4

Table 4.4: Runtime needed for the verification of a symmetric safety property for the parameterized timed systems using the basic workflow without optimizations: The safety property was verified for the entire parameterized system (marked as $\forall n \in \mathbb{N}$ in the lowermost row) or for only a few instances with fixed n, where we give the sizes of the models that were successfully verified

As can be seen, our basic workflow was not able to verify most of the parameterized timed systems. For five out of seven instances, the Termination Theorem could not be applied before running out of time or memory. However, the presented technique successfully verifies the safety property for two models for all $n \in \mathbb{N}$, e.g., the shrunk Lamport model. In addition, it verifies some single instances with fixed $n \in \mathbb{N}$ for the other parameterized systems before running out of memory or time, e.g., the Shavit Lynch model *ShavitLynch_B* has been verified for instances with 1,2 and 3 timed automata in the network.

We conclude that the general concept is very promising with huge potential as shown by the successful runs. The useful intermediate results, namely the verification of the safety property for models with fixed $n \in \mathbb{N}$, demonstrate that it is of use even in case of an incomplete run. However, the results also show that the overall percentage of models verified for all $n \in \mathbb{N}$ is not satisfying.

Hence, we examine the effect of the proposed optimizations. Individual experiments with these optimizations including all their combinations result in thorough data for estimating their benefits. We start by studying the effect of the *Generalization* step shown in Table 4.5.

	Fischer_U	*Fischer_U* (switched)	*Fischer_B*	*Lamport_B*	*Lamport_S*	*ShavitLynch_B*	*ShavitLynch_P*
Runtime (s)	1,3	2,5	50,2	OOT	1,3	OOM	OOT
Successfully verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-4	$\forall n \in \mathbb{N}$	1-3	1-4

Table 4.5: Runtime needed for the verification of a symmetric safety property for the parameterized timed systems using the workflow with *Generalization* optimization only: The safety property was verified for the entire parameterized system (marked as $\forall n \in \mathbb{N}$ in the lowermost row) or for only a few instances with fixed n, where we give the sizes of the models that were successfully verified

A comparison of the results with those obtained without optimization clearly shows the benefit of the optimization. The number of models that could be verified for all $n \in \mathbb{N}$ grew from two to four instances. As can be seen, this optimization does only affect the applicability of Theorem 4.5.1, as it has no influence on the IC3 runs (lowermost row if theorem not applicable).

Yet, some of the instances could still not be verified for all $n \in \mathbb{N}$. Thus, we examine the effect of the other proposed optimization, namely the *Feedback*-technique. The results are shown in Table 4.6, showing the performance of the workflow using the *Feedback*-loop without any other optimization.

	Fischer_U	*Fischer_U* (switched)	*Fischer_B*	*Lamport_B*	*Lamport_S*	*ShavitLynch_B*	*ShavitLynch_P*
Chockler - Runtime (s)	14,0	OOM	OOM	OOM	504,0	OOM	1264,2
Chockler - Verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	1-9	1-6	1-7	$\forall n \in \mathbb{N}$	1-3	$\forall n \in \mathbb{N}$
Frame1 - Runtime (s)	1,6	1,2	87,8	8,5	1,7	28,1	2778,5
Frame1 - Verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$
Both - Runtime (s)	OOM	OOM	243,1	51,9	1,9	329,7	OOM
Both - Verified models NTA_n for which n?	1-9	1-9	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-4

Table 4.6: Runtime needed for the verification of a symmetric safety property for the parameterized timed systems using the workflow with the *Feedback*-optimization: The safety property was verified for the entire parameterized system (marked as $\forall n \in \mathbb{N}$ in the lower rows) or for only a few instances with fixed n, where we give the sizes of the models that were successfully verified

Obviously, the *Feedback*-optimization works well. In fact, it shows the best overall performance in comparison to the previously tested settings (Tables 4.4 and 4.5). The *Feedback*-mechanism should, thus, be preferred to the *Generalization* optimization. Depending on the actual *Feedback*-technique, the performance of the verifications is very distinct.

The only option capable of verifying all instances for all $n \in \mathbb{N}$ is the *Frame*1-Feedback. Apart from the last instance (*ShavitLynch_P*), all instances have been verified with reasonable runtime. Noticeably, the combination of both distinct options (*Frame*1 and *Chockler*) is unable to outperform the single options. This fact can be seen when comparing the respective rows in Table 4.6.

Note, that the optimization speeds up the verification runs. This fact can be seen, when comparing the runs for *Lamport_B*. Without the optimization, the approach ran out of time after verifying models with size 1 to 4. In contrast, the runs with optimization are faster, e.g., when using the *Chockler*-Feedback, it does not run out of time. Instead, it runs out of memory first, after verifying models of size 1 to 7.

The injection of clauses additionally alters each run of *IC3 with Zones*, s.t. the

resulting inductive strengthening might be different, which in turn affects the subsequent cycles of our approach. Thus, the *Feedback*-optimization has a significant impact on our approach.

In the following, we evaluate whether the combination of both optimizations further improves the performance. As stated before, this combination allows for two options. On the one hand, the extrapolated generalized set of clauses can be used in the *Feedback*-loop. On the other hand, the extrapolated non-generalized set of clauses can be used.

Table 4.7 shows the performance using the two options.

	Fischer_U	*Fischer_U* (switched)	*Fischer_B*	*Lamport_B*	*Lamport_S*	*ShavitLynch_B*	*ShavitLynch_P*
Feedback using non-generalized set of clauses (all clauses)							
Chockler - Runtime (s)	1,3	2,0	55,1	OOM	1,3	OOT	OOM
Chockler - Verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-7	$\forall n \in \mathbb{N}$	1-3	1-6
Frame1 - Runtime (s)	1,2	1,1	34,9	9,0	1,7	199,7	OOT
Frame1 - Verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-4
Both - Runtime (s)	1,3	1,2	36,0	55,7	1,6	958,1	OOT
Both - Verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-4
Feedback using generalized set of clauses							
Chockler - Runtime (s)	1,3	2,0	50,5	OOM	1,4	OOM	880,3
Chockler - Verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-8	$\forall n \in \mathbb{N}$	1-3	$\forall n \in \mathbb{N}$
Frame1 - Runtime (s)	1,5	1,1	35,4	OOM	1,5	6622,6	OOT
Frame1 - Verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-8	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-4
Both - Runtime (s)	1,3	1,1	35,7,1	OOM	1,6	6981,5	OOT
Both - Verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-8	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	1-4

Table 4.7: Runtime needed for the verification of a symmetric safety property for the parameterized timed systems using the workflow with both optimizations. The safety property was verified for the entire parameterized system (marked as $\forall n \in \mathbb{N}$ in the second row) or for only a few instances with fixed n, where we give the sizes of the models that were successfully verified

Clearly, the experiments show that the combination of optimizations does not improve the overall performance. The basic workflow optimized only by the *Frame*1-Feedback loop still shows the best results.

In summary of the above experiments, we conclude that our workflow is of value, even if the verification run is incomplete. It verifies smaller models with fixed $n \in \mathbb{N}$ before running out of time or memory and is, thus, not entirely useless in such a

	Fischer_U	*Fischer_U* (switched)	*Fischer_B*	*Lamport_B*	*Lamport_S*	*ShavitLynch_B*	*ShavitLynch_P*
Runtime (s)	1,6	1,2	87,8	8,5	1,7	28,1	2778,5
Verified models NTA_n for which n?	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$	$\forall n \in \mathbb{N}$
Uppaal - Runtime (s)	0,7	0,7	17,8	5,3	0,4	0,3	1218,5
Uppaal - Verified models NTA_n for which n?	8	8	5	6	6	4	13
IC3 - Runtime (s)	0,9	0,8	6,7	5,3	1,1	17,3	149,5
IC3 - Verified models NTA_n for which n?	2	2	2	2	2	2	3

Table 4.8: Comparison of our workflow (first rows) with verifications for instances with fixed n that could be verified in the same time

case. Furthermore, our proposed *Feedback*-optimization accelerates the runs of our *IC3 with Zones* algorithm, which renders larger models verifiable that could not be verified without the injected clauses. The overall performance of the workflow using the optimizations is very promising and the technique, in general, has the benefit of verifying parameterized timed systems for all $n \in \mathbb{N}$. In order to illustrate this benefit, we show a comparison with verifications of models with fixed $n \in \mathbb{N}$ below.

4.7.2 Comparison with fixed Models

We demonstrate this benefit by comparison with the tool Uppaal and a single verification run of our algorithm *IC3 with Zones* of the previous chapter. As these tools can only verify models with a fixed $n \in \mathbb{N}$, we show the largest model with fixed n that these tools were able to verify in the same time that the above presented technique requires for the verification of the entire parameterized system for all $n \in \mathbb{N}$. Table 4.8 shows the runtimes and sizes of the verified models.

The presented incremental workflow (using the optimization *Frame*1-Feedback) clearly improves the verification of single models with fixed size. Due to its ability to extrapolate inductive strengthenings for larger models, it is able to reason about all $n \in \mathbb{N}$ by only verifying small models. The reuse of previous computation results by injecting clauses of the candidate in the next verification run additionally increases the efficiency. Thus, tools capable of only verifying models with fixed $n \in \mathbb{N}$ are not able to compete with the presented approach.

In the following, we illustrate the importance of inductive strengthenings for validation and reuse. If known, an inductive strengthening can easily be validated as shown in Subsection 3.5.5. Our approach results in an inductive strengthening that can be extrapolated for any $n \in \mathbb{N}$. We show the validation time for Uppaal's Fischer model (*Fischer_U*). Note, that Uppaal was only able to verify models up to $n = 13$ and our approach of the previous chapter has verified models up to

Fischer_U	n=10	n=20	n=30	n=40	n=50	n=100
Time (s)	0,8	1,5	3,7	6,8	12,7	135,3

Table 4.9: Validation times for the inductive strengthenings computed and extrapolated by the presented approach

$n = 50$. Table 4.9 shows the time needed to validate the extrapolated inductive strengthenings for models with fixed n.

Clearly, the approach presented in this chapter performs well. Due to the presented Termination Theorem, such a validation is not necessary. Our incremental workflow provides for a verification for all $n \in \mathbb{N}$ upon success.

4.8 Related Work

The related work for IC3 and verification of safety properties for timed automata has been presented in Chapter 2. The following related work is, thus, only concerned with the verification of safety properties for parameterized (timed) systems and symmetry.

Parameterized Systems There exist numerous publications concerning the untimed case. It has been shown that this verification question is, in general, undecidable [AK86]. Yet, several works deal with this verification question, resulting in semi-algorithms or working with restricted families of models. The works for parameterized systems might be categorized as follows.

Network Invariants: Many of these approaches require human interaction, e.g., the proposition of an invariant or closure. Wolper et al. [WL90] require the manual specification of a *network invariant*. The invariants are used to check whether the property holds true for the entire set of models. The manual specification, however, is a drawback. A similar finding has been made by Kurshan and McMillan [KM89], where the invariant is denoted as *process invariant*. It is used for verification of the entire set of models, but needs to be specified manually. Several approaches try to overcome the problem of manual proposition of an invariant. They synthesize invariants automatically, e.g., based on on network grammars and abstraction (Clarke et al. [CGJ95]) or based on a fixpoint computation using heuristics (Lesens et al. [LHR97]).

Regular Model Checking: Other approaches are based on reachability analysis with states represented as words and finite state transducers in between. These techniques are denoted as *regular model checking* and differ by the employed models, transducers and widening operators. Bouajjani et al. were amongst the first to publish about this technique [Bou+00]. Another representative is Kesten et al. [Kes+97] using symbolic model checking. Abdulla et al. extended the techniques in various directions. They examined the effect of having global conditions [Abd+99], and introduced incremental updates to the

transducers [Abd+02b]. Further work concerns other improvements [Abd+03] and specific structures [Abd+02a], [BT02].

Symmetry Reduction: In addition to the above techniques some researchers reduce the verification in parameterized systems to the one in a quotient structure, which only includes representatives for symmetric states. Two important works in this field are Emerson et al. [ES96] and Clarke et al. [Cla+96]. Later, Emerson et al. [ES97] and Gyuris et al. [GS97] introduced fairness and on-the fly model checking to the approach. The usage of symmetry in quotient structures was improved by an explicit modeling of symmetric variables, denoted scalarsets, by Ip and Dill [ID96]. This explicit modeling is close to the one employed in our work, where we explicitly define integer variables $\mathcal{IV}_{id}$ that are affected by symmetry. However, the technique closest to ours is the following.

Invisible Invariants: Pnueli et al. [PRZ01], [Aro+01] introduced an approach that automatically computes strengthenings of properties that may be used to prove the property for all models. To this end, they employ a project-abstract-procedure, where the set of reachable states is computed, which is projected to a smaller model (some references to variables are deleted). Afterwards, the result is abstracted to again include all processes of the model (similar to our extrapolation procedure). The outcome is checked for inductiveness. Additionally, the authors introduce a *Small Model Theorem*, which states that an inductive strengthening holds for all models, if it holds in the models up to a specific size depending on the structure of the models. This theorem defines a cutoff up to which the models have to be considered only. It is close in spirit to our Termination Theorem, which also defines sort of a cutoff (dynamically checked every iteration). Our theorem, however, is suitable for the incremental workflow with changing candidate formulae every iteration.

Apart from the above method, there exist other works that utilize a cutoff for verification. They require specific families of models (e.g., rings [EN95]) to compute a cutoff [EK00] up to which the model needs to be verified in order to guarantee the property to hold for all models.

Parameterized Timed Systems In addition to the above related work aiming for untimed models, there exist various, but fewer, publications aiming for parameterized timed systems. One of the first to actually use this terminology was Mahata [Mah05]. Apart from work with Abdulla [Abd+04] in which they use timed petri nets for the modeling of parameterized timed systems, they also reason about decidability for parameterized timed systems in general. Abdulla et al. showed that reachability analysis for timed networks with only a single clock per process is decidable [AJ03]. However, when increasing the number of clocks the reachability question becomes undecidable [ADM04].

Various other works exist that consider parameterized timed systems. Bruttomesso et al. [Bru+12] and Carioni et al. [CGR10] employ SMT-encodings in a decidable fragment in the theory of array for verification. Astefanoaei et al. [Ast+15]

combine locally computed invariants in order to yield invariants for all models. It builds upon the *Small Model Theorem* of Pnueli transferred to the domain of hybrid automata by Johnson et al. [JM12]. For their own approach, Johnson et al. employ Pnueli's Project-Abstract-Procedure in a fixpoint computation that searches an inductive strengthening or a counterexample for a fixed number of processes. With its usage of the small model theorem, this work is closest to our approach. As explained above, our Termination Theorem is similar in that is computes a cutoff up to which size the models have to be considered. The main difference, however, is the incremental nature of our theorem, as its applicability to the current candidate formula is checked every cycle.

4.9 Summary

The technique presented in this chapter is capable of verifying safety properties for parameterized timed systems given as networks of timed automata. To this end, we employ an incremental workflow that verifies the safety property for models of a fixed size, starting with the smallest one possible. Using the inductive strengthening computed during this verification, we try to reason about the entire parameterized system. If this step fails, the next cycle of the workflow is started with the next larger model. Otherwise, the reasoning in form of our Termination Theorem could be applied and we successfully verified the safety property for the entire parameterized timed system, i.e., for all its models with any number of timed automata.

The ability to reason about an entire parameterized system is of high value, as many mutual exclusion algorithms and peer to peer protocols can be modeled as such a system. The benefit is that properties for these algorithms or protocols can be checked for any number of processes executing them. It is a vital aspect when dealing with adaptive systems, as is the case in the context of Industry 4.0. Using our technique allows the verification of the entire parameterized timed system. As an example for the application context, consider its usage during the design phase, which allows the safe reconfiguration of the system during lifetime without the need for further verification. The addition or removal of one of the processes that execute the mutual exclusion algorithm or protocol has already been proven to be safe. This a priori verification of the entire system is a vital aspect when dealing with adaptive and scalable systems.

In the following chapter, we will relax some of the restrictions imposed on the models allowed in the presented approach. Our technique will, thus, be applicable to an even larger set of systems, which will greatly improve its practicality.

5 Verification of Extended Parameterized Timed Systems

With most of today's systems being safety critical, model-based design processes offer a structured procedure to ensure a certain quality. They allow the formal modeling and verification of properties during the design phase of a system. However, during this phase, the models might be repeatedly reconfigured until a final design is found. In addition, such reconfigurations may also occur at lifetime of the system, e.g., due to self-adaptation. Since the system's behavior might change unexpected, these reconfigurations raise the need to redo verifications that have already been done.

In such a setting, a rerun of the verification from scratch is not efficient. We have, thus, proposed to reuse the result from the previous verification run. In the previous chapter, we have successfully introduced an incremental technique that follows this paradigm for parameterized timed systems modeled via a timed automaton template. It can be applied for various mutual exclusion algorithms or peer to peer protocols. Its benefit is the verification of the safety property for systems with an arbitrary, but fixed number of processes executing such an algorithm. Considering a running system of such processes, the system is safe (w.r.t. the verified safety property) irrespective of the number of processes. In particular, it implies that a reconfiguration of the system by adding or removing a process is safe and does not require a new verification run. This approach enables an a priori verification of the entire system and, thereby, avoids online verifications during lifetime.

In this chapter, we will relax some restrictions imposed on the models for which our technique can be employed to verify safety properties. We broaden the parameterized timed systems that were considered in the previous chapter ($\forall n \in \mathbb{N} : [P_1 || \ldots || P_n]$). Up to now they consisted of a parameterized number n of instantiations of a process P, which introduced a symmetric state space. The extended parameterized timed systems ($\forall n \in \mathbb{N} : [S || \ldots || R || P_1 || \ldots || P_n]$) now additionally include a finite number of extra processes S to R. We also permit a restricted sort of synchronization. As an example, this extension allows to model

an additional communication medium, e.g., a bus. This relaxation provides for a broader applicability of the technique.

Up to now, the models were created as networks of timed automata, where the automata are instances of a timed automaton template. The size of the model was given as parameter denoting the number of automata.

In the extended formalism, an additional, fixed number of extra timed automata accompany the parameterized number of automata instantiated from the template. These extra automata allow for the modeling of components associated with the parameterized system, e.g., a communication medium or a server in combination with the parameterized number of timed automata modeling clients. Furthermore, we allow a limited sort of synchronization between the timed automata via channels with the restriction that at most one of the symmetric automata must be involved in the synchronization.

In the following, we provide the updated definitions used for the models. Afterwards, we redefine the considered *swap* operations that are used to define the intended notion of symmetry. We briefly recall the workflow and give the proof for our Termination Theorem, before concluding this chapter with a short experiment using one such extended model specifying a gate and controller for a train crossing with an arbitrary number of trains. Note, that the previous chapter is a special case of the formalism introduced below. When considering models without extra timed automata and without synchronization, we receive the formalism and notion of symmetry as before.

We start with the redefinition of the employed models.

5.1 Extension of the Modeling Approach

The models considered in this chapter are extensions of those from the previous chapter. They additionally allow extra timed automata in the network that are not instantiated from the template, as well as synchronization with the extra automata. The symmetric automata instantiated from the template can not synchronize with each other, as this would invalidate our Termination Theorem employed for the reasoning about the entire parameterized timed system. We first update the definition of templates in order to reflect the added ability to synchronize edges. Note, that the employed synchronization channel is equal in each instantiation, i.e., the synchronization channel does not depend on the automaton's identifier. In addition, synchronization is only allowed with the extra automata, not among the instantiations of the template. The definition for templates using synchronization is given below.

Definition 5.1.1 (Timed Automaton Template using Synchronization)**.** Let the set C^g of global clocks be given, as well as the global set of integer variables $\mathcal{IV} = \mathcal{IV}_{i\!\!/d} \cup \mathcal{IV}_{id}$ as the union of disjoint sets of identifier unaware ($\mathcal{IV}_{i\!\!/d}$) and aware ($\mathcal{IV}_{id}$) integer variables, s.t. $\mathcal{IV}_{i\!\!/d} \cap \mathcal{IV}_{id} = \emptyset$. In addition, let Σ be the global set of channels. A *symmetric timed automaton template* $\mathcal{A}(pid)$ *using synchronization* defined over glob-

ally shared C^g, Σ and $\mathcal{IV}$ is a tuple $\mathcal{A}(pid) = (L, l_0, C, \mathcal{IV}, \Sigma, Inv^c, Inv^i(pid), E(pid))$ such that

- $pid \in \mathbb{N}_{\geq 1}$ is a unique parameter, called identifier,
- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- $C = C^l \cup C^g$ is the union of the finite and disjoint sets of local and global clocks with initial valuation v_0^c,
- $\mathcal{IV}$ is the finite set of shared integer variables with initial valuation v_0^i,
- Σ is the finite set of shared synchronization channels,
- $Inv^c : L \to \Phi(C)$ is a total function of clock invariants, s.t. $v_0^c \models Inv^c(l_0)$,
- $Inv^i(pid) : L \to \Psi(\mathcal{IV}, pid)$ is a total function of integer invariants, s.t. $v_0^i \models Inv^i(pid)(l_0)$, and
- $E(pid) \subseteq L \times \Sigma_{sync} \times \Phi(C) \times \Psi(\mathcal{IV}, pid) \times \Omega(\mathcal{IV}, pid) \times 2^C \times L$ is the set of edges.

It must hold that $\forall e_1 \in E$ with synchronization label $a? \in \Sigma_{sync}$, there does not exist an edge $e_2 \in E$ with synchronization label $a!$ and vice versa.

As before, instantiating a template $A(pid)$ with unique identifier $pid = j \in \mathbb{N}_{\geq 1}$ results in timed automaton A_j. Unlike before, the networks of timed automata considered in this chapter do not only include instantiations of a template, but additionally contain a finite number of extra timed automata. These supplementary automata have to obey the restrictions for identifier aware integer variables $\mathcal{IV}_{id}$ as defined in the previous chapter. Effectively this means that even in the extra automata the variables that are aware of the automata's identifier can only be applied in restricted constraints and assignments (using each automaton's own identifier). Otherwise, the symmetry of the state space, on which our approach is based, could be destroyed. We formalize our extended formalism with extra automata below.

Definition 5.1.2 (Extended Symmetric Network of Timed Automata)**.** Given $x \in \mathbb{N}_{\geq 0}$ extra timed automata A_1 to A_x with unique identifiers 1 to x that are defined as in Definition 2.1.8, but with the restrictions on integer variables $\mathcal{IV}_{id}$ as above, where in each automaton A_i ($i \in \{1, \ldots, x\}$) only instantiated constraints $\Psi(\mathcal{IV}, i)$ and assignments $\Omega(\mathcal{IV}, i)$ are allowed. Given parameter $n \in \mathbb{N}_{\geq 1}$ and the timed automaton template $\mathcal{A}(pid)$ using synchronization as defined in Definition 5.1.1. The *Extended Symmetric Network of Timed Automata* with $x + n$ timed automata is defined as $NTA_n = \langle A_1, \ldots, A_x, A_{x+1} \ldots, A_{x+n} \rangle$, where A_i ($i \in \{x + 1, \ldots, x + n\}$) is the instantiation of $\mathcal{A}(pid)$ with $pid = i$.

Note, that the definition of symmetric networks of timed automata in the previous chapter is a special case of the above redefinition, in which no synchronization is used and $x = 0$ extra automata are given.

In the following, we illustrate the concept of extra automata in the network with an example first proposed in 1994 [AD94] (here slightly changed).

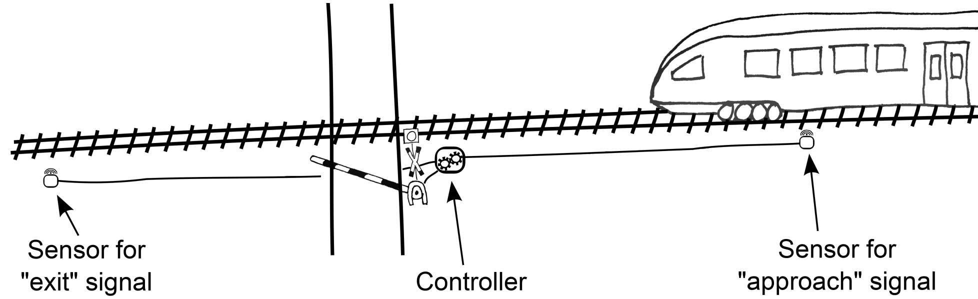


Figure 5.1: Illustration of the Train-Controller-Gate model

Example 5.1.3. Consider a train crossing that contains a gate actuated by a controller as depicted in Figure 5.1. Whenever one of a fixed, but arbitrary number of trains is crossing a sensor, the controller receives the signal *approaching*. It requires 1 time unit to send the *lowering* signal to the gate. The gate needs at most 1 additional time unit, until the gate is lowered. The train reaches the crossing after at least 3 time units and exits it after at most 5 time units. When exiting, a sensor tells the controller that the train has *exited*, which needs up to 1 time unit to signal the gate that it should *raise*. This finishing move requires 1 to 2 time units.

Figure 5.2 shows the two timed automata representing the gate (Figure 5.2(a)) and the controller (Figure 5.2(b)), as well as the template that is instantiated to represent the trains (Figure 5.2(c)). Clearly, no identifier aware integer variable is needed in this scenario. It does, however, make use of the added synchronization capabilities introduced in this chapter. As can be seen, all trains use the same synchronization with the extra automata.

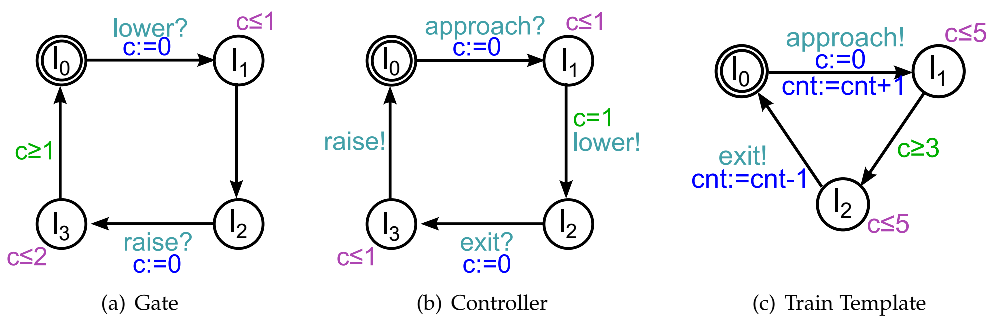


Figure 5.2: *Train Template* and extra timed automata modeling the *Controller* and *Gate*

The introduction of extra automata in the models does not comply with our previous notion of symmetry, as these extra automata can not be swapped with any other one. Thus, we redefine the intended notion such that it only covers the instantiations of the template. To this end, the swap operation needs to be adapted.

5.2 Symmetry

The extra automata introduced above are not instantiated from the template like all the other automata. They are, thus, not symmetric, i.e., they can not be used interchangeably and must not be taken into account in the swap operation. The necessary adaptation to the swap operation is given below. It restricts the swap to be only applicable to symmetric timed automata, i.e., those that are instantiated from the same template. As explained, the intuition behind this restriction is that the symmetric automata can be used interchangeably, while all others can't.

Definition 5.2.1 (Swap)**.** Let an extended network of timed automata $NTA_n = \langle A_1, ..., A_{x+n} \rangle$ be given as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Let A_1 to A_x be the extra timed automata and A_{x+1} to A_{x+n} the symmetric timed automata instantiated from template $\mathcal{A}(pid)$ as defined in Definition 5.1.1. A *swap* $\pi = swap_{i,j}$ swaps two timed automata A_i and A_j ($i, j \in \{x+1, \ldots, x+n\}$) instantiated from $\mathcal{A}(pid)$. It changes the state $s = ((l_1, ..., l_{x+n}), v^c, v^i) \in S$ into the state $\pi(s) = ((\pi(l_1), ..., \pi(l_{x+n})), \pi(v^c), \pi(v^i)) \in S$ with

- $\forall k \in \{1, \ldots, x+n\} : \pi(l_k) = \begin{cases} l_i & if\ k = j, \\ l_j & if\ k = i, \\ l_k & else. \end{cases}$
- $\forall c \in C^g : \pi(v^c)(c) = v^c(c).$
- $\forall k \in \{1, \ldots, x+n\} : \forall c \in C^l_k : \pi(v^c)(c) = \begin{cases} v^c(c_i) & if\ k = j, \\ v^c(c_j) & if\ k = i, \\ v^c(c) & else, \end{cases}$

  where c_i and c_j refer clock c in automaton A_i ($c \in C^l_i$) and A_j ($c \in C^l_j$), respectively.
- $\forall iv \in \mathcal{IV}_{\not{id}} : \pi(v^i)(iv) = v^i(iv).$
- $\forall iv \in \mathcal{IV}_{id} : \pi(v^i)(iv) = \begin{cases} i & if\ v^i(iv) = j, \\ j & if\ v^i(iv) = i, \\ v^i(iv) & else. \end{cases}$

Note, that we also use $\pi(m)$ to denote the identifier of the automaton with which m has been swapped, formally $\pi(m) = \begin{cases} i & if\ m = j, \\ j & if\ m = i, \\ m & else. \end{cases}$

The effect of the *swap* operation is the same as in the previous chapter. It swaps the locations of the two automata, the values of their local clocks, as well as values for identifier aware integer variables that refer to one of the two automata. The only difference is the restriction to be applied only for symmetric timed automata. Again, a permutation involving more than two automata being swapped can easily

be realized via several swap operations in sequence. Note, that the *swap* operation as used in the previous chapter is a special case of the adapted one, namely for those networks that include $x = 0$ extra automata.

As in the previous chapter, we apply the swap operation to define the intended notion of symmetry. Apart from the updated operation, the symmetry notion remains the same. We recall it below.

Definition 5.2.2 (Symmetry of the State Space w.r.t the Symmetric Timed Automata)**.** Let an extended network of timed automata $NTA = \langle A_1, ..., A_{x+n} \rangle$ be given with concrete semantics $TS = (S, s_0, \rightarrow)$. It is *symmetric*, if for any swap π (Def. 5.2.1) and states $s_1, s_2 \in S$ it holds that $s_1 \rightarrow s_2$ with time delay δ and taken edge e of timed automata A_i (synchronized edges e_1, e_2 of A_i, A_j) if and only if $\pi(s_1) \rightarrow \pi(s_2)$ with time delay δ and taken edge e of timed automata $A_{\pi(i)}$ (synchronized edges e_1, e_2 of $A_{\pi(i)}$, $A_{\pi(j)}$). Furthermore, it must hold that $s = s_0$ if and only if $\pi(s) = s_0$.

This definition requires the initial state to be symmetric, as well as paths to be symmetric. This implies for a state s that is reachable via a path from an initial state, that all its swapped states $\pi(s)$ are reachable via the respective swapped paths. Note, that the notion of symmetry characterizes π to be an automorphism on the transition system that is the concrete semantics TS (c.f. [Hen+04]). In the Appendix B.1 we prove that all networks of timed automata created as defined in Definition 5.1.2 meet this notion of symmetry.

In addition to the adaptation of the above definitions, we need to update the definition of safety properties. Previously, these were defined over every permutation of automata in the network. Given a model that includes extra automata, we have to consider only those permutations that permute symmetric automata instantiated from the template. The intended notion of a symmetric safety property is, thus, defined using the adapted definition of the *swap* operation.

Definition 5.2.3 (Extended Symmetry of a Safety Property)**.** Let an extended network of timed automata $NTA = \langle A_1, ..., A_{x+n} \rangle$ be given with concrete semantics $TS = (S, s_0, \rightarrow)$. Given any swap π (Def. 5.2.1) and state $s \in S$, a *safety property ρ is symmetric* if it holds that $s \models \rho$ if and only if $\pi(s) \models \rho$.

This updated notion of symmetry is also reflected in the redefinition of safety properties. The adapted definition includes all permutations of symmetric timed automata in the network and, thus, is inherently symmetric in the above sense. First, we redefine the employed error state specification.

Definition 5.2.4 (Extended Symmetric Error State Specification)**.** Given an error state specification $err = (\bar{l_m}, \phi, \psi)$ for an extended symmetric network of timed automata NTA_m (Definition 5.1.2), where $\psi = \psi_0 \wedge \psi_1(1) \wedge \cdots \wedge \psi_{x+m}(x+m)$ contains parameterized constraints for the $x + m$ automata. The *scaled error state specification* for NTA_n ($n \geq m$) is defined as $err^n = (\bar{l}, \phi, \psi)$ with $\forall i \in \{1, \ldots, x+m\} : \bar{l}[i] = \bar{l_m}[i]$ and $\forall i \in \{x+m+1, \ldots, x+n\} : \bar{l}[i] = *$.

The *symmetric error state specification* for NTA_n is defined as $err^n_{sym} = \exists \pi : err^n_\pi$ for permutation π according to the redefined swap operation (Def. 5.2.1) with $err^n_\pi = (\pi(\bar{l}), \pi(\phi), \pi(\psi))$ being defined as

- $\forall i \in \{1, \ldots, x+n\} : \pi(\bar{l})[i] = \bar{l}[\pi(i)]$,
- $\pi(\phi) = \begin{cases} \pi(\phi_1) \wedge \pi(\phi_2) & \text{if } \phi = \phi_1 \wedge \phi_2, \\ (\pi(x) - \pi(y)) \bowtie n & \text{if } \phi = (x - y) \bowtie n, \\ \pi(x) \bowtie n & \text{if } \phi = x \bowtie n, \\ true & \text{if } \phi = true, \end{cases}$
  where $\pi(x)$ refers to clock x itself if $x \in C^g$ or otherwise refers to the respective clock in $C^l_{\pi(i)}$ if $x \in C^l_i$,
- $\pi(\psi) = \psi_0 \wedge \psi_1(\pi(1)) \wedge \cdots \wedge \psi_{x+m}(\pi(x+m))$.

As in the previous chapter, we implement the existential quantification by enumeration. The application of the redefined *swap* operation ensures the intended notion of symmetry within the employed safety properties. There is no need to redefine the safety properties or the resulting verification, else than using the above adapted definitions. However, we recall the definitions for symmetric safety properties and the verification question below for readability reasons.

Definition 5.2.5 (Symmetric Safety Property)**.** Let NTA_n be an extended network of $x+n$ timed automata. The *symmetric safety property* is defined as $\rho^n = \neg(err1^n_{sym}) \wedge \neg(err2^n_{sym}) \wedge \ldots$ for symmetric error state specifications $err1^n_{sym}, err2^n_{sym}, \ldots$.

> **Verification Question:** Given the timed automaton template $\mathcal{A}(pid)$ and a symmetric safety property ρ, does the safety property ρ^n hold for all extended networks of timed automata NTA_n consisting of $n \in \mathbb{N}_{\geq 1}$ instantiations of the template with x extra timed automata?

In the following, we continue the above Train-Controller-Gate example and illustrate the updated definitions.

Example 5.2.6. Let the above example be given with a controller and a gate guarding a train crossing with a fixed, but arbitrary number of trains. The safety property, which we are interested in, specifies that when a train is in the crossing (location l_2 of the *Train* automaton), then the gate has to be lowered entirely (location l_2 of the *Gate* automaton). In other words, an undesired error occurs if the train automaton is in location l_2, but the gate automaton is in a location distinct from l_2. This fact can be specified in three error state specifications, namely $err1 = ((l_0, *, l_2), true, true)$ and $err2 = ((l_1, *, l_2), true, true)$ and $err3 = ((l_3, *, l_2), true, true)$. They are defined over $m = 1$ symmetric automata. The scaled versions for $n = 3$ symmetric automata are $err1^3 = ((l_0, *, l_2, *, *), true, true)$ and $err2^3 = ((l_1, *, l_2, *, *), true, true)$ and $err3^3 = ((l_3, *, l_2, *, *), true, true)$.

Using enumeration, the symmetric specifications are given below.

$$\begin{aligned} err1^3_{sym} =& ((l_0, *, l_2, *, *), true, true) \\ & \vee ((l_0, *, *, l_2, *), true, true) \\ & \vee ((l_0, *, *, *, l_2), true, true) \end{aligned}$$

$$\begin{aligned} err2^3_{sym} =& ((l_1, *, l_2, *, *), true, true) \\ & \vee ((l_1, *, *, l_2, *), true, true) \\ & \vee ((l_1, *, *, *, l_2), true, true) \end{aligned}$$

$$\begin{aligned} err3^3_{sym} =& ((l_3, *, l_2, *, *), true, true) \\ & \vee ((l_3, *, *, l_2, *), true, true) \\ & \vee ((l_3, *, *, *, l_2), true, true) \end{aligned}$$

If any of them is satisfied by a state s, then s is an error state. Trivially, any swapped state $\pi(s)$ is also an error state, since all permutations of the error state specification are considered. The resulting safety property for $n = 3$ symmetric automata is $\rho^3_{lowered} = \neg err1^3_{sym} \wedge \neg err2^3_{sym} \wedge \neg err3^3_{sym}$. Note, that it requires the error state specification to be scaled for the respective number of symmetric automata.

As can be seen, the adaptations to the formalism and definitions are, in total, small. Using the redefined symmetry, the incremental workflow from the previous chapter can be employed unchanged, except that the extrapolation procedure has to adhere the adapted swap operation, i.e., it only affects literals referring automata A_{x+1} to A_{x+n}. The incremental workflow including the optimizations as proposed in the previous chapter can be found in Figure 5.3.

Using the adapted extrapolation procedure, the Termination Theorem holds true unmodified. We recall it below and give the proof.

Theorem 5.2.1 (Termination Theorem)**.** *Given the SMT-formula $\|F\|$ representing an inductive strengthening F of ρ^n for NTA_n. If $\|F\|^{exp(n+1)}$ is an inductive strengthening of ρ^{n+1} for NTA_{n+1}, then $\|F\|^{exp(n+2)}$ is an inductive strengthening of ρ^{n+2} for NTA_{n+2}.*

In the following, we provide the proof of correctness. The referred lemmata can be found in Appendix B.2.

Proof: Assume the contrary, meaning $\|F\|^{exp(n+2)}$ is not an inductive strengthening of ρ^{n+2} for NTA_{n+2}, but $\|F\|^{exp(n+1)}$ is a valid inductive strengthening of ρ^{n+1} for NTA_{n+1}. At least one of the three properties *Initiation*, *Consecution* or *Strengthen* is not met by $\|F\|^{exp(n+2)}$, since it is not a valid inductive strengthening.

We show that each of these three violated properties leads to a contradiction to the assumption that $\|F\|^{exp(n+1)}$ is a valid inductive strengthening. Thus, the theorem holds true.

Let $TS^{n+2} = (S^{n+2}, s_0^{n+2}, \rightarrow^{n+2})$ be the concrete semantics of NTA_{n+2} and $TS^{n+1} = (S^{n+1}, s_0^{n+1}, \rightarrow^{n+1})$ be the concrete semantics of NTA_{n+1}.

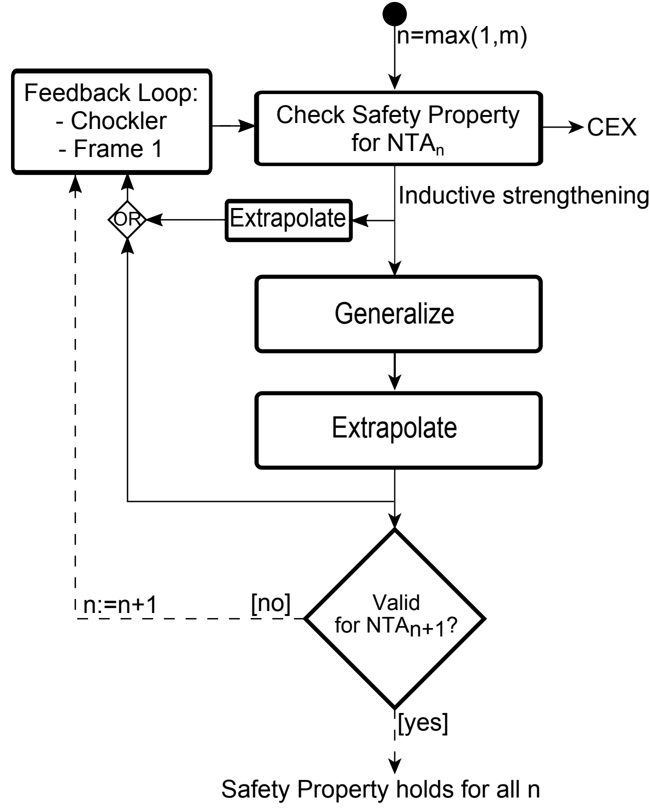


Figure 5.3: Optimized workflow

- If *Initiation* does not hold true for $\|F\|^{exp(n+2)}$ and NTA_{n+2}, there exists an initial state $s = s_0^{n+2}$ that is not a member in the set of states represented by $\|F\|^{exp(n+2)}$. Clearly, s is not a member of at least one of the permutations of $\|F\|$, as otherwise s would be a member of $\|F\|^{exp(n+2)}$. We denote this permutation of $\|F\|$ as $\pi(\|F\|)$. With $\|F\|$ being an inductive strengthening computed by *IC3 with Zones* for NTA_n, it reasons only about n or less of the symmetric timed automata A_{x+1} to A_{x+n}. Thus, the permutation $\pi(\|F\|)$ also only reasons about n or less symmetric timed automata in A_{x+1} to A_{x+n+2}. In consequence, there exist at least two symmetric timed automata that are not referred in $\pi(\|F\|)$. Either the timed automaton A_{x+n+2} is among these or not.
  - If A_{x+n+2} is not referred in $\pi(\|F\|)$, then we know that $\pi(\|F\|)$ is a conjunct in $\|F\|^{exp(n+1)}$. In addition, Lemma B.2.7 guarantees that the reduced state (Def. B.2.1) $s|_{n+1}$, in which the location and values of local clocks of A_{x+n+2} are discarded, is not a member of $\pi(\|F\|)$. It is, thus, not a member of the set of states represented by $\|F\|^{exp(n+1)}$. Furthermore, Lemma B.2.5 guarantees the reduced state $s|_{n+1}$ to be initial in NTA_{n+1}. Clearly, *Initiation* of $\|F\|^{exp(n+1)}$ for NTA_{n+1} does not hold, which is a contradiction.

- If A_{x+n+2} is referred in $\pi(\|F\|)$, we deduce the same result using a swap. Since A_{x+n+2} is referred in the formula, there exist two symmetric automata A_{x+i} and A_{x+j} ($i \neq j \in \{1, \ldots, n+1\}$) that are not. We use the swap π_1 that swaps automata A_{x+n+2} and A_{x+i}. As a result, we know that $\pi_1(\pi(\|F\|))$ is a conjunct of $\|F\|^{exp(n+1)}$ as it does not reason about A_{x+n+2}. Lemma B.2.1 states that $\pi_1(s)$ is not a member of $\pi_1(\pi(\|F\|))$. Lemma B.2.7 guarantees that the reduced state (Def. B.2.1) $\pi_1(s)|_{n+1}$, in which the location and values of local clocks of A_{x+n+2} are discarded, is not a member of $\pi_1(\pi(\|F\|))$. Thus, $\pi_1(s)|_{n+1}$ is not a member of $\|F\|^{exp(n+1)}$. Furthermore, with Lemma B.1.1, we know that $\pi_1(s)$ is still initial and Lemma B.2.5 guarantees the reduced state $\pi_1(s)|_{n+1}$ to be initial in NTA_{n+1}. Clearly, *Initiation* of $\|F\|^{exp(n+1)}$ for NTA_{n+1} does not hold, which is a contradiction.

In summary, we have reached a contradiction in both cases.

- For *Strengthen*, a similar argumentation can be carried out as above. There exists a state $s \in S^{n+2}$ that is a member of $\|F\|^{exp(n+2)}$, but violates the safety property ρ^{n+2}. Since the symmetric safety property ρ^{n+2} is violated, one of the symmetric error state specification has to be satisfied. The symmetric error state specification is a disjunction of all permutations of the scaled specification for $m \leq n$ symmetric timed automata. Thus, one of these permutations must be satisfied. Let err_{π}^{n+2} be the respective specification. Again, we split the proof into two cases depending on whether the automaton A_{x+n+2} is referred in err_{π}^{n+2} or not.

  - If A_{x+n+2} is not referred in err_{π}^{n+2}, we know that this specification is part of the symmetric safety property ρ^{n+1}. Lemma B.2.8 states that the reduced state $s|_{n+1}$ satisfies err_{π}^{n+1} and, thus, violates ρ^{n+1}. Furthermore, Lemma B.2.6 states that $s|_{n+1}$ is a member of $\|F\|^{exp(n+1)}$. Thus, *Strengthen* of $\|F\|^{exp(n+1)}$ does not hold for NTA_{n+1}, which is a contradiction.

  - If A_{x+n+2} is referred in err_{π}^{n+2}, we deduce the same result using a swap. With $m \leq n$ symmetric timed automata referred in err_{π}^{n+2}, we know that at least two automata A_{x+i} and A_{x+j} ($i \neq j \in \{1, \ldots, n+1\}$) exist that are not referred. We use the swap π_1 that swaps automata A_{x+n+2} and A_{x+i}. By symmetry, we know that $err_{\pi_1(\pi)}^{n+2}$ is satisfied by $\pi_1(s)$. Corollary B.2.3 states that $err_{\pi_1(\pi)}^{n+2}$ does not specify values for A_{x+n+2}. Using Lemma B.2.8, we know that the reduced state $\pi_1(s)|_{n+1}$ satisfies $err_{\pi_1(\pi)}^{n+1}$. Thus, $\pi_1(s)|_{n+1}$ violates ρ^{n+1}. Furthermore, Corollary B.2.2 states that $\pi_1(s)$ is still an element in $\|F\|^{exp(n+2)}$. With Lemma B.2.6, $\pi_1(s)|_{n+1}$ is a member of $\|F\|^{exp(n+1)}$. Thus, *Strengthen* of $\|F\|^{exp(n+1)}$ does not hold for NTA_{n+1}, which is a contradiction.

In summary, we have reached a contradiction in both cases.

- If *Consecution* does not hold, there exist states $s, t \in S^{n+2}$, s.t. $(s,t) \in \rightarrow^{n+2}$ via one or two edges. By the definition of synchronization in timed automata templates, we know that at most one of the edges is taken in automata A_{x+1} to A_{x+n+2}. Assume w.l.o.g. that this edge is not taken in A_{x+n+2}. This assumption holds true due to symmetry, as otherwise, we could use a swap that swaps the automaton A_{x+n+2} with any automaton in A_{x+1} to A_{x+n+1} (Lemmata B.1.2, B.1.3 and B.1.4). The resulting swapped states would fulfill the above assumption, while still violating consecution (Corollary B.2.2).

  Since the combination of s and t violates consecution, t is not a member of $\|F\|^{exp(n+2)}$. Clearly, t is not a member of at least one of the permutations of $\|F\|$. We denote this permutation of $\|F\|$ as $\pi(\|F\|)$. With $\|F\|$ being an inductive strengthening computed by *IC3 with Zones* for NTA_n, it reasons only about n or less of the symmetric timed automata A_{x+1} to A_{x+n}. Thus, the permutation $\pi(\|F\|)$ also only reasons about n or less symmetric timed automata in A_{x+1} to A_{x+n+2}. In consequence, there exist at least two symmetric timed automata that are not referred in $\pi(\|F\|)$. Either the timed automaton A_{x+n+2} is among these or not.

  - If A_{x+n+2} is not referred in $\pi(\|F\|)$, then we know that $\pi(\|F\|)$ is a conjunct in $\|F\|^{exp(n+1)}$. In addition, Lemma B.2.7 guarantees that the reduced state (Def. B.2.1) $t|_{n+1}$, in which the location and values of local clocks of A_{x+n+2} are discarded, is not a member of $\pi(\|F\|)$. It is, thus, not a member of the set of states represented by $\|F\|^{exp(n+1)}$. Lemmata B.2.9, B.2.10 and B.2.11 guarantee that edges can still be taken for the reduced states, i.e., $(s|_{n+1}, t|_{n+1}) \in \rightarrow^{n+1}$. Furthermore, Lemma B.2.6 guarantees the reduced state $s|_{n+1}$ to be a member of $\|F\|^{exp(n+1)}$. Clearly, *Consecution* of $\|F\|^{exp(n+1)}$ for NTA_{n+1} does not hold, which is a contradiction.
  - If A_{x+n+2} is referred in $\pi(\|F\|)$, we deduce the same result using a swap. Since A_{x+n+2} is referred in the formula, there exist two symmetric automata A_{x+i} and A_{x+j} ($i \neq j \in \{1, \ldots, n+1\}$) that are not. Using the above knowledge that at most one edge is taken in automata A_{x+1} to A_{x+n+1}, we know that at least one of the two automata A_{x+i} and A_{x+j} is not used in the edges. W.l.o.g assume A_{x+i} to be the one that is not used. In consequence, we can use the swap π_1 that swaps automata A_{x+n+2} and A_{x+i}. The swapped states still violate consecution using the same edges as the before (Lemmata B.1.2, B.1.3 and B.1.4). In addition, we know that $\pi_1(s)$ is still a member of $\|F\|^{exp(n+2)}$ and that $\pi_1(t)$ is not a member of $\|F\|^{exp(n+2)}$ (Corollary B.2.2). For the reduced states, we conclude the following by taking into account that A_{x+n+2} is not referred in $\pi_1(\pi(\|F\|))$ and no edge is taken in automaton A_{x+n+2}. $\pi_1(s)|_{n+1}$ is a member of $\|F\|^{exp(n+1)}$ (Lemma B.2.6). $\pi_1(t)|_{n+1}$ is not a member of $\|F\|^{exp(n+1)}$ (Lemma B.2.7), which includes the permutation $\pi_1(\pi(\|F\|))$ that does not reason about A_{x+n+2} (Lemma B.2.1). Lemmata B.2.9, B.2.10 and B.2.11 guarantee that the edges can still be taken for the reduced

states. Clearly, *Consecution* of $\|F\|^{exp(n+1)}$ for NTA_{n+1} does not hold, which is a contradiction.

In summary, we have reached a contradiction in both cases.

□

We demonstrate the practicality of the extended approach in the following.

5.3 Experiments

Considering the above example of the train crossing, we have conducted experiments with two distinct safety properties. The first property $\rho_{lowered}$ has been presented above (Example 5.2.6) and specifies that the gate has to be closed entirely, when a train is crossing. The second one ($\rho_{occupied}$) denotes that the crossing is not occupied for no reason, i.e., the gate is only lowering, if at least one train is approaching or in the crossing. It is specified via the error state specification $err = ((l_1, *, *), true, cnt < 1)$ using an identifier unaware variable *cnt* that counts the trains that are not far away (that are not in location l_0).

Our workflow successfully verifies these safety properties. The respective runtimes with and without optimizations are shown in Tables 5.1 and 5.2.

	Runtime (s)	Verified models NTA_n for which n?
No Optimization	OOM	1-8
Generalization	8,5	$\forall n \in \mathbb{N}$
Chockler-Feedback	7,4	$\forall n \in \mathbb{N}$
Generalization and *Chockler*-Feedback using non-generalized set of clauses	3,6	$\forall n \in \mathbb{N}$
Generalization and *Chockler*-Feedback using generalized set of clauses	3,6	$\forall n \in \mathbb{N}$
*Frame*1-Feedback	3,0	$\forall n \in \mathbb{N}$
Generalization and *Frame*1-Feedback using non-generalized set of clauses	2,6	$\forall n \in \mathbb{N}$
Generalization and *Frame*1-Feedback using generalized set of clauses	2,5	$\forall n \in \mathbb{N}$
Both-Feedback	4,8	$\forall n \in \mathbb{N}$
Generalization and *Both*-Feedback using non-generalized set of clauses	3,3	$\forall n \in \mathbb{N}$
Generalization and *Both*-Feedback using generalized set of clauses	3,5	$\forall n \in \mathbb{N}$

Table 5.1: Verification experiments for the safety property $\rho_{lowered}$ and the Train-Controller-Gate model

As can be seen, both safety properties have been successfully verified within reasonable time. Again, the optimization of using the *Frame*1-Feedback has shown to be a good option.

	Runtime (s)	Verified models NTA_n for which n?
No Optimization	OOM	1-7
Generalization	13,0	$\forall n \in \mathbb{N}$
Chockler-Feedback	OOM	1-9
Generalization and *Chockler*-Feedback using non-generalized set of clauses	OOM	1-9
Generalization and *Chockler*-Feedback using generalized set of clauses	6,2	$\forall n \in \mathbb{N}$
*Frame*1-Feedback	25,7	$\forall n \in \mathbb{N}$
Generalization and *Frame*1-Feedback using non-generalized set of clauses	4,6	$\forall n \in \mathbb{N}$
Generalization and *Frame*1-Feedback using generalized set of clauses	OOM	1-7
Both-Feedback	11,4	$\forall n \in \mathbb{N}$
Generalization and *Both*-Feedback using non-generalized set of clauses	8,7	$\forall n \in \mathbb{N}$
Generalization and *Both*-Feedback using generalized set of clauses	OOM	1-7

Table 5.2: Verification experiments for the safety property $\rho_{occupied}$ and the Train-Controller-Gate model

5.4 Summary

In summary, the presented extensions to the allowed modeling formalism in our parameterized setting are valuable in that they significantly increase the modeling capabilities without a drawback. They allow the modeling of server-client situations, as well as other effects like a communication medium.

The necessary adaptations were small and we were able to use the overall workflow, as well as the Termination Theorem without modifications. The experiments were successful and promising and emphasize the practicality of the approach.

Using this extended formalism, our incremental workflow enables the verification of safety properties for a broad number of parameterized timed systems that are composed of a finite number of auxiliary processes accompanying a fixed, but arbitrary large number of instantiations of the same process. This extension adds significant value to our technique. Yet, it's application area remains the a priori verification of properties for an arbitrary, but fixed number of equal processes. This context is in line with the paradigm of Industry 4.0, which facilitates the adaptation of systems during lifetime. The benefits of our proposed technique for this scenario are obvious. It avoids an online verification by executing an a priori verification, which removes the necessity to verify every single reconfigured model, in which the number of processes has been changed.

Other reconfigurations, however, can not be handled using this technique. We will examine the effects of general reconfigurations in the following chapter and give a best-guess approach, as the effects of reconfigurations can not be estimated easily in general.

6 Inductive Verification of Reconfigured Models

Timed formalisms are of significant importance for the modeling of today's systems. As explained, most of these systems are safety critical and, thus, model-based design processes are employed to ensure a certain quality. To this end, formal models are created and properties are verified for these models. During such design procedures, however, the models might repeatedly be reconfigured, resulting in a need to redo the verification. These reconfigurations might also occur later on, e.g., if the model represents an adaptive and self-optimizing system as in Industry 4.0.

As every such reconfiguration changes the state space of the model, the desired safety properties need to be verified again. Doing this from scratch is very inefficient, in particular, when considering that the original, similar model has been examined before.

In the previous two chapters, we have proposed a technique that verifies safety properties for parameterized timed systems. Within such restricted models, our technique prevents the costly verification from scratch when dealing with reconfigurations that change the number of processes. It does so using an incremental workflow specifically designed for parameterized systems. One of the optimizations presented for this technique, namely the *Feedback*-loop, is suitable to be used in a general setting.

It injects an inductive strengthening computed for the original model into the verification run of the reconfigured model. By doing so, some of the clauses in the inductive strengthening are injected into all frames of the IC3 run, or at least into the frame F_1 (see Subsection 4.6.1). Our experiments have shown that this reuse of clauses successfully prevents the costly rediscovery of some of them and, by doing so, speeds up the entire verification run.

So far, we have employed this acceleration technique only for symmetric reconfigurations in the setting of parameterized timed systems. In the following, we will try to utilize its benefits for reconfigured models in general and examine

the effectiveness in several experiments. We start with a classification of possible reconfigurations.

Taking into account the generality of models expressible with our formalism of networks of timed automata (Chapter 2), several types of reconfiguration are possible as detailed in the following. We characterize reconfigurations of a model within the following three categories:

- Addition of new parts to the model,
- Deletion of parts of the model,
- Replacement/Modification of parts of the model.

We illustrate these categories below.

Addition Analog to the addition of a symmetric timed automaton as in the previous chapter, other parts can be added to a model. When considering a network of timed automata modeling the interaction of several components in the real world, the addition of an extra timed automaton may represent the addition of a new component interacting with the others. Furthermore, smaller additions are possible. An extra location might model a new discrete state of a system, an edge models additional interaction. Note, that the addition of a part to the model does not always result in additional behavior, e.g., a constraint added to an edge restricts the existing behavior.

Deletion In contrast, deleted portions of the model might specify the removal of a part of the system, a state or interaction. These would be performed via the removal of an entire timed automaton in the model, or of a location or edge, respectively. Even smaller changes to the constraints or updates of edges can be considered, where it should be noted that the removal of a constraint might allow additional behavior of the model.

As illustrated, the options to modify real-world systems and, in consequence, their models are diverse. The third category contains reconfigurations that are very specific as they replace detailed parts of the model.

Replacement In addition to the above mentioned reconfigurations, there exist those that alter parts of the model without addition or deletion. These reconfigurations essentially rely on clock or integer constraints being altered, or assignments and resets being changed. They correspond, for instance, to the adaptation of timing parameters in the modeled systems.

The above variety of reconfigurations makes the reuse of previous verification results extremely challenging. In this chapter, we propose a general procedure to employ these results anyhow. It is heavily based on the *Feedback*-techniques presented in the previous chapter, which allow the reuse of several clauses of an

inductive strengthening computed for the original model in order to speed up the verification run for the reconfigured model.

Our algorithm presented in Chapter 3 successfully combines the IC3 algorithm with the Zone abstraction for the verification of safety properties for networks of timed automata. In case the safety property is invariant, it yields an inductive strengthening of the safety property as additional outcome. Via an efficient validity check, the inductive strengthening can easily be checked to guarantee the safety property's invariance for the original model. In combination with a reconfigured model, however, a simple check for validity of the inductive strengthening will most often fail though the safety property is indeed invariant. The reason is that the formula is not inductive for the reconfigured model. The failed inductiveness originates from a changed state space. Depending on the performed reconfiguration, the change to the state space may be drastic.

In the previous two chapters, this change was anticipated based on the structure of the models, i.e., the symmetry, via an extrapolation procedure. This procedure adapts the inductive strengthening as anticipated, s.t. it is close to a valid inductive strengthening for the reconfigured model. However, due to the large variety of reconfigurations, the change of state space can not be anticipated in general.

In order to give an intuition why the reuse of a formula might be of benefit even without an adaptation, consider the following example.

Example 6.0.1. Let the *Fischer_U_2* model be given as depicted in Figure 3.1. The inductive strengthening of the safety property $\rho := \neg(cnt > 1)$ computed by our algorithm *IC3 with Zones* is listed in the first column of Table 6.1

Fischer_U_2	*Fischer_U_2*(2048)
$(int_1 \leq 1)$	$(int_1 \leq 1)$
$\wedge((int_0 \neq 0) \vee (int_1 \leq 0))$	$\wedge((int_0 \neq 0) \vee (int_1 \leq 0))$
$\wedge(\neg l_0^1 \vee l_1^1 \vee (int_1 \leq 0))$	$\wedge(\neg l_0^1 \vee l_1^1 \vee (int_1 \leq 0))$
$\wedge(\neg l_0^2 \vee l_1^2 \vee (int_1 \leq 0))$	$\wedge(\neg l_0^2 \vee l_1^2 \vee (int_1 \leq 0))$
$\wedge(l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$	$\wedge(l_0^1 \vee (int_0 \neq 1) \vee (int_1 \leq 0))$
$\wedge(l_0^2 \vee (int_0 \neq 2) \vee (int_1 \leq 0))$	$\wedge(l_0^2 \vee (int_0 \neq 2) \vee (int_1 \leq 0))$
$\wedge((c_0^1 \leq 1024.0) \vee (int_0 \neq 1) \vee (c_0^2 > 1024.0))$	$\wedge((c_0^1 \leq 2048.0) \vee (int_0 \neq 1) \vee (c_0^2 > 2048.0))$
$\wedge((c_0^2 \leq 1024.0) \vee (int_0 \neq 2) \vee (c_0^1 > 1024.0))$	$\wedge((c_0^2 \leq 2048.0) \vee (int_0 \neq 2) \vee (c_0^1 > 2048.0))$

Table 6.1: Inductive strengthenings computed for the models *Fischer_U_2* and the reconfigured model *Fischer_U_2*(2048) as depicted in Figure 6.1

When changing the timing parameters of the Fischer algorithm, a reconfigured model might look as depicted in Figure 6.1. It equals the original model, except that the waiting times that the algorithm adheres to are now doubled. The corresponding inductive strengthening for the reconfigured model computed without the reuse of the original inductive strengthening is listed in the second column of Table 6.1.

Clearly, the two inductive strengthenings are similar. In fact, they both consists of eight clauses and differ only at literals referring the time constants. In particular, the first six clauses are exactly the same. Thus, the injection of the former inductive strengthening into the run of our algorithm *IC3 with Zones* (Chapter 3) to verify the

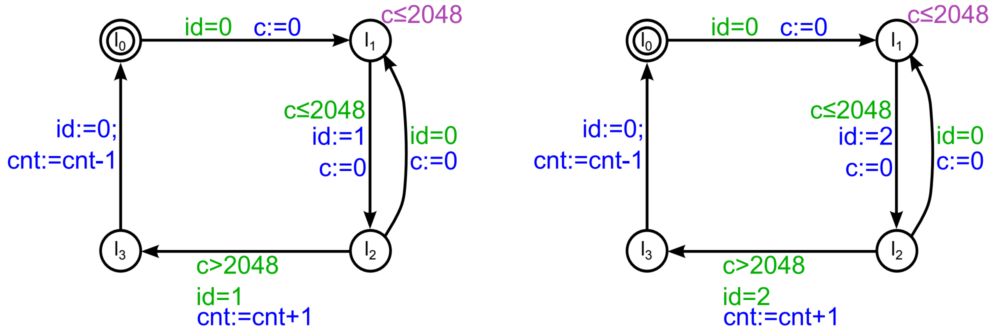


Figure 6.1: Reconfigured Fischer model *Fischer_U_2*(2048) with altered time constants, where all constants 1024 are replaced by 2048

reconfigured model, should prevent the costly rediscovery of these six clauses and, thus, speed up the verification run.

However, the injection of a previously computed inductive strengthening might not always be beneficial, as observable in the runtimes depicted in Table 6.2. For the model *Fischer_U_15*(2048) with 15 processes, it compares a verification *from scratch* as described in Chapter 3 to the one that utilizes the inductive strengthening previously computed for the original model *Fischer_U_15*. We show the results for all three distinct *Feedback*-techniques presented in the previous chapter (*Chockler*-Feedback [Cho+11], *Frame*1-Feedback, *Both*-Feedback). All of them are clearly inferior to a verification from scratch. The reason is that the change in the model is rather drastic (all time constants are changed) resulting in a significant change of the state space concerning the time domain. In addition, the ration of clauses that were of value, i.e., actually occur in the new inductive strengthening computed for the reconfigured model, is rather small (compared to the 6 out of 8 clauses reused for *Fischer_U_2*(2048)).

However, the overall idea of reusing old verification results might be of value in the general setting even so. Interestingly, in the symmetric setting of the previous chapters the reuse could be optimized. The employed extrapolation procedure

	Runtime (s)	Memory (MB)
Verification reconfigured model (from scratch)	140,6	267,8
Verification reconfigured model with *Chockler*-Feedback	176,8	287,5
Verification reconfigured model with *Frame*1-Feedback	245,9	245,1
Verification reconfigured model with *Both*-Feedback	359,5	261,5

Table 6.2: Comparison of the runtimes for verification *from scratch* or using the distinct *Feedback*-techniques with model *Fischer_U_15*(2048)

adapts the original inductive strengthening in order to reflect the reconfiguration, that is the addition of a symmetric timed automaton. In general, this modification is not mandatory. In the symmetric setting, however, the results proved to be of significant value in form of the presented workflow. Hence, we will examine the feasibility of an adaptation procedure for the general case in the following.

6.1 Adaptation of the Formula

Optimally, an adaptation of the inductive strengthening formula according to the applied reconfiguration would result in a valid inductive strengthening for a reconfigured model. To illustrate such an optimal approach, consider the following example.

Example 6.1.1. Consider again the original *Fischer_U_*2 model and the reconfigured *Fischer_U_*2(2048) model as explained in the previous example. They are depicted in Figures 3.1 and 6.1, where the reconfiguration replaces all time constants 1024 in the original model by the constant 2048. An optimal adaptation of the formula would change all time constants according to the reconfiguration, i.e., replace all constants 1024 in the formula by 2048. The resulting formula is a valid inductive strengthening for the reconfigured model. In fact, it equals the inductive strengthening for the reconfigured model listed before in the second column of Table 6.1.

We also performed the adaptation with the *Fischer_U_*15 example above, in which the reuse of the inductive strengthening was unsuccessful. When comparing the runtimes of the verifications, it is obvious that the adapted formula improves on the non-adapted one. The results are shown in Table 6.3, which displays the runtimes of *IC3 with Zones* with injection of the adapted inductive strengthening using the distinct *Feedback*-techniques. Furthermore, it includes the runtime of a simple validation of the adapted inductive strengthening as a reference time. The comparison with Table 6.2 shows a significant improvement over the reuse of the original (non-adapted) inductive strengthening and over the verification from scratch.

This example represents an optimal setting, where a valid inductive strengthening for the reconfigured model could be produced by adapting the inductive strengthening for the original model. It is, however, not representative, as many challenges hinder such perfect adaption in the usual case. This is due to the fact that the impact of a reconfiguration can hardly be estimated, in particular if the reconfiguration is more complex. In the following, we propose adaptations to the formula for the three distinct categories of reconfigurations shown above.

Adaptation to *Addition*-Reconfigurations The impossibility of an adaptation can best be observed when considering reconfigurations that include the addition of parts to the model. In general, an additional timed automaton, as well as additional edges or locations in existing automata introduce additional behavior in interaction with the original parts of the model. This means the state space may change in

	Runtime (s)	Memory (MB)
Validation of adapted inductive strengthening	1,1	48,4
Verification reconfigured model with *Chockler*-Feedback (adapted inductive strengthening)	36,6	100,0
Verification reconfigured model with *Frame*1-Feedback (adapted inductive strengthening)	25,8	82,8
Verification reconfigured model with *Both*-Feedback (adapted inductive strengthening)	36,7	102,2

Table 6.3: Comparison of the runtimes for validation and verification injecting the adapted formula using the distinct *Feedback*-techniques with model $Fischer_U_15(2048)$

various ways, which can not be estimated without knowledge about any structural characteristics like symmetry. Thus, a perfect adaptation of the original inductive strengthening is not feasible in general. For reconfigurations that include the addition of parts, we are unable to adapt the formula, as we can not estimate the changed state space. In consequence, the formula remains unchanged with the hope that the state space has not changed, or at least is similar to the original one.

Adaptation to *Deletion*-Reconfigurations The same argumentation holds true when considering reconfigurations including the deletion of parts of the model. As above, the effect on the state space can hardly be estimated. For illustration purpose, consider the deletion of a single constraint. Its removal weakens the restrictions on an edge or location and might allow for additional behavior that might increase the reachable portion of the state space, s.t. a state violating the safety property is now reachable. Adapting the inductive strengthening of the safety property according to these changes in the state space is not feasible. However, when considering the removal of other parts of a model, e.g., an entire timed automaton, the formula can, and in fact must, be adapted to reflect this change. It would otherwise include variables that do not exist in the encoding of the model. Thus, we adapt the formula as follows.

There might exist literals in some of the clauses of the formula that refer to deleted parts removed during the reconfiguration. We propose two distinct options how these literals can be handled.

Definition 6.1.2 (Adaptation to Deleted Parts)**.** Let an inductive strengthening F of a safety property be computed for an original model NTA. We *adapt the formula* $||F||$ concerning parts of the model deleted in a reconfiguration as follows. For each clause c in $||F||$ that includes at least one literal referring a nonexistent part (clock, location or integer variable) in the reconfigured model, we either

- Delete the entire clause, or
- Delete all disjuncts that refer nonexistent parts in the reconfigured model.

This approach is illustrated in the following example.

Example 6.1.3. Consider a model and a reconfiguration in which a local clock c_0 in timed automaton with identifier 1 is deleted. As a result, the respective clock variable c_0^1 must not be used and has to be removed from the inductive strengthening formula. However, simply deleting the clock variable is not an option, as it would result in an invalid formula that includes deformed constraints. Considering the clause $l_0^1 \vee (c_0^1 \geq 1)$ illustrates this issue. Deleting the clock variable c_0^1 encoding clock c_0 in timed automaton with identifier 1 results in the invalid formula $l_0^1 \vee (\geq 1)$. The above presented procedures of deleting the entire clause or only the affected literals yield distinct results. Either the clause is discarded in its entirety or the disjunct $(c_0^1 \geq 1)$ is removed resulting in clause l_0^1.

Adaptation to *Replacement*-Reconfigurations Lastly, reconfigurations that alter the model by specific replacements of parts offer the best chance to adapt the inductive strengthening accordingly. This is due to the structure of these reconfigurations, where the model is modified without addition or deletion. Thus, these reconfigurations include the modification of constraints, assignments and sets of clocks to be reset. The latter two modifications (reset clocks and assignments) are of various effect and, thus, remain without adaptation of the inductive strengthening formula, much like the addition or deletion of parts. Changes on constraints, however, can be handled partly as we will see in the following. Basically, the changes in constraints that can be handled are those that replace constants. As explained in Chapter 3, the constraints are incorporated in the predecessor computation within the weakest precondition computation. To this end, they may be combined with other constraints (for example in the All-Pairs-Shortest-Paths algorithm for the backwards computation of the predecessor zone) or be combined with clock resets or assignments (for integer variables). As a result, the constraints may occur modified or unmodified in the inductive strengthening, or don't occur at all. Hence, the chances for a reasonable adaptation of the formula can be very diverse.

Best Case: Whenever the reconfigured constraint can be uniquely related to parts of the formula, an adaptation is easy, e.g., whenever all constraints in the model are unique and occur unmodified in the formula. For illustration, again consider Example 6.1.1. All constraints in the model are changed and the adaptation of the formula can, thus, be easily performed by replacing all constants in the formula. Since all constraints are changed, it does not matter whether the literal $(c_0^1 \geq 1024)$ refers the invariant constraint of location l_1 or the guard constraint of the edge between locations l_1 and l_2.

Problematic Mapping: The lack of said knowledge, however, results in the question which parts of the formula should be replaced. Since no reliable information is given

about the mapping of constraints in the model to literals in the formula, one can only guess. We illustrate this problem in the following.

Example 6.1.4. Consider again the original Fischer model (Figure 3.1) and a reconfiguration that changes only one of the time constants 1024 to 2048, namely the one in the guard constraint of the edge between locations l_1 and l_2 in the first timed automaton. The resulting reconfigured model is depicted in Figure 6.2. The safety property still holds true based on the invariant constraint in location l_1.

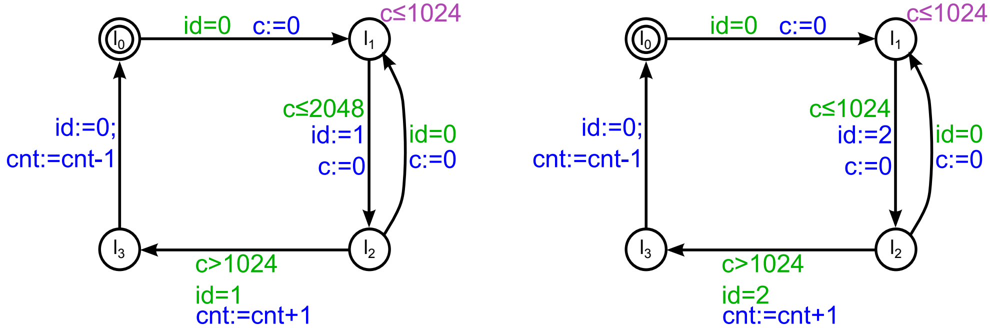


Figure 6.2: Reconfigured Fischer model with a single time constant 1024 changed to 2048

Trying to adapt the inductive strengthening according to this change of the single constant is infeasible. The reason is that it is impractical to identify which literals will be affected by the reconfiguration since we can not precisely relate a clause to a specific state.

We overcome this deficit by a best-guess use of the *Feedback*-loop. Since we do not know which clauses are affected, we duplicate all those clauses in which we find the altered constants directly used. Each duplicated clause is adapted according to the modification. As a result, the adapted inductive strengthening formula includes the adapted and also the original clauses. All these clauses are used in one of the presented *Feedback*-techniques, relying on IC3 and the *Feedback*-techniques to filter out non-relevant clauses. We will examine its success in the experiments section.

However, there exist instances, where this best-guess approach does not work. Whenever the constraints occur in modified form, i.e., combined with other constraints or assignments, the above adaptation does not work, as we can not find literals that include the same constant.

Worst Case: Constraints that occur modified within the formula can not be adapted. We illustrate this problem with the following example.

Example 6.1.5. Consider the Lemgo model depicted in Figure 3.12. The inductive strengthening of the safety property found during a run of our algorithm *IC3 with Zones* is shown in the first column of Table 6.4.

Clearly, the time constant 8001 is used directly in the formula, as well as in combination with constant 4000 in the literal $(c_0^2 - c_0^1) > (-4001.0)$, which uses

Lemgo	*Lemgo*(10000)
$((c_0^2 - c_0^1) > (-4001.0))$	$((c_0^2 - c_0^1) > (-6000.0)))$
$\wedge(l_0^1 \vee l_1^1 \vee \neg l_0^2 \vee l_1^2)$	$\wedge(l_0^1 \vee l_1^1 \vee \neg l_0^2 \vee l_1^2)$
$\wedge(l_0^1 \vee l_1^1 \vee l_0^2 \vee \neg l_1^2)$	$\wedge(l_0^1 \vee l_1^1 \vee l_0^2 \vee \neg l_1^2)$
$\wedge(l_1^1 \vee \neg l_0^1 \vee \neg l_1^2 \vee l_0^2)$	$\wedge(l_1^1 \vee \neg l_0^1 \vee \neg l_1^2 \vee l_0^2)$
$\wedge((c_0^2 > 1000.0) \vee (c_0^1 < 4000.0))$	$\wedge((c_0^2 > 1000.0) \vee (c_0^1 < 4000.0))$
$\wedge(l_1^1 \vee (c_0^2 > 1000.0) \vee (c_0^1 < 2000.0))$	$\wedge(l_1^1 \vee (c_0^2 > 1000.0) \vee (c_0^1 < 2000.0))$
$\wedge(l_1^2 \vee (c_0^2 > 4000.0) \vee (c_0^1 < 8001.0))$	$\wedge(l_1^2 \vee (c_0^2 > 4000.0) \vee (c_0^1 < 10000.0))$
$\wedge(\neg l_1^1 \vee l_1^2 \vee (c_0^2 > 4000.0) \vee (c_0^1 < 3000.0))$	$\wedge(\neg l_1^1 \vee l_1^2 \vee (c_0^2 > 4000.0) \vee (c_0^1 < 3000.0))$
$\wedge(\neg l_0^2 \vee \neg l_0^1 \vee l_1^2 \vee (c_0^1 > 3000.0) \vee (c_0^2 < 4000.0))$	
	$\wedge(\neg l_0^1 \vee l_0^2)$
	$\wedge(\neg l_0^1 \vee \neg l_1^2)$
	$\wedge(\neg l_0^1 \vee (c_0^2 < 4000.0))$
	$\wedge(\neg l_0^2 \vee (c_0^1 < 10000.0))$
	$\wedge(\neg l_0^1 \vee l_1^2 \vee (c_0^2 < 4000.0))$
	$\wedge(\neg l_0^1 \vee \neg l_0^2 \vee (c_0^2 < 4000.0))$

Table 6.4: Inductive strengthenings computed for the *Lemgo* model and the reconfigured *Lemgo*(10000) model as depicted in Figure 6.3: Clauses occurring in both formulae (with resp. constants) are displayed at the same row

the combined constant -4001 $(4000 - 8001)$. The adaption of the formula would be feasible in case these combinations of constants were known. However the computation or a logging of the combinations is infeasible, when considering large models. Taking into account the All-Pairs-Shortest-Paths algorithm responsible for the combinations of the constraints, a used constant can theoretically be composed of $n - 1$ time constants for models with n clocks. Similar arguments hold true for constants within integer constraints as they can be composed using several constants from assignments.

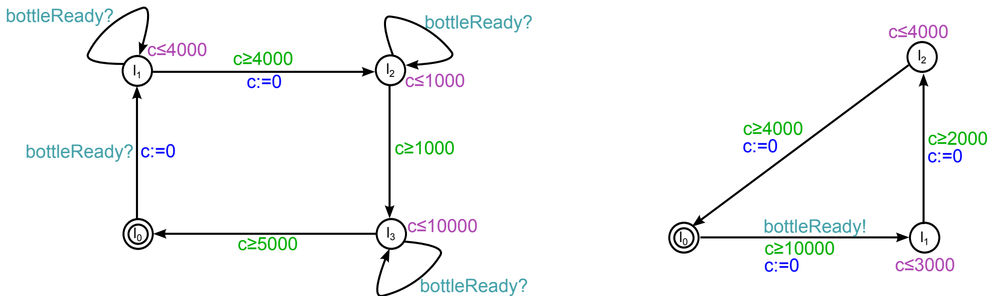


Figure 6.3: Reconfigured model *Lemgo*(10000) of the interaction between a conveyor belt (right) and a picker arm (left)

Consider the reconfiguration of the Lemgo model that replaces the constant 8001 by 10000. It might specify a change in the smart factory that the conveyor belt runs in an energy saving mode and is, thus, slower. As explained before, the inductive strengthening of the safety property for the original model does also include the time constant 8001 indirectly. Thus, the formula can hardly be adapted entirely in order to reflect the reconfiguration.

The second column of Table 6.4 shows the inductive strengthening computed for the reconfigured model. A comparison with the original formula shows the distinctiveness of the composed constant in the first clause. In addition, it can be seen that the rest of the formula is also different due to diverging runs of the algorithm.

Counterexample Trace: Finally, we have to take into account that reconfigurations are able to alter the state space, s.t. a previously invariant safety property does not hold true any longer. In such cases, the verification using *IC3 with Zones* will find a counterexample trace. As in the other examples without a counterexample trace, the injected clauses are reused to build the internal frame structure of the IC3 algorithm, ultimately leading to the discovery of the counterexample trace. In the following experiments section, we will examine whether the use of *Feedback*-techniques will be of significant help to find the counterexample trace.

6.2 Approach in Summary

Taking into account the diversity of possible reconfigurations and the impossibility to estimate their effects on the state space, we propose the following *Best-Guess* approach.

1. Store the inductive strengthening computed for the original model.
2. Delete all parts in the formula referring to deleted parts (locations, clocks or integer variables) that no longer exist in the reconfigured model by
   - Deletion of all clauses containing literals that refer the respective parts, or
   - Deletion of the disjuncts in the clauses that refer the respective parts.
3. For all constants changed during the reconfiguration, copy the clauses containing the constant and replace it in the copied clauses.
4. Use one of the three *Feedback*-techniques proposed in Section 4.6 to inject the adapted formula.

The presented approach adapts the formula and injects it with the hope to prevent the costly rediscovery of some of the clauses. To this end, it relies on the mechanisms of the injection and the IC3 algorithm to filter out irrelevant clauses.

We test this approach in the following using several experiments.

6.3 Experiments

We start with experiments using models we already verified in Chapter 3. In these experiments, we verified the respective safety properties for models with a distinct number of timed automata in order to check the scalability of our algorithm *IC3 with Zones*. We employ the inductive strengthenings computed during these runs in the following.

6.3.1 Experiments with Addition

We start with a use-case that does not require adaptation of the found inductive invariant. We employ the models of the *Carrier Sense Multiple Access with Collision Detection Protocol* and the *Token Ring FDDI Protocol*. We assume the following scenario. A model was created that represents n stations using the protocol. The safety property given in Chapter 3 was successfully verified for this model and the computed inductive strengthening has been stored. At some point, it was decided that the number of stations it too small and, in consequence, the model is reconfigured to include $n+1$ stations. Since no part of the model is deleted and no constant has been changed, the inductive invariant remains unadapted. We inject its clauses into the verification run for the reconfigured model with $n+1$ timed automata.

In our experiments, we used all three *Feedback*-techniques ($Chockler$, $Frame1$ and their combination). In general, all of them are of value for speeding up the verification for the reconfigured model. The $Frame1$-technique showed the best overall performance. Figure 6.4 depicts the reduction of runtime achieved in percent of the time needed for the original verification from scratch in the $CSMA/CD$ examples (Subsection 3.5.4). Figure 6.5 shows the respective reduction for the $FDDI$ models. As can be seen, our technique works well, in particular for the larger models. It is capable of reducing the runtime by up to 99,5%, or in other words achieves a speedup of up to 200 times.

6.3.2 Experiments with Deletion

In the following, we examine the utility of the two approaches for adaptation of the formula that are used whenever parts of the model are deleted during the reconfiguration. Again, we employ the $CSMA/CD$ and $FDDI$ models. The scenario is as follows. A model was created that represents n stations using the protocol. The safety property given in Chapter 3 was successfully verified for this model and the computed inductive strengthening has been stored. At some point, it was decided that the number of stations is too large and, in consequence, the model is reconfigured to include $n-1$ stations. As before, we employed all three *Feedback*-techniques. We executed the experiments for both options of adaptation, namely the deletion of entire clauses or of literals. In general, all *Feedback*-techniques were of value in both options. The $Frame1$-Feedback showed the best overall performance. The results for both options were very similar. In practice, we would prefer the *delete literals* option, as it preserves more clauses that could potentially be reused.

We present the reductions of the runtime achieved using the $Frame1$-Feedback with this option in Figure 6.6 ($CSMA/CD$) and Figure 6.7 ($FDDI$). The concept is clearly of help, especially for the large models. We achieved a reduction of runtime by up to 99,7%, or in other words a speedup of up to 330 times.

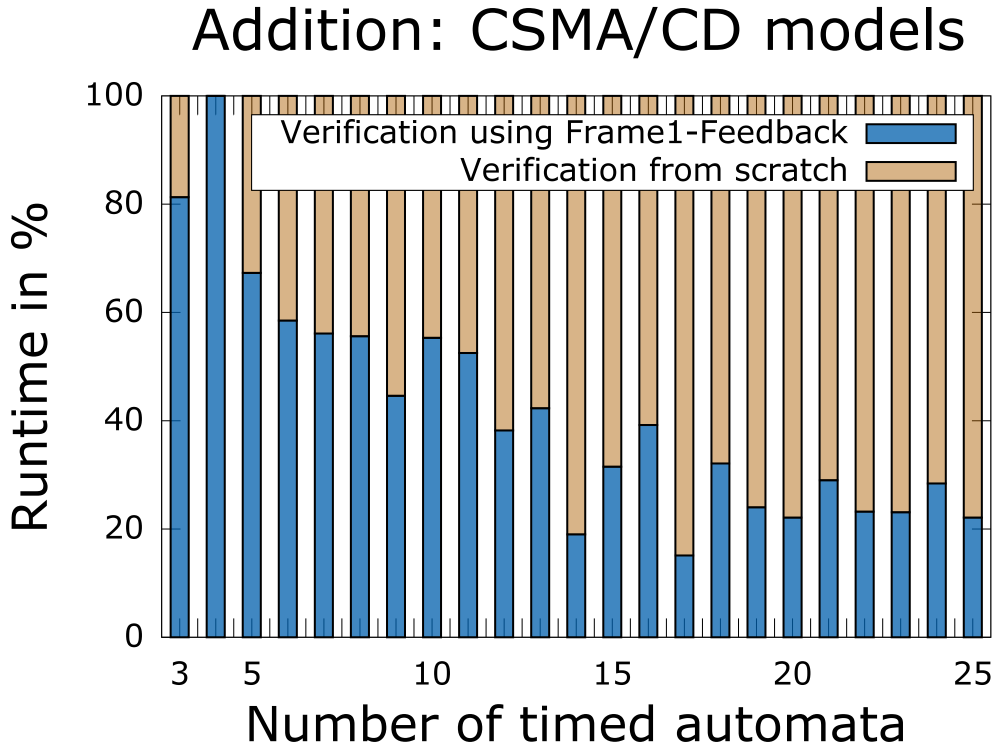


Figure 6.4: Reduction of runtime achieved with our presented approach when using the inductive strengthening computed for NTA_n in the verification for a reconfigured model NTA_{n+1} of the $CSMA/CD$-protocol

6.3.3 Experiments with Large Models

The speedup achieved in the previous experiments can be employed to verify safety properties for models for which verification previously failed. The largest $CSMA/CD$ and $FDDI$ models for which we could verify the safety property previously are $CSMA/CD$_25 and $FDDI$_18 that include 25 and 18 timed automata modeling stations, respectively. We apply a cascading approach in order to verify larger models. To this end, we reuse the inductive strengthenings computed for $CSMA/CD$_25 and $FDDI$_18 in the verification runs for models $CSMA/CD$_26 and $FDDI$_19. The inductive strengthenings computed during these verifications are then injected into the runs for models $CSMA/CD$_27 and $FDDI$_20 and so on.

Using this cascading technique, we were able to verify the models $CSMA/CD$_28 and $FDDI$_39. These successful verifications are a significant increase. The usage of our technique in such an cascading way might be useful for many models, for which the safety property could not be verified previously.

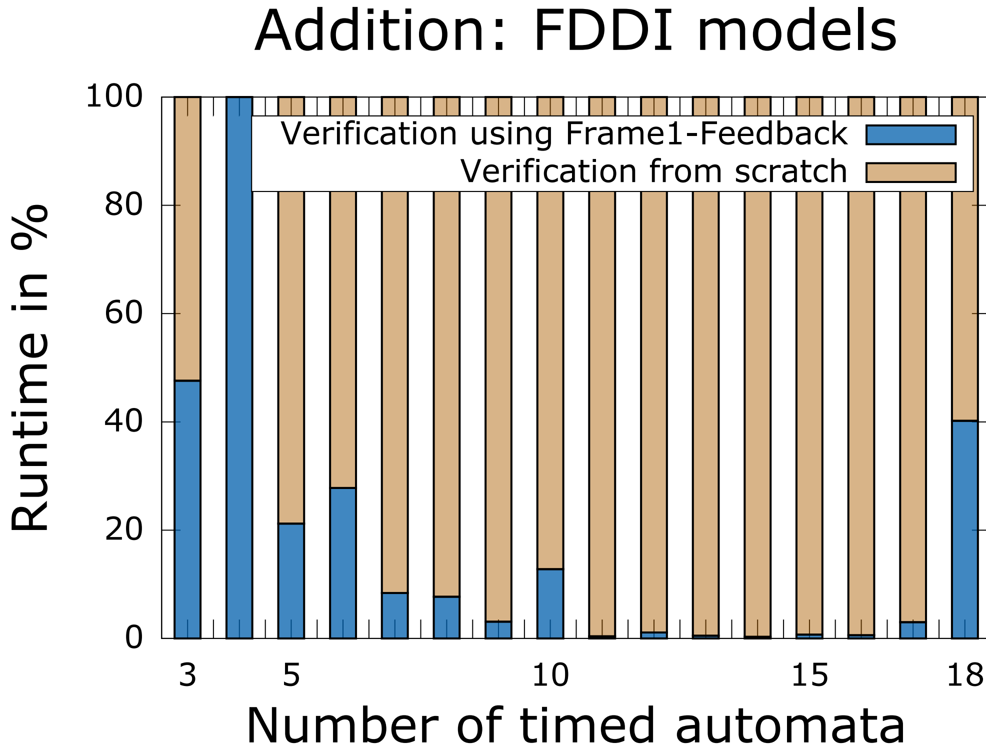


Figure 6.5: Reduction of runtime achieved with our presented approach when using the inductive strengthening computed for NTA_n in the verification for a reconfigured model NTA_{n+1} of the $FDDI$-protocol

6.3.4 Experiments with Constants

In the following, we examine the value of the adaptation that takes place when a constant is changed. To this end, we employ the *Fischer_U_15* model that has been adapted as presented in Figure 6.1. The significant value of our adaptation for this example has been illustrated in Tables 6.2 and 6.3.

In addition, we employ a reconfigured *Fischer_U_15* model in which only a single constant is changed (Figure 6.2). Table 6.5 compares the runtimes of the verification using the original and the adapted inductive strengthening with the three distinct *Feedback*-techniques. Note, that the verification of the reconfigured model from scratch (without the use of the *Feedback*-techniques) needed 139,9 seconds. All of the runtimes using the *Feedback*-techniques are significantly lower, which speaks in favor of our method.

The model is reconfigured only slightly, which might be the reason, why the adaptation does not work as well in this example, as in the one where all constants were changed (*Fischer_U_15*(2048)). The *Feedback*-techniques perform worse using the adapted inductive invariant. Yet, they significantly outperform a verification from scratch.

As we can see, an adaptation of the formula is not strictly necessary. In fact, in

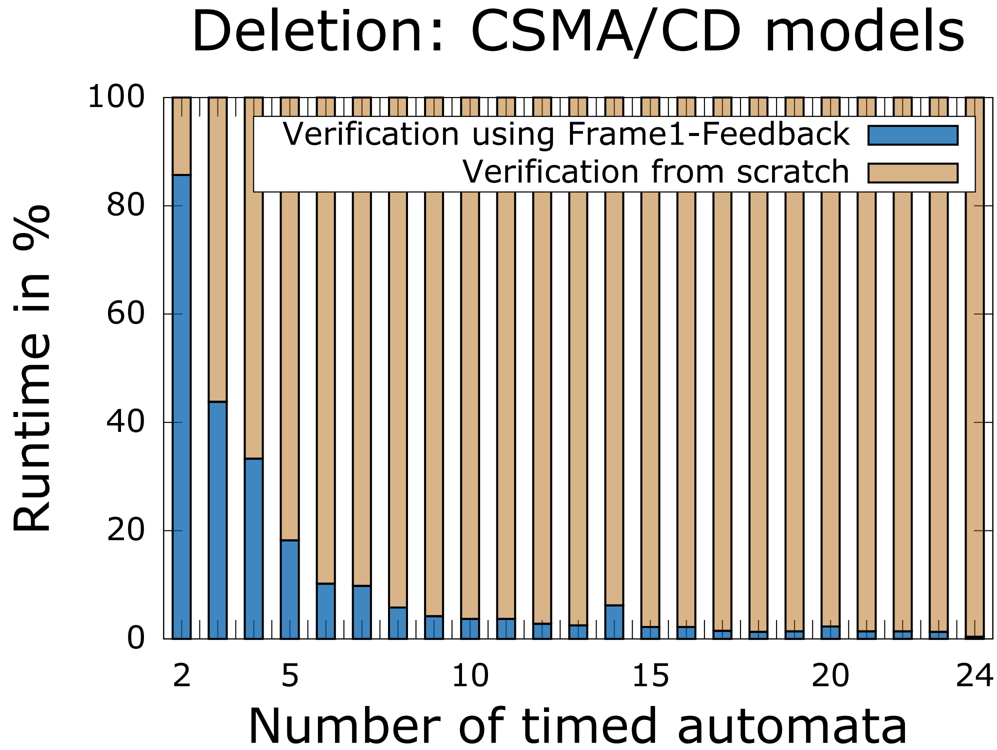


Figure 6.6: Reduction of runtime achieved with our presented approach when using the inductive strengthening computed for NTA_{n+1} (with deleted literals) in the verification for a reconfigured model NTA_n of the $CSMA/CD$-protocol

	Runtime (s) using adapted inductive invariant	Runtime (s) using original inductive invariant
Verification reconfigured model with *Chockler*-Feedback	14,1	7,3
Verification reconfigured model with *Frame*1-Feedback	9,6	5,3
Verification reconfigured model with *Both*-Feedback	14,4	7,5

Table 6.5: Comparison of runtimes of our approach with and without adaptation of the inductive strengthening for the *Fischer_U_15* model reconfigured only by the change of a single time constant in the first automaton as depicted in Figure 6.2

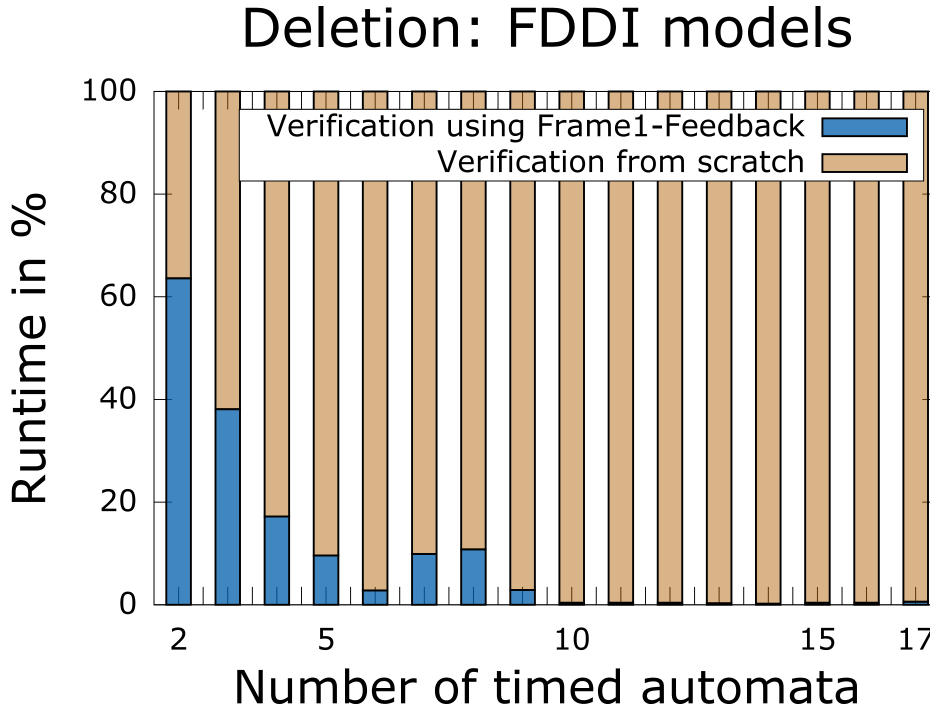


Figure 6.7: Reduction of runtime achieved with our presented approach when using the inductive strengthening computed for NTA_{n+1} (with deleted literals) in the verification for a reconfigured model NTA_n of the $FDDI$-protocol

this example the *Feedback*-techniques work worse with an adapted formula than with the original one. This setback might be due the overhead of adaptation and, consequently, having more clauses to be injected. Furthermore, the considered reconfiguration is very small and, thus, the original inductive strengthening might be closer to a valid inductive invariant for the reconfigured model. Nevertheless, for other models and reconfigurations the adaptation procedure might be of significant value as can be seen in the experiment with the $Fischer_U_15(2048)$ model (Tables 6.2 and 6.3). The difference between both *Fischer* examples is the level of change. With increased level of change in a reconfiguration, the need for adaptation of the formula is increased, too. Thus, for small reconfigurations, an adaptation might not be necessary. For large reconfiguration, however, it seems to be of benefit.

We have executed an additional experiment to check the effects on smaller models. Table 6.6 shows the same comparison as Table 6.5 for the reconfigured model $Lemgo(10000)$ (Figure 6.3). The runtimes are averages over 100 runs due to imprecisions in the small runtime.

Clearly, the runtimes differ not that much for smaller models. Thus, the effect of an adaptation of the formula is hardly visible. Yet, the improvement over a verification *from scratch*, needing 0,83 seconds in average, is visible. Thus, our

technique can be of value even for such small models. The best acceleration, however, is achieved with large models, as the overhead of preprocessing the model and the injection of the formula is negligible there.

In summary, we can state that the technique presented in this chapter is of value for the acceleration of verifications for reconfigured models. There exist models and reconfigurations, where the technique works very well and others, where it does not. When considering models with small reconfigurations, an adaptation of the formula might be too much overhead. In contrast, when the reconfiguration introduces a large change to the state space, then an adaptation of the formula is strongly encouraged.

6.3.5 Counterexample Experiment

In this last experiment, we examine the utility of our approach in matters of counterexamples. To this end, we again consider a reconfiguration to the *Lemgo* model. In this setting, the time constant 8001 is changed to 7000. The reconfigured model is shown in Figure 6.8. The reconfiguration is small as only a single time constant has

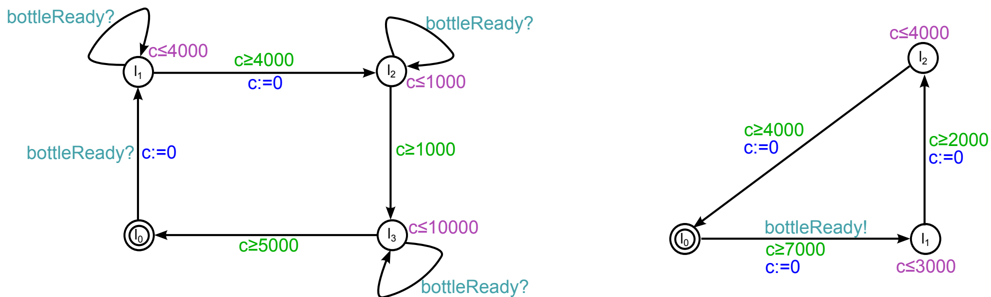


Figure 6.8: Reconfigured model *Lemgo*(7000) of the interaction between a conveyor belt (right) and a picker arm (left)

been modified. Its effect, however, is huge considering the given safety property. The reconfiguration introduces the reachability of an error state, i.e., a state that violates the safety property. Our technique will, thus, extract a counterexample trace.

	Runtime (s) using adapted inductive invariant	Runtime (s) using original inductive invariant
Verification reconfigured model with *Chockler*-Feedback	0,65	0,66
Verification reconfigured model with *Frame*1-Feedback	0,71	0,71
Verification reconfigured model with *Both*-Feedback	0,66	0,67

Table 6.6: Comparison of runtimes of our approach with and without adaptation of the inductive strengthening for the *Lemgo*(10000) model

After the reconfiguration, the conveyor belt can send the signal *bottleReady!* after 13000 time units. The picking arm, however, still requires up to 14000 time units to be ready to reasonably process the signal (in location l_0). Thus, after a first cycle of both timed automata, when 13000 time units have passed, the signal may be send allowing the conveyor belt to reach location l_1 for the second time. When receiving the signal, the picking arm may not be in location l_0, but it may subsequently switch over to that location, resulting in a violation to the safety property $\rho := G\neg((l_0, l_1), true, true)$. This sequence models a situation where the picking arm has missed a bottle.

As can be seen, this small reconfiguration provides for a counterexample trace violating the property, which previously held true in the unchanged model. Each reconfiguration alters the state space and without an actual verification it is hard to determine the effects.

We have examined the runtime when injecting the adopted inductive invariant into the verification run for the reconfigured model. As can be seen, it is only of small help. Table 6.7 shows the results, which are averages over 100 runs due to imprecisions in the small runtime.

	Runtime (s)	Memory (MB)
Verification without *Feedback*-technique (from scratch)	0,68	62,81
Verification reconfigured model with *Chockler*-technique	0,64	63,51
Verification reconfigured model with *Frame*1-technique	0,65	69,61
Verification reconfigured model with *Both*-technique	0,65	65,60

Table 6.7: Runtimes of the verification for the reconfigured model $Lemgo(7000)$ that includes a reachable error state violating the safety property

Yet, this example shows that our technique can also be applied when the reconfiguration introduces a violation of the safety property.

In general, the utility of our approach is very much dependent on the employed models, reconfigurations and safety properties. Nevertheless, our experiments show that it is a promising option to accelerate verifications for reconfigured models.

6.4 Related Work

To the best of our knowledge, there does not exist related work that is directly concerned with the acceleration of verification runs for reconfigured models in the timed setting.

In the untimed context, the task is often denoted as regression verification or incremental verification. The work closest to ours is the one by Chockler et al. [Cho+11] employed in Chapters 3 to 6. It uses an inductive invariant computed by IC3 for the acceleration of a verification for a reconfigured model. To this end, it computes

the maximum set of clauses of the invariant that is inductive in the changed model. Our *Frame*1 method employs the same ideas, but instead of using an inductive set of clauses, we compute a maximum set of clauses that are inductive relative to the initial states of the model. Our technique allows to reuse more clauses, as the condition is weaker. However, these clauses can not be injected into every frame of the algorithm as is the case in Chockler's approach. IC3's propagation mechanism compensates for this drawback. Both techniques can also be employed in combination. In addition, a distinctive characteristic of our approach is the added adaptation of the employed inductive invariant.

Other work employing inductive invariants for the accelerated verification of changed models was done by Cabodi [Cab+09; CNQ09]. It is concerned with the computation of inductive invariants as constraints that are hidden in the design or property. They are used in non-inductive model checking procedures in different ways, e.g., via constraining the transition relation.

In general, there exist many works in the context of incremental verification or regression verification. They can roughly be divided into two categories as they employ distinct strategies, namely to analyze the changes to the model or to reuse intermediate results. Either way, their aim is to speed up the verification of the reconfigured model.

Within the first category there are works that restrict the part of the state space considered in the new verification, e.g., Böhme et al. [BOR13] use partitions. Backes et al. [Bac+13] use a combination of symbolic execution and static analysis to compute the partition of the state space that is impacted by the reconfiguration. There exist several other approaches, e.g., the reduction of the equivalence of programs to horn constraints, which are afterwards checked by an SMT-solver [Fel+14]. Godlin et al. [GS13] base their approach for programs on a representation of the procedures by uninterpreted functions or give an incremental algorithm for fixpoint computation that takes into account the changes done to the model [SS94].

The works in the second category all employ the strategy of reusing intermediate result from the previous verification. Henzinger et al. [Hen+03] reuse a previously computed abstract reachability tree and check conformance with the control flow automaton of the changed program. Other employed artifacts are derivation graphs [Con+05] or hash values for computed reductions [KM89]. Sery et al. [SFS12] employ function summaries that are over-approximations computed as Craig Interpolants. Other work by Visser et al. [VGD12] stores and reuses reduced constraints to avoid solver calls in a symbolic analysis. Using abstract analysis, Beyer et al. [Bey+13] store the precision of a previous verification for reuse.

Our work definitely lies within the second category, as it reuses previously computed inductive invariants. In addition, it analyses the reconfiguration in order to adapt this artifact. It, thus, also partly falls into the other category.

6.5 Summary

In summary, we have examined the application and potential benefit of the *Feedback*-technique for the acceleration of verifications for reconfigured models. We have shown the diversity of reconfigurations and illustrated their various effects, which can hardly be anticipated. In consequence, we have given a best-guess approach to adapt an inductive strengthening computed for an original model in order to optimize the injection of this formula into the verification run of the reconfigured model. We have conducted numerous experiments that indicate the practicality within certain bounds, since the approach works well for reconfigured models that do not differ too much from their original ones.

From a broader perspective, we conclude that the presented method is a good starting point, when dealing with reconfigured systems. The method reduces the effort during online verification that is generated by models being changed during their lifetime. It can, furthermore, also be applied for reconfigurations during the design process. It it of practical value, even if the benefit of it might be small for some instances.

7

Conclusion

This thesis addressed the formal verification guaranteeing the absence of erroneous behavior in real-time systems.

As explained in the first chapter, an increasing number of today's systems is timing-based, e.g., relies on real-time communication or operating systems. In addition, many of these systems are safety critical such that their faulty behavior results in danger to life or production value. In order to avoid these hazards, model-based design processes are employed, in which the systems can be checked for erroneous behavior. To this end, models of the systems are build during the design phase. These models represent the behavior of the systems and can be checked for reachability of erroneous states. In this thesis, we employed the modeling formalism of networks of timed automata, as it is one of the most common real-time formalisms. The undesired error states are specified as safety properties, which express that an error state should not be reachable.

Within the last 25 years, an enormous amount of research has been carried out in order to verify safety properties for timed automata. There are sophisticated tools available that are extremely efficient in answering such verification questions. Yet, there exist shortcomings in these approaches, most notably the enormous need for memory. As a result, there are instances in which these tools run easily out of memory. This issue affects mostly large models that include many timed automata. Unfortunately, the systems that require formal real-time verification are increasingly complex and progressively interconnected. Their respective models will, thus, bring up the above issue.

In this thesis, we have proposed a novel combination of techniques in order to approach the presented problem. To this end, we combined the *IC3* algorithm [Bra11] for the verification of safety properties designed for the hardware domain with the *Zone abstraction* designed for real-time formalisms. Both are sophisticated techniques with important characteristics, which we tried to utilize in their combination. The former has proven to be very efficient on finite transition systems, both regarding

runtime and memory requirements. The latter has successfully been employed to abstract the infinite transition system introduced by the real-time clocks. Chapter 2 presented these foundations to our work, as well as other related work and the employed formalism of networks of timed automata. In the subsequent Chapter 3, we presented the combination detailed above. It includes an encoding of the model as *Satisfiability Modulo Theories (SMT)*-formulae and, in particular, the integration of the *Zone abstraction* into the *IC3 algorithm*. To this end, we employ weakest predecessor computation in an additional step that abstracts the concrete states found via the issued SMT-queries. We have implemented and tested our proposed approach *IC3 with Zones* in numerous experiments. The results indicate a good scalability. Our implementation was not capable to outperform the established tools in all experiments, but for larger models it showed a promising performance. We were able to verify several models that the state-of-the-art tool Uppaal could not verify due to running out of memory. Additional experiments present the benefit of our encoding and show that our technique is a significant improvement over a previous approach using the region abstraction [KJN12b].

In Chapter 4, we employed our presented approach *IC3 with Zones* in an incremental workflow, which enables the verification of even larger models. These models are special in that they incorporate a notion of symmetry, i.e., all timed automata in a model are symmetric. This speciality in the structure of the models is the basis for our incremental workflow, which is capable of verifying the models for any fixed, but arbitrary large number of symmetric timed automata in the model, denoted a *Parameterized Timed System*. We have shown several mutual exclusion algorithms, which lie within this restricted formalism of symmetric models. Our workflow starts with the verification of the symmetric safety property for the smallest model using our *IC3 with Zones* technique. The outcome of the successful verification is an inductive strengthening of the safety property, which is reused to reason about all larger models. To this end, we have proven a *Termination Theorem* that allows us to draw conclusions about all larger models in case its premise is fulfilled. Otherwise, the incremental workflow starts all over with the verification of the next larger model. We have proposed two optimizations to this approach that alter and reuse the inductive strengthening and tested their utility in several experiments. The results of these experiments confirm the overall value of our approach, in particular as it verifies the safety property for the smaller models even if the Termination Theorem is not applicable. The approach is, thus, a definite improvement to the verification of the single models and it allows for an efficient verification of parameterized timed systems, when using the optimizations.

Chapter 5 extended the above technique to models that include synchronization and a fixed number of extra timed automata accompanying the parameterized symmetric automata. It is most suitable to model client-server scenarios or peer to peer protocols, that require the communication medium to be modeled as well. We have proven that our Termination Theorem is still valid with the extension and performed two experiments to show the practicality. As in the previous chapter, the

approach works best using the proposed optimizations and is of significant value in direct comparison with the verification of safety properties for fixed models.

One optimization presented in Chapter 4 was a *Feedback*-loop that injects the inductive strengthening in a verification run of a reconfigured, i.e., changed model. To this end, we employed a technique presented in 2011 [Cho+11], and proposed an additional approach. In Chapter 6, we examined the utility of this *Feedback*-technique for reconfigurations in general, not just the addition of a symmetric timed automaton as before. We explicated the distinct ways of reconfiguring models and the chances to adapt the inductive strengthening accordingly. As a result, we gave a sequence of steps that allow the usage of such a *Feedback*-loop in this general setting. Our experiments indicate a benefit for some models, but also point out definite limits. All things considered, the general application of a *Feedback*-loop can be considered to be a best-guess approach that might be of help.

In the following, we will discuss the decisions and their effects made during this thesis.

7.1 Discussion

During the research and writing of this thesis several decisions were made that influenced the course and characteristics of the presented work. The following subsection explains these decisions and their reasons and effects. Afterwards, we list the restrictions of our approaches, before concluding with an enumeration of the strengths and weaknesses.

7.1.1 Design Decisions

One of our most important design decisions was made concerning the modeling formalism as detailed below.

Modeling Formalism Our work aims to improve the formal verification of safety properties in real-time formalisms. We employ the formalism of timed automata as it is well known and understood. Yet, there exists many, sometimes significantly distinct, formalizations of timed automata. These offer different modeling capabilities and drawbacks.

The formalization used in this thesis is close to the one employed in the most well-known tool for timed automata, namely Uppaal. This decision enables a fast and easy understanding for those readers that are familiar with Uppaal. One of the main differences to other formalizations used in the community is the use of synchronization. Uppaal employs a synchronization feature in a CCS-like fashion, where there exist a sender that synchronizes with one (handshake synchronization) or many (broadcast synchronization) receivers. Unlike Uppaal, we do not offer the latter option. Our formalization does not allow broadcast synchronization. The reason for this design decision is the SMT-encoding.

SMT-Encoding The process of encoding the network of timed automata as an SMT-formula is of vital importance for the performance of all presented techniques in this thesis. In Section 3.1, we have presented the encoding that is used throughout the rest of our work. It includes the encoding of all combinations of synchronized edges, which are computed upfront. This procedure is hardly feasible when encoding broadcast synchronizations as allowed by Uppaal. For each sender edge, each possible combination of receiver edges has to be considered, i.e., with 0 receivers enables, 1 receiver enabled and so on. The resulting encoding would easily grow too large to be handled efficiently by the solver. As a result, we abstained from including broadcast synchronization into our formalization. Instead, we consider it as potential future work to find an improved encoding that enables the use of broadcast synchronization.

The second design decision concerning the encoding is the handling of locations. As explained in the mentioned section, we encode a location using a unique identifier, whose boolean representation is encoded using boolean variables. This procedure has its benefit, as well as a drawback. We illustrated the impact on the generalization procedure of the IC3 algorithm in the experiments section of Chapter 3. Since each location of each automaton is encoded using several variables, the generalization procedure enables the reasoning about several locations of a single automaton at once. This ability has been a huge benefit when verifying the mutual exclusion property of the *FDDIcount* model, which reasons about all locations of the first automaton with an even location identifier. The drawback of the presented location encoding is a significantly increased number of variables. As a result, the SMT-solver might require more time to answer the queries and the generalization procedure of IC3 has to test an increased set of literals for removal, which also needs additional time.

As closing design decision, one can also discuss the usage of the IC3 algorithm.

IC3 Algorithm The IC3 algorithm [Bra11] is the basis for our work. Its advantages are numerous, from its efficiency in time and memory requirements to its output of an inductive invariant. It has enabled the work as presented in this thesis, in which it might be replaced only in (part of) the technique presented in Chapter 4 by another algorithm that computes an inductive invariant. However, despite all these positive aspects, IC3 also has its drawbacks as shown in the experiment section of Chapter 3. Using the SMT-encoding, it always considers the entire state space and not just the reachable part, as other techniques do. Thus, the usage of the IC3 algorithm has its benefits and drawbacks, but we believe the advantages outweigh the disadvantages by far.

In summary, the design decisions we made are of various importance and effect. In the following, we will discuss the implied application scope and restrictions.

7.1.2 Application Scope and Restrictions

During the presentations and illustrations of the presented techniques in Chapters 3 to 6, we discussed their applicability and some restrictions. We will briefly recall and comment on these details below.

IC3 with Zones - Technique Our approach combining the IC3 algorithm with the Zone abstraction has a broad scope of application. As explained above, the employed formalism is well known for the modeling of timed systems. We use an expressive formalization similar to the one employed in Uppaal, which is sufficient for the modeling of various systems. However, there exist certain restrictions.

Based on our used SMT-encoding, we disallow broadcast communication as explained before. Furthermore, we did not include boolean variables in our formalism, as they can easily be mimicked using integer variables. Thus, their integration into our formalism would not add expressiveness. It might, however, result in a faster runtime to model boolean variables directly.

Other formalisms also allow for procedures being executed when taking an edge in an automaton. We did not include such capabilities. They would require a suitable encoding, which might be a complex task dependent on the structure of the procedure. An additional obstacle is the weakest predecessor computation, which must be possible for the considered kind of procedure.

In summary, we conclude that the application scope of our first technique is rather general, but there exist restrictions that slightly limit the expressiveness. These restrictions also exist within our other techniques presented in this thesis.

Parameterized Timed Systems - Technique Several additional restrictions apply for the technique presented in Chapters 4 and 5. Though we relax some of them in the latter chapter, the application scope is limited.

The main limitation is the restriction to a single template that models a single parameterized class of processes. As a reason, consider the structure of our incremental workflow underlying the technique. Whenever the verification result for a model with fixed number of processes could not be used to reason about all larger models, we restart the main cycle in the workflow with the next larger model. When dealing with more than one parameterized class, this concept can not be applied unchanged, as it is unclear which of the parameters should be increased.

Additional restrictions, in particular those on the integer variables, are introduced to guarantee symmetry of the state space and, thus, ensure the correctness of our Termination Theorem. In consequence, their relaxation is not desirable.

General Reconfigurations - Technique Considering the last technique proposing the use of the *Feedback*-loop for general reconfigurations (Chapter 6), we conclude that the same application scope and restrictions apply as for the algorithm *IC3 with Zones*.

We complete this section with the summary of strengths and weaknesses of the approaches.

7.1.3 Strengths and Weaknesses

The techniques presented in this thesis heavily profit from the efficiency of the IC3 algorithm. Our *IC3 with Zones* approach has shown to verify safety properties for models that other state-of-the-art tools could not verify. In particular, the verification in combination with a large reachable state space is promising using our technique.

Furthermore, the algorithm can easily be parallelized in order to gain additional efficiency. The states that need blocking in the algorithm can be queried in parallel along with their generalization into a blocking clause. What must be ensured it that the same state would not be handled twice and that the blocking clauses are always up to date.

As explained above, an additional strength of our work is the similarity of our formalism to the one used in Uppaal, which allows an easy use for people that are familiar with Uppaal.

Our *IC3 with Zones* approach guarantees termination (under certain restrictions on the integer variables) and its output is perfectly suitable to be reused. The application for this reuse ranges from simple certification scenarios, where a successful verification has to be proven, to more complex scenarios as detailed in this thesis.

We reuse the inductive invariant computed by our technique to speed up subsequent verifications for reconfigured models. These reuse techniques are devised, s.t. the termination guarantee of our approach still holds true.

In addition, we have applied the reuse capabilities in a setting that allows the reasoning about an entire parameterized timed system on the basis of verifications of single models from the system. A positive advantage of our concept is that the smaller models have been verified, even if the technique was not capable of reasoning about the entire system, which might happen due to undecidability.

However, the presented techniques do have some deficiencies. Considering the experiments done in Chapter 3, we conclude that there exist models for which our technique works worse than other state-of-the-art-tools. One reason, in particular when considering small models, is the overhead of SMT-encoding and -solving inherent in our approach. An additional reason is the internal functioning of IC3, which does not only consider the reachable portion of the state space, but the entire state space encoded in the SMT-formulae. As a result, our technique works poorly on models with a large state space of which only a tiny portion is reachable. We have shown and explained this effect in the experiments of Chapter 3.

These performance issues of the *IC3 with Zones* approach also influence the performance of the other two presented approaches. In addition, the technique proposed in Chapters 4 and 5 are susceptible to a large number of clauses including many distinct automata, as all permutations have to be computed in the extrapolation procedure.

Taking into account all these strengths and weaknesses, we conclude that our techniques work well in many cases, but also fail in some others. These deficiencies can be regarded as starting point for future work.

7.2 Future Work

In the following, we list the starting points for future work. We list them separately for the distinct techniques presented in this thesis.

7.2.1 *IC3 with Zones* - Technique

There exist several options for improvements and optimizations for the *IC3 with Zones* approach. An interesting research question addresses the heuristic employed for the ordering of literals in the generalization procedure. By determining which literals might first be deleted and, in consequence, shaping the resulting blocking clause, the heuristic is of high value for the overall performance of our approach as shown above. We have shown in the experiments that none of our proposed heuristics is completely superior. However, there might be a chance to adapt the used heuristic based on the model to be verified. A course of action for this future work would be to think about which characteristics of the models can easily be determined by a static analysis and try to relate these qualities with the performance of our approach under different heuristics. Additionally, these static characteristics might be employed to decide between the presented SMT-encoding and adapted ones, e.g., one without locations being encoded via boolean variables.

Possible future work also concerns the abstract CTIs. As these are generalized with as few literals as possible, it could be worthwhile to investigate whether a reduced constraint system might be applicable. Larsen et al. [Lar+97] store zones in their work as a minimal constraint system. A course of action would be to examine whether such a minimal constraint system is sufficient for an encoding of an abstract CTI. As a consequence, the generalization procedure might be facing fewer literals and would, thus, possibly terminate faster. However, a reduced set of constraints might also be counterproductive as a missing constraint might hinder the removal of other constraints and, in the end, result in larger clauses and larger runtime.

Furthermore, there exist several additional research ideas. They vary from general ideas such as to directly encode an abstract transition system based on zones to more specialized ideas such as adapted generalization procedures that may apply widening operators to change bounds on (integer) constraints.

Furthermore, an interesting experiment might be to implement an IC3 variant that is directed the other way, i.e., it employs forward computation instead of backward computation. This might be combined with a structural analysis that decides upfront which direction might be more profitable. In addition, the substitution of some or all of the SMT-queries by actual successor/predecessor computation might be of interest. While this replacement might be counterproductive in the generalization procedure since the efficient usage of the UNSAT-core would be omitted, it might be worthwhile in other parts of the algorithm, e.g., the strengthenClauses procedure. Since the error state specifications heavily limit the set of abstract successor states, a direct computation of predecessors in the mentioned procedure might be beneficial over the SMT-query with all its overhead.

Additionally, benefit might be achieved via an extension of the formalism. When getting rid of the SMT-encoding as proposed before and doing a forward analysis, the extension of the formalism with broadcast synchronization and procedures can be easily implemented. Other extensions might be worthwhile, too.

As an idea to tackle the performance of the approach when dealing with large state spaces with only a small portion being reachable, a combination with a bounded exploration upfront might be thinkable. The problem in this scenario is the question in which way the knowledge obtained in the exploration can be injected and used in a run of the algorithm *IC3 with Zones*.

In summary, there exist many possible challenges for future work, but it is hard to estimate which ones might be profitable.

The outcome of most of the proposed future work would also affect our technique presented for the verification for parameterized timed systems. There exist, however, also ideas for future work specifically concerned with the latter technique.

7.2.2 Parameterized Timed Systems - Technique

One of the most obvious directions for future work is the extension of the formalism to include more than one template, i.e., more than one parameterized class of processes. The course of action for this idea, however, is not obvious. Our incremental workflow currently limits the formalism to only a single template. When adding an additional template, one can not simply increment both numbers of instantiations, in which case the parameterized timed system would only be verified if all classes of processes have the same number of instantiations. But when incrementing only one of the parameters, the other would remain fixed and needs to be incremented sometime later. Thus, the combination of more than one parameter introduces a lot of challenges that need to be solved.

As most promising future work we see the following. Other researchers have applied symmetry to reduce the number of states that need to be explored and stored via representatives (Symmetry Reduction), e.g., Hendriks et al. [Hen+04]. A similar technique might be applicable, where in the *IC3 with Zones* run itself, the found clauses are permuted and all permutations are conjoined to the frames. It spares the discovery and blocking of symmetric CTIs and could speed-up the algorithm significantly.

When considering the best-guess approach proposed for general reconfigurations, we have the following ideas as starting points for future work.

7.2.3 General Reconfigurations - Technique

Considering general reconfigurations of a single time constant is closely related to the problem of parameter synthesis, that is, for which interval of constants is the system safe. Our technique can easily be applied to test specific single reconfigurations. So far, however, it is not suitable to be used for a clever finding of intervals of such. An interesting research direction would, thus, be the synthesis of such intervals using our technique.

Additional future work could examine the usage of the *Feedback*-technique for frequent reconfigurations. In these cases, inductive strengthenings are computed and injected repeatedly. They could be studied to find only those clauses that are relevant in all the verification runs. Their sole usage might speed up the upcoming verification runs for the next reconfigurations as the irrelevant clauses are filtered out.

In summary, there exist many starting points for future work.
We will, finally, conclude this thesis with a small summary.

7.3 Summary

The aim of this thesis was the advancement of timed verification in the context of model-based design processes and Industry 4.0. We identified the existence of large, complex models as crucial characteristics of this setting, as well as the existence of reconfigurations (changes) to the models that require the verification to be redone in an online fashion during lifetime.

In the thesis, we have introduced a verification technique for timed systems that combines two existing concepts. Unlike other approaches, it avoids explicit exploration of the state space by utilization of induction. Due to this distinctiveness in concept, it has strengths and weaknesses different to existing approaches and is, thus, a valuable alternative and complement.

We have furthermore presented techniques that are able to cope with reconfigurations introduced during the design phase, e.g., in the model-based design process, or during the lifetime of a system, e.g., introduced as adaptation and self-optimization.

For a specific class of systems that are parameterized in the number of instantiations of the same process, we proposed a technique to enable a priori verification of the entire system, irrespective of the actual number of instances. This approach avoids the need to do online verification for a reconfigured system in which instantiations are added or deleted.

In a final step, we have applied one of the basic ideas of the previous technique for general reconfigurations. As a result, we proposed a best-guess approach that significantly accelerates the verification for some reconfigured models, which is a huge benefit in the setting of online verification.

Thus, in summary, we achieved our goals and delivered work that is a valuable complement to existing technologies and is of importance for timed verification in online scenarios.

A Experimental Results

A.1 Runtime and Memory Consumption of Experiments

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
CSMA/CD_2	0,7	69,9	0,8	67,7	0,1	35,1
CSMA/CD_3	1,6	74,9	1,5	74,9	0,1	35,1
CSMA/CD_4	2,4	69,9	3	72,8	0,1	35,2
CSMA/CD_5	5,5	84,8	5,6	83,7	0,1	35,3
CSMA/CD_6	11,8	94,7	11,7	92,7	0,1	35,3
CSMA/CD_7	16,4	100,5	16,3	93,5	0,1	35,6
CSMA/CD_8	29,3	101,1	37,3	105,8	0,3	36,3
CSMA/CD_9	42,4	109,4	49	111,4	0,8	37,8
CSMA/CD_10	78	112,5	65,8	123,3	2,4	42,2
CSMA/CD_11	116,8	141,6	120,1	146,1	6,6	53,1
CSMA/CD_12	205,4	182,4	185,2	182,2	18,1	111,1
CSMA/CD_13	220,8	183,2	267,1	202,4	48,8	228,1
CSMA/CD_14	351,9	239,4	337,5	222,7	129,4	547,0
CSMA/CD_15	406,8	251,5	323,1	230,8	333,8	1293,2
CSMA/CD_16	513,5	303	524	275	-	OOM
CSMA/CD_17	742,7	322,1	674,7	308	-	OOM
CSMA/CD_18	839,1	380,9	847,6	383,5	-	OOM
CSMA/CD_19	1110,1	469	889,1	380,1	-	OOM
CSMA/CD_20	1324,8	510,3	1204	494	-	OOM
CSMA/CD_21	1231,9	473,8	1477,5	495,6	-	OOM
CSMA/CD_22	2187,7	617,8	1536,2	570,1	-	OOM
CSMA/CD_23	2105,0	603,8	2068,9	594,6	-	OOM
CSMA/CD_24	2665,4	637,8	2665,5	638,4	-	OOM
CSMA/CD_25	3203,4	679,3	2840,7	631,8	-	OOM
CSMA/CD_26	-	OOM	-	OOM	-	OOM

Table A.1: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the Uppaal models of the CSMA/CD communication protocol (e.g. depicted in Figure 3.4)

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
Fischer_U_1	0,5	60,3	0,5	60,3	0,1	35,0
Fischer_U_2	0,9	78	0,9	80,1	0,1	35,1
Fischer_U_3	1,8	95,2	2,2	99,5	0,1	35,1
Fischer_U_4	3,5	100	4,3	108,9	0,1	35,1
Fischer_U_5	4,3	98,7	5,2	110,5	0,1	35,1
Fischer_U_6	10	147,3	9,9	137,7	0,1	35,2
Fischer_U_7	14,1	148,7	15,5	159,2	0,2	35,6
Fischer_U_8	14,1	139,3	22,3	175,1	0,7	36,7
Fischer_U_9	26,6	171,9	35,1	180,2	3,0	40,5
Fischer_U_10	34,4	175,8	33,3	180,5	12,6	53,4
Fischer_U_11	54,2	235,6	61,9	219,9	51,4	110,9
Fischer_U_12	73,2	220,5	83,1	267,4	205,3	300,7
Fischer_U_13	97,4	232,5	90,7	262,5	808,5	901,0
Fischer_U_14	104,9	253,6	144,7	287,2	-	OOM
Fischer_U_15	158,1	269,7	176,1	306,8	-	OOM
Fischer_U_16	153,9	282,6	208,4	324,9	-	OOM
Fischer_U_17	221,7	328,1	226,5	383,2	-	OOM
Fischer_U_18	290	355,6	347,6	369,8	-	OOM
Fischer_U_19	384,9	421,3	351,7	417,3	-	OOM
Fischer_U_20	432,9	382	408,7	403,6	-	OOM
Fischer_U_21	418,4	415	546,2	459,3	-	OOM
Fischer_U_22	727	497,8	615,6	454	-	OOM
Fischer_U_23	690,1	475,5	789,3	563,7	-	OOM
Fischer_U_24	714,3	490,1	929,6	559,6	-	OOM
Fischer_U_25	887,8	527,3	986,8	623,9	-	OOM
Fischer_U_26	1022,6	521,1	1258,6	639,8	-	OOM
Fischer_U_27	1190,8	522,8	1406,7	648,6	-	OOM
Fischer_U_28	1345,8	592,9	1392,9	716,9	-	OOM
Fischer_U_29	1631	652,6	1809,7	749,3	-	OOM
Fischer_U_30	2110,4	747,9	2202,5	796,9	-	OOM
Fischer_U_31	2060,1	746,8	2561,2	842,7	-	OOM
Fischer_U_32	2320,8	763,2	3256,1	860,6	-	OOM
Fischer_U_33	3016,2	825,3	2867,8	827,5	-	OOM
Fischer_U_34	2978,5	894,6	3634,8	969,2	-	OOM
Fischer_U_35	3376,9	899,6	3745,8	953,8	-	OOM
Fischer_U_36	3587,8	845,2	4905,9	1040,6	-	OOM
Fischer_U_37	4291,7	912,0	4910,9	1103,4	-	OOM
Fischer_U_38	4941,2	1079,5	5430,3	1098,6	-	OOM
Fischer_U_39	5670,1	1131,4	7856,7	1363,2	-	OOM
Fischer_U_40	6302,7	1167,9	8360,4	1350,5	-	OOM
Fischer_U_41	7391,9	1200,3	9565,2	1398,8	-	OOM
Fischer_U_42	6699,5	1100,7	8415,5	1289,6	-	OOM
Fischer_U_43	8034,3	1174,1	9676,7	1306,1	-	OOM
Fischer_U_44	8851,7	1184,0	11419,8	1459,3	-	OOM
Fischer_U_45	10316,0	1425,4	13345,9	1591,4	-	OOM
Fischer_U_46	13116,2	1599,6	13137,1	1570,1	-	OOM
Fischer_U_47	15380,4	1691,6	-	OOM	-	OOM
Fischer_U_48	13901,1	1515,9	-	OOM	-	OOM
Fischer_U_49	13420,2	1458,5	-	OOT	-	OOM
Fischer_U_50	18316,5	1793,6	-	OOT	-	OOM

Table A.2: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the Uppaal models of the Fischer mutual exclusion algorithm (e.g. depicted in Figure 3.1)

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
FDDI_2	1,1	87,2	1,0	87,2	0,1	35,1
FDDI_3	2,1	99,0	2,2	99,1	0,1	35,2
FDDI_4	5,8	128,1	7,3	123,5	0,1	35,2
FDDI_5	10,4	128,5	11,9	129,1	0,1	35,3
FDDI_6	38,8	190,9	26,5	178,1	0,1	35,3
FDDI_7	13,1	129,5	20,1	139,4	0,1	35,4
FDDI_8	13,0	129,7	36,5	160,8	0,1	35,4
FDDI_9	42,1	141,7	135,8	248,4	0,1	35,4
FDDI_10	316,8	369,1	136,0	341,7	0,1	35,5
FDDI_11	346,5	398,1	355,6	396,6	0,2	35,5
FDDI_12	423,6	490,4	401,3	379,1	0,2	35,6
FDDI_13	536,4	504,2	135,0	241,0	0,4	35,6
FDDI_14	857,1	603,3	958,8	726,9	0,6	35,7
FDDI_15	579,7	641,9	1141,3	711,7	0,9	35,7
FDDI_16	486,7	567,9	572,9	659	1,2	35,8
FDDI_17	299,5	461,6	724,7	729,8	2,0	35,9
FDDI_18	150,0	325,6	1689,8	953,8	2,7	35,9
FDDI_19	-	OOM	1254,8	899,6	4,3	36,0
FDDI_20	-	OOM	-	OOM	5,5	36,0
FDDI_21	-	OOM	-	OOM	9,1	35,8
FDDI_22	-	OOM	-	OOM	12,1	35,8
FDDI_23	-	OOM	-	OOM	17,4	35,9
FDDI_24	-	OOM	-	OOM	24,8	35,9
FDDI_25	-	OOM	-	OOM	24,2	36,0
FDDI_26	-	OOM	-	OOM	42,0	36,1
FDDI_27	-	OOM	-	OOM	67,1	36,1
FDDI_28	-	OOM	-	OOM	122,1	36,2
FDDI_29	-	OOM	-	OOM	109,0	36,4
FDDI_30	-	OOM	-	OOM	215,2	36,5
FDDI_31	-	OOM	-	OOM	324,4	36,5
FDDI_32	-	OOM	-	OOM	428,0	36,6
FDDI_33	-	OOM	-	OOM	850,2	36,8
FDDI_34	-	OOM	-	OOM	846,9	36,9
FDDI_35	-	OOM	-	OOM	1158,5	36,9
FDDI_36	-	OOM	-	OOM	1556,9	37,1
FDDI_37	-	OOM	-	OOM	1932,4	37,1
FDDI_38	-	OOM	-	OOM	3092,2	37,3
FDDI_39	-	OOM	-	OOM	3430,7	37,3
FDDI_40	-	OOM	-	OOM	4832,4	37,4
FDDI_41	-	OOM	-	OOM	8229,8	37,6
FDDI_42	-	OOM	-	OOM	9196,0	37,9
FDDI_43	-	OOM	-	OOM	OOT	-

Table A.3: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the Uppaal models of the FDDI token ring protocol (e.g. depicted in Figure 3.5)

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
Fischer_B_1	0,5	65,7	0,5	65,0	0,1	34,7
Fischer_B_2	6,7	106,6	5,9	97,5	0,1	34,8
Fischer_B_3	390,3	287,3	512,5	329,6	0,1	34,8
Fischer_B_4	11823,1	956,5	17572,2	1082,8	0,2	35,4
Fischer_B_5	OOT	-	659,9	474,3	17,8	43,7
Fischer_B_6	OOT	-	OOT	-	16212,9	326,1
Fischer_B_7	OOT	-	OOT	-	OOT	-

Table A.4: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the models of the Fischer mutual exclusion algorithm from Bruttomesso [Bru+12] (e.g. depicted in Figure 3.7)

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
Lamport_B_1	0,3	64,7	0,5	64,6	0,1	34,7
Lamport_B_2	5,3	117,7	5,2	117,7	0,1	34,7
Lamport_B_3	116,5	185,6	116,6	185,4	0,1	34,7
Lamport_B_4	3764,9	490,9	3761,7	490,3	0,1	35,5
Lamport_B_5	OOT	-	OOT	-	0,5	40,5
Lamport_B_6	OOT	-	OOT	-	5,3	78,8
Lamport_B_7	OOT	-	OOT	-	48,7	394,2
Lamport_B_8	OOT	-	OOT	-	-	OOM

Table A.5: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the models of the Lamport mutual exclusion algorithm from Bruttomesso [Bru+12] (e.g. depicted in Figure 3.8)

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
Lamport_S_1	0,5	68,6	0,5	68,6	0,1	34,7
Lamport_S_2	1,1	79,8	1,2	79,8	0,1	34,7
Lamport_S_3	2,4	101,9	2,4	101,9	0,1	34,7
Lamport_S_4	4,2	100,7	4,4	100,7	0,1	34,9
Lamport_S_5	13,4	106,5	13,3	106,7	0,1	35,4
Lamport_S_6	12,6	113,3	12,7	113,2	0,4	38,1
Lamport_S_7	22,7	138,7	22,6	138,4	2,3	51,2
Lamport_S_8	58,0	130,0	58,9	129,8	12,9	112,7
Lamport_S_9	80,2	135,7	79,9	135,8	71,7	427,8
Lamport_S_10	85,9	154,1	85,9	154,2	392,5	1844,1
Lamport_S_11	469,9	204,8	471,5	205,3	-	OOM
Lamport_S_12	572,9	246,3	567,9	244,7	-	OOM
Lamport_S_13	477,0	221,1	478,6	221,5	-	OOM
Lamport_S_14	1734,3	351,1	1735,7	350,8	-	OOM
Lamport_S_15	267,0	199,5	267,6	199,9	-	OOM
Lamport_S_16	12639,2	773,6	12654,1	772,8	-	OOM
Lamport_S_17	5567,5	469,0	5550,6	463,4	-	OOM
Lamport_S_18	6713,9	590,0	6743,7	590,3	-	OOM
Lamport_S_19	OOT	-	OOT	-	-	OOM

Table A.6: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the models of the shrunk Lamport mutual exclusion algorithm (e.g. depicted in Figure 3.10)

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
ShavitLynch_B_1	0,7	81,6	0,7	81,6	0,1	34,8
ShavitLynch_B_2	17,3	127,3	29,6	144,9	0,1	34,8
ShavitLynch_B_3	12603,7	1028,7	3870,3	599,1	0,1	34,9
ShavitLynch_B_4	2194,2	633,1	2626,1	665,3	0,3	35,7
ShavitLynch_B_5	OOT	-	OOT	-	28,5	46,8
ShavitLynch_B_6	OOT	-	OOT	-	OOT	-

Table A.7: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the models of the Shavit-Lynch mutual exclusion algorithm from Bruttomesso [Bru+12] (e.g. depicted in Figure 3.9)

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
ShavitLynch_P_1	0,6	69,1	0,5	69,1	0,1	34,7
ShavitLynch_P_2	9,9	135,4	10,7	141,7	0,1	34,8
ShavitLynch_P_3	149,5	203,6	208,0	215,7	0,1	34,8
ShavitLynch_P_4	3091,8	409,8	4425,3	457,8	0,1	34,8
ShavitLynch_P_5	OOT	-	OOT	-	0,1	34,9
ShavitLynch_P_6	OOT	-	OOT	-	0,1	35,0
ShavitLynch_P_7	OOT	-	OOT	-	0,2	35,5
ShavitLynch_P_8	OOT	-	OOT	-	1,0	36,8
ShavitLynch_P_9	OOT	-	OOT	-	4,5	40,9
ShavitLynch_P_10	OOT	-	OOT	-	18,8	54,6
ShavitLynch_P_11	OOT	-	OOT	-	76,7	113,4
ShavitLynch_P_12	OOT	-	OOT	-	307,8	305,2
ShavitLynch_P_13	OOT	-	OOT	-	1218,5	930,4
ShavitLynch_P_14	OOT	-	OOT	-	-	OOM

Table A.8: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the PAT models of the Shavit-Lynch mutual exclusion algorithm (e.g. depicted in Figure 3.11)

	IC3 with Zones(LCI)		IC3 with Zones(CLI)		Uppaal	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
FDDIcount_1	0,4	56,6	0,5	56,6	0,1	34,8
FDDIcount_2	0,7	63,7	0,7	63,7	0,1	34,8
FDDIcount_3	0,6	57,9	0,5	57,9	0,1	34,8
FDDIcount_4	1,4	77,9	1,5	77,9	0,1	34,9
FDDIcount_5	1,7	82,7	2,2	82,8	0,1	34,9
FDDIcount_6	2,4	88,7	4,7	105,0	0,1	35,0
FDDIcount_7	0,8	59,6	0,7	59,7	0,1	35,0
FDDIcount_8	2,6	83,4	2,4	73,7	0,1	35,1
FDDIcount_9	1,9	77,2	2,9	77,5	0,1	35,1
FDDIcount_10	0,9	60,8	1,0	61,1	0,1	35,2
FDDIcount_11	4,4	95,7	7,8	100,6	0,2	35,3
FDDIcount_12	3,3	76,9	7,3	77,3	0,2	35,3
FDDIcount_13	4,7	87,4	6,0	86,3	0,4	35,4
FDDIcount_14	5,6	93,1	7,1	93,4	0,6	35,4
FDDIcount_15	1,7	63,2	1,6	63,1	0,9	35,5
FDDIcount_16	2,0	72,0	3,8	81,1	1,2	35,5
FDDIcount_17	3,8	80,8	4,1	84,0	2,0	35,6
FDDIcount_18	9,1	110,0	16,3	121,5	2,6	35,7
FDDIcount_19	5,2	101,0	6,0	103,8	4,3	35,7
FDDIcount_20	10,6	115,0	10,7	116,1	5,5	35,8
FDDIcount_21	10,2	123,2	5,3	94,1	9,1	35,8
FDDIcount_22	9,5	126,0	11,9	134,0	12,2	36,0
FDDIcount_23	7,7	121,4	14,4	131,8	17,4	36,1
FDDIcount_24	17,7	144,6	31,7	154,0	24,8	36,1
FDDIcount_25	5,7	106,4	11,5	132,7	24,2	36,1
FDDIcount_26	18,0	158,2	30,3	179,0	42,0	36,2
FDDIcount_27	3,7	87,5	3,4	110,0	67,1	36,3
FDDIcount_28	3,9	111,7	3,6	112,5	122,1	36,4
FDDIcount_29	18,9	166,0	23,2	184,2	108,9	36,5
FDDIcount_30	18,4	160,6	30,3	168,1	215,3	36,5
FDDIcount_31	27,4	182,4	81,9	322,7	327,6	36,7
FDDIcount_32	3,8	111,9	4,3	112,2	428,3	36,7
FDDIcount_33	16,1	185,3	29,3	214,3	851,3	36,9
FDDIcount_34	18,5	193,9	31,9	241,6	846,9	37,0
FDDIcount_35	8,4	141,3	10,1	186,7	1158,9	37,0
FDDIcount_36	27,0	253,7	60,4	318,3	1157,3	37,2
FDDIcount_37	19,3	212,8	24,3	240,0	1932,9	37,2
FDDIcount_38	35,1	300,8	87,7	351,9	3096,8	37,4
FDDIcount_39	29,2	245,6	69,9	347,1	3429,3	37,5
FDDIcount_40	36,0	326,6	64,9	401,0	4833,3	37,5
FDDIcount_41	67,3	378,2	134,1	424,3	8226,1	37,7
FDDIcount_42	31,0	357,1	44,8	383,8	9188,8	38
FDDIcount_43	26,6	302,6	82,8	405,0	OOT	-
FDDIcount_44	37,6	390,3	129,5	440,4	OOT	-
FDDIcount_45	30,9	367,7	42,0	375,2	OOT	-
FDDIcount_46	42,2	395,6	62,6	405,8	OOT	-
FDDIcount_47	41,4	400,0	58,8	402,8	OOT	-
FDDIcount_48	74,3	374,0	500,7	431,9	OOT	-
FDDIcount_49	27,9	375,5	46,8	387,1	OOT	-
FDDIcount_50	36,7	384,5	58,4	387,6	OOT	-

Table A.9: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the altered models of the FDDI token ring protocol including an integer variable *cnt* to count the number of automata in the critical section (e.g. depicted in Figure 3.6)

	Fischer_U as before				Fischer_U with switched identifiers			
	IC3 with Zones(LCI)		IC3 with Zones(CLI)		IC3 with Zones(LCI)		IC3 with Zones(CLI)	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)
1 process	0,5	60,3	0,5	60,3	0,5	60,3	0,4	60,3
2 proc.	0,9	78	0,9	80,1	0,8	75,4	0,8	71,3
3 proc.	1,8	95,2	2,2	99,5	1,3	82,7	1,3	82,8
4 proc.	3,5	100	4,3	108,9	2,5	91,6	2,4	89,5
5 proc.	4,3	98,7	5,2	110,5	5,5	101	5,9	99,7
6 proc.	10	147,3	9,9	137,7	11,7	117,2	12,2	113,1
7 proc.	14,1	148,7	15,5	159,2	18,4	123	21,4	127,2
8 proc.	14,1	139,3	22,3	175,1	44,9	156	50,6	155,9
9 proc.	26,6	171,9	35,1	180,2	51,9	145,5	61,3	154,8
10 proc.	34,4	175,8	33,3	180,5	108,7	167,5	83,8	162,1
11 proc.	54,2	235,6	61,9	219,9	147,2	186,6	139,8	182,5
12 proc.	73,2	220,5	83,1	267,4	204,9	200,7	181,2	187,8
13 proc.	97,4	232,5	90,7	262,5	309,3	231,5	363	235,9
14 proc.	104,9	253,6	144,7	287,2	478,7	273,7	407,5	239,2
15 proc.	158,1	269,7	176,1	306,8	481,4	262,3	735	306,3
16 proc.	153,9	282,6	208,4	324,9	638,7	284,4	824,3	311
17 proc.	221,7	328,1	226,5	383,2	1084,4	371,9	1099,1	355,6
18 proc.	290	355,6	347,6	369,8	1475,6	434,3	1483,5	408,1
19 proc.	384,9	421,3	351,7	417,3	2003,2	475,9	2495,8	516,3
20 proc.	432,9	382	408,7	403,6	2646,8	561,4	4561,7	689,1
21 proc.	418,4	415	546,2	459,3	3459,4	608,2	4130,9	629,8
22 proc.	727	497,8	615,6	454	4590,4	707,5	5157,5	705,2
23 proc.	690,1	475,5	789,3	563,7	5452,5	781,7	7743,2	861,1
24 proc.	714,3	490,1	929,6	559,6	7877,7	973,0	7850,3	870,0
25 proc.	887,8	527,3	986,8	623,9	8373,4	966,4	9984,7	971,5
26 proc.	1022,6	521,1	1258,6	639,8	11198,8	1136,2	12908,5	1151,9
27 proc.	1190,8	522,8	1406,7	648,6	OOT	-	15701,0	1270,2
28 proc.	1345,8	592,9	1392,9	716,9	OOT	-	OOT	-
29 proc.	1631	652,6	1809,7	749,3	OOT	-	OOT	-
30 proc.	2110,4	747,9	2202,5	796,9	OOT	-	OOT	-
31 proc.	2060,1	746,8	2561,2	842,7	OOT	-	OOT	-
32 proc.	2320,8	763,2	3256,1	860,6	OOT	-	OOT	-
33 proc.	3016,2	825,3	2867,8	827,5	OOT	-	OOT	-
34 proc.	2978,5	894,6	3634,8	969,2	OOT	-	OOT	-
35 proc.	3376,9	899,6	3745,8	953,8	OOT	-	OOT	-
36 proc.	3587,8	845,2	4905,9	1040,6	OOT	-	OOT	-
37 proc.	4291,7	912,0	4910,9	1103,4	OOT	-	OOT	-
38 proc.	4941,2	1079,5	5430,3	1098,6	OOT	-	OOT	-
39 proc.	5670,1	1131,4	7856,7	1363,2	OOT	-	OOT	-
40 proc.	6302,7	1167,9	8360,4	1350,5	OOT	-	OOT	-
41 proc.	7391,9	1200,3	9565,2	1398,8	OOT	-	OOT	-
42 proc.	6699,5	1100,7	8415,5	1289,6	OOT	-	OOT	-
43 proc.	8034,3	1174,1	9676,7	1306,1	OOT	-	OOT	-
44 proc.	8851,7	1184,0	11419,8	1459,3	OOT	-	OOT	-
45 proc.	10316,0	1425,4	13345,9	1591,4	OOT	-	OOT	-

Table A.10: Comparison of runtime (seconds) and memory consumption (MB) during our scalability experiments using two heuristics for variable ordering (LCI, CLI) and the Uppaal models of the Fischer mutual exclusion algorithm (e.g. depicted in Figure 3.1) with distinct assignment of identifiers to the locations: Runtime and memory consumption are shown for the assignment as used before, where l_i ($i \in \{0, 1, 2, 3\}$) is assigned identifier i, and for the switched assignments of identifiers for l_0 and l_2, s.t. l_0 is assigned identifier 2 and l_2 is assigned 0

	Size (KB)	Runtime of Validation (s)
1 process	1	0,4
2 processes	1	0,4
3 processes	2	0,5
4 processes	3	0,5
5 processes	4	0,5
6 processes	7	0,5
7 processes	10	0,7
8 processes	14	0,7
9 processes	17	0,7
10 processes	17	0,7
11 processes	28	0,8
12 processes	40	1,0
13 processes	38	0,8
14 processes	47	1,0
15 processes	162	2,2
16 processes	66	1,1
17 processes	80	1,9
18 processes	82	1,9
19 processes	106	1,8
20 processes	97	1,9
21 processes	100	2,1
22 processes	119	2,7
23 processes	168	3,4
24 processes	176	3,4
25 processes	191	3,7
26 processes	222	4,0
27 processes	213	4,0
28 processes	258	4,5
29 processes	294	5,3
30 processes	300	5,4
31 processes	318	6,0
32 processes	355	6,1
33 processes	298	5,8
34 processes	435	7,7
35 processes	392	9,3
36 processes	391	8,6
37 processes	437	8,8
38 processes	499	10,0
39 processes	645	13,3
40 processes	656	13,4
41 processes	634	15,0
42 processes	550	12,2
43 processes	687	14,2
44 processes	678	17,2
45 processes	807	17,5
46 processes	704	15,8
47 processes	791	15,6
48 processes	850	20,0
49 processes	595	11,6
50 processes	291	14,3

Table A.11: Size (MB) of the inductive strengthenings computed for the *Fischer_U* models during the verification runs in Table A.2, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
1 process	1	0,3
2 processes	1	0,5
3 processes	1	0,5
4 processes	2	0,5
5 processes	2	0,5
6 processes	5	0,5
7 processes	5	0,7
8 processes	9	0,6
9 processes	9	0,6
10 processes	13	0,6
11 processes	26	0,7
12 processes	16	0,7
13 processes	31	1,0
14 processes	26	0,9
15 processes	56	1,1
16 processes	45	1,0
17 processes	50	1,1
18 processes	62	1,5
19 processes	66	1,9
20 processes	81	2,0
21 processes	108	2,8
22 processes	108	3,0
23 processes	161	3,6
24 processes	132	3,2
25 processes	119	3,2
26 processes	146	3,7
27 processes	165	3,8

Table A.12: Size (MB) of the inductive strengthenings computed for the *Fischer_U* models with switched location identifiers during the verification runs in Table A.10, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
2 processes	1	0,5
3 processes	2	0,6
4 processes	2	0,5
5 processes	3	0,5
6 processes	6	0,7
7 processes	7	0,6
8 processes	12	0,6
9 processes	13	0,7
10 processes	15	0,7
11 processes	26	0,9
12 processes	40	1,0
13 processes	51	1,1
14 processes	48	1,1
15 processes	162	2,2
16 processes	67	1,4
17 processes	78	1,6
18 processes	88	1,7
19 processes	80	1,6
20 processes	99	2,2
21 processes	114	2,3
22 processes	112	2,2
23 processes	153	3,3
24 processes	160	3,3
25 processes	154	2,8

Table A.13: Size (MB) of the inductive strengthenings computed for the $CSMA/CD$ models during the verification runs in Table A.1, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
2 processes	2	0,4
3 processes	2	0,5
4 processes	4	0,5
5 processes	7	0,6
6 processes	4	0,6
7 processes	5	0,6
8 processes	7	0,6
9 processes	28	0,9
10 processes	11	0,7
11 processes	38	1,1
12 processes	20	0,9
13 processes	19	0,9
14 processes	32	1,0
15 processes	23	0,9
16 processes	120	1,9
17 processes	70	1,3
18 processes	175	2,6
19 processes	55	1,5

Table A.14: Size (MB) of the inductive strengthenings computed for the $FDDI$ models during the verification runs in Table A.3, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
all verified instances	1	0,5-2,2

Table A.15: Size (MB) of the inductive strengthenings, which is the same for all the *FDDIcount* models, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
1 process	1	0,4
2 processes	10	0,6
3 processes	202	2,9
4 processes	2040	152,5
5 processes	247	8,0

Table A.16: Size (MB) of the inductive strengthenings computed for the *Fischer_B* models during the verification runs in Table A.4, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
1 process	1	0,3
2 processes	10	0,6
3 processes	132	1,9
4 processes	1288	68,7

Table A.17: Size (MB) of the inductive strengthenings computed for the *Lamport_B* models during the verification runs in Table A.5, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
1 process	1	0,4
2 processes	2	0,4
3 processes	4	0,5
4 processes	6	0,5
5 processes	22	0,6
6 processes	11	0,6
7 processes	9	0,6
8 processes	67	1,0
9 processes	11	0,7
10 processes	81	1,2
11 processes	330	3,6
12 processes	392	4,1
13 processes	431	5,0
14 processes	1088	23,1
15 processes	143	1,8
16 processes	1873	49,0
17 processes	376	4,1
18 processes	2191	53,5

Table A.18: Size (MB) of the inductive strengthenings computed for the *Lamport_S* models during the verification runs in Table A.6, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
1 process	1	0,4
2 processes	36	1,0
3 processes	1077	52,1
4 processes	755	42,3

Table A.19: Size (MB) of the inductive strengthenings computed for the *ShavitLynch_B* models during the verification runs in Table A.7, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
1 process	1	0,4
2 processes	12	0,5
3 processes	177	2,7
4 processes	1008	55,8

Table A.20: Size (MB) of the inductive strengthenings computed for the *ShavitLynch_P* models during the verification runs in Table A.8, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

	Size (KB)	Runtime of Validation (s)
1 process	1	0,3

Table A.21: Size (MB) of the inductive strengthenings computed for the *Lemgo* model during the verification run, and runtime (seconds) needed for the validation checks that the formulae are indeed inductive strengthenings

B Proofs

B.1 Templates guarantee Symmetry

In the following, we show that all networks of timed automata created as defined in Definition 5.1.2 fulfill our notion of symmetry (Def. 5.2.2). The definitions used in Chapter 4 (Def. 4.3.2 and 4.2.2) can be considered a special case in which no synchronization is used and no extra automata are given.

We prove that the result of any swap (Def. 5.2.1) applied to an initial state is itself initial. In addition, any swap applied to two states s_1 and s_2 connected via transition using an edge e in A_i with time delay δ results in the states $\pi(s_1)$ and $\pi(s_2)$, which are connected via transition using the edge e in $A_{\pi(i)}$ with time delay δ. In the case of a synchronized edge, it holds that any swap applied to two states s_1 and s_2 connected via transition using edges e_1 and e_2 in A_i, A_j with time delay δ results in the states $\pi(s_1)$ and $\pi(s_2)$, which are connected via transition using edges e_1 and e_2 in $A_{\pi(i)}$, $A_{\pi(j)}$ with time delay δ. The first statement is proven in Lemma B.1.1, while the latter ones are proven in Lemmata B.1.2, B.1.3 and B.1.4. As a result of the combination of these Lemmata, our notion of symmetry holds for the defined networks of timed automata.

Recall, that π only swaps automata that are created from the template. A_1 to A_x are unaffected.

Lemma B.1.1. *Let $NTA_n = \langle A_1, \dots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. If π is a swap (Def. 5.2.1), then $s = s_0$ if and only if $\pi(s) = s_0$ for all $s \in S$.*

Proof: Given the initial state $s = s_0 = (\vec{l}, v_0^c, v_0^i)$, we show $\pi(s) = s$. The opposite direction follows from the fact that $\pi(\pi(s)) = s$.

- **Location vector:** Swaps are defined, s.t. they swap only symmetric timed automata that are instantiated from the template. Thus, the locations of the automata A_1 to A_x are unchanged, formally $\forall i \in \{1, \ldots, x\} : \pi(\vec{l})[i] = \vec{l}[\pi(i)] = \vec{l}[i]$ since $\pi(i) = i$. For all symmetric timed automata, all initial locations are the same as defined by the template (l_0). A swap is, thus, without effect on these locations, formally $\forall i \in \{x+1, \ldots, x+n\} : \pi(\vec{l})[i] = \vec{l}[\pi(i)] = l_0 = \vec{l}[i]$. Thus, it holds true that $\pi(\vec{l}) = \vec{l}$.

- **Integer valuation:** Since swap does not affect the values of identifier unaware integer variables, they remain unchanged. Formally, $\forall iv \in \mathcal{IV}_{\not{id}} : \pi(v_0^i)(iv) = v_0^i(iv)$. All identifier aware integer variables are initialized to the neutral value 0 and, thus, remain unchanged, too. Formally, $\forall iv \in \mathcal{IV}_{id} : \pi(v_0^i)(iv) = 0 = v_0^i(iv)$. In summary, $\pi(v_0^i) = v_0^i$ holds true.

- **Clock valuation:** Since all clocks are of the same value initially (0), their values remain the same when applying a swap. Formally, $\forall c \in C : \pi(v_0^c)(c) = 0 = v_0^c(c)$.

□

Lemma B.1.2. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Given any swap π (Def. 5.2.1), then $(s_1, s_2) \in \rightarrow_d$ if and only if $(\pi(s_1), \pi(s_2)) \in \rightarrow_d$ for all $s_1, s_2 \in S$.*

Proof: We prove that if $(s_1, s_2) \in \rightarrow_d$, then $(\pi(s_1), \pi(s_2)) \in \rightarrow_d$. The opposite direction follows from the fact that $\pi(\pi(s)) = s$. Let the states be given as $s_1 = (\vec{l}, v^i, v^c)$ and $s_2 = (\vec{l}, v^i, v^c + \delta)$ for a $\delta \geq 0$. Clearly, the location vectors in $\pi(s_1)$ and $\pi(s_2)$ are equal, as well as the integer valuations since they are equal in s_1 and s_2 and the swap is deterministic. In addition, the order whether a swap is applied before or after the time delay does not matter, formally $\pi(v^c) + \delta = \pi(v^c + \delta)$, since δ is added to every clock. It remains to be shown that the locations' invariants are satisfied by the swapped clock valuations if they are by the original ones. Assume the contrary, meaning the invariants are satisfied for the original states, but not the swapped ones. Then there exists δ' with $0 \leq \delta' \leq \delta$ such that $\pi(v^c + \delta') \not\models Inv^c(\pi(\vec{l}))$. Lemma B.1.5 states that $v^c + \delta' \not\models Inv^c(\vec{l})$, which contradicts the assumption that $\forall 0 \leq \delta' \leq \delta : v^c + \delta' \models Inv^c(\vec{l})$. □

Lemma B.1.3. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Given any swap π (Def. 5.2.1), then $(s_1, s_2) \in \rightarrow_e$ via unsynchronized edge e in A_k ($k \in \{1, \ldots, x+n\}$) if and only if $(\pi(s_1), \pi(s_2)) \in \rightarrow_e$ via e in $A_{\pi(k)}$ for all $s_1, s_2 \in S$.*

Proof: We prove that if $(s_1, s_2) \in \rightarrow_e$ via unsynchronized edge e in A_k ($k \in \{1, \ldots, x + n\}$) then $(\pi(s_1), \pi(s_2)) \in \rightarrow_e$ via e in $A_{\pi(k)}$. The opposite direction follows from the fact that $\pi(\pi(s)) = s$.

Let the states be given as $s_1 = (\vec{l}, v^c, v^i)$ and $s_2 = (\vec{l'}, v^{c\prime}, v^{i\prime})$ with edge $e = (l_x \xrightarrow{\epsilon, \phi, \psi(k), \omega(k), R} l_y) \in E_k$ for automaton A_k. We know the following facts.

- $\vec{l}[k] = l_x$, $\vec{l'}[k] = l_y$ and $\forall i \neq k : \vec{l'}[i] = \vec{l}[i]$,
- $v^c \models \phi$, $v^{c\prime} = v^c[R]$ and $v^{c\prime} \models Inv^c(\vec{l'})$,
- $v^i \models \psi(k)$, $v^{i\prime} = v^i[\omega(k)]$ and $v^{i\prime} \models Inv^i(\vec{l'})$.

There exists edge $e_\pi = (l_x \xrightarrow{\epsilon, \phi_\pi, \psi(\pi(k)), \omega(\pi(k)), R_\pi} l_y) \in E_{\pi(k)}$ in automaton $A_{\pi(k)}$ due to the definition of symmetric networks of timed automata, where ϕ_π and R_π are defined over the clocks $(C^g \cup C^l_{\pi(k)})$ usable in $A_{\pi(k)}$. Note, that π swaps only two timed automata and that $\pi(k) = k$ holds true if A_k is not swapped. Recall, that the extra automata are never swapped. We will show that e_π is applicable to $\pi(s_1)$ and results in $\pi(s_2)$. The edge is clearly enabled by $\pi(s_1) = (\pi(\vec{l}), \pi(v^c), \pi(v^i))$ as shown in the following.

- $\pi(\vec{l})[\pi(k)] = \vec{l}[\pi(\pi(k))] = \vec{l}[k] = l_x$,
- $\pi(v^c) \models \phi_\pi$ holds, since all global clocks keep their values and all local clocks of $A_{\pi(k)}$ are assigned the values of their respective local clocks in A_k during the swap. Formally, $\forall c_{\pi(k)} \in C^l_{\pi(k)} : \pi(v^c)(c_{\pi(k)}) = v^c(c_{\pi(\pi(k))}) = v^c(c_k)$, where c_x refers local clock $c \in C^l_x$.
- $\pi(v^i) \models \psi(\pi(k))$ holds due to Lemma B.1.6.

Furthermore, the resulting location vector and valuations after applying the enabled edge e_π to state $\pi(s_1)$ results in $\pi(s_2) = (\pi(\vec{l'}), \pi(v^{c\prime}), \pi(v^{i\prime}))$.

- Lemma B.1.9 guarantees that $\pi(\vec{l'}) = \pi(\vec{l})'$, where the latter denotes the location vector obtained by application of edge e_π to $\pi(s_1)$.
- Furthermore, the resulting location for automaton $A_{\pi(k)}$ is correct. Formally, $\pi(\vec{l'})[\pi(k)] = \vec{l'}[\pi(\pi(k))] = \vec{l'}[k] = l_y$.
- Lemma B.1.7 guarantees that $\pi(v^{c\prime}) = \pi(v^c[R]) = \pi(v^c)[R_\pi]$.
- Lemma B.1.8 guarantees that $\pi(v^{i\prime}) = \pi(v^i[\omega(k)]) = \pi(v^i)[\omega(\pi(k))]$.

Furthermore, the resulting valuations still satisfy the invariants.

- With $v^{c\prime} \models Inv^c(\vec{l'})$, Lemma B.1.5 states that $\pi(v^{c\prime}) \models Inv^c(\pi(\vec{l'}))$.
- It holds that $v^{i\prime} \models Inv^i(\vec{l'})$ with $v^{i\prime} = v^i[\omega(k)]$. We show that $\pi(v^{i\prime}) \models Inv^i(\pi(\vec{l'}))$. Assume the contrary. Then there exists $m \in \{1, \ldots, x + n\}$ with $\pi(v^{i\prime}) \not\models Inv^i_m(\pi(\vec{l'})[m]) = \psi(m) \in \Psi(\mathcal{IV}, m)$ of location $\pi(\vec{l'})[m]$ in automaton A_m. Lemma B.1.6 states, that $v^{i\prime} \not\models \psi(\pi(m)) \in \Psi(\mathcal{IV}, \pi(m))$ for $A_{\pi(m)}$

and since $\pi(\vec{l'})[m] = \vec{l'}[\pi(m)]$ it holds $v^{i'} \not\models Inv^i_{\pi(m)}(\vec{l'}[\pi(m)])$ and, thus, $v^{i'} \not\models Inv^i(\vec{l'})$. This is a contradiction to the above assumption. Thus, the integer invariant is satisfied.

The backwards direction simply follows from the fact, that $\pi(\pi(s)) = s$. □

Lemma B.1.4. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Given any swap π (Def. 5.2.1), then $(s_1, s_2) \in \rightarrow_e$ via synchronized edges e_1, e_2 in A_i and A_j ($i \in \{1, \ldots, x\}$, $j \in \{1, \ldots, x+n\}$) if and only if $(\pi(s_1), \pi(s_2)) \in \rightarrow_e$ via e_1, e_2 in $A_{\pi(i)}$, $A_{\pi(j)}$ for all $s_1, s_2 \in S$.*

Proof: We prove that if $(s_1, s_2) \in \rightarrow_e$ via synchronized edges e_1, e_2 in A_i and A_j ($i \in \{1, \ldots, x\}$, $j \in \{1, \ldots, x+n\}$), then $(\pi(s_1), \pi(s_2)) \in \rightarrow_e$ via e_1, e_2 in $A_{\pi(i)}$, $A_{\pi(j)}$. The opposite direction follows from the fact that $\pi(\pi(s)) = s$.

Let the states be given as $s_1 = (\vec{l}, v^c, v^i)$ and $s_2 = (\vec{l'}, v^{c'}, v^{i'})$ and the edges be given as $e_1 = (l_{x1} \xrightarrow{chan!, \phi_1, \psi_1(i), \omega_1(i), R_1} l_{y1}) \in E_i$ of automaton A_i and $e_2 = (l_{x2} \xrightarrow{chan?, \phi_2, \psi_2(j), \omega_2(j), R_2} l_{y2}) \in E_j$ of automaton A_j synchronized by channel *chan*. We know the following facts.

- $\vec{l}[i] = l_{x1}$, $\vec{l'}[i] = l_{y1}$, $\vec{l}[j] = l_{x2}$, $\vec{l'}[j] = l_{y2}$ and $\forall h \in \{1, \ldots, x+n\} \setminus \{i, j\}$: $\vec{l'}[h] = \vec{l}[h]$,
- $v^c \models \phi_1 \wedge \phi_2$, $v^{c'} = v^c[R_1 \cup R_2]$ and $v^{c'} \models Inv^c(\vec{l'})$,
- $v^i \models \psi_1(i) \wedge \psi_2(j)$, $v^{i'} = v^i[\omega_1(i); \omega_2(j)]$ and $v^{i'} \models Inv^i(\vec{l'})$.

With A_i being an extra automaton ($i \leq x$), we know that $\pi(i) = i$ and, thus, e_1 is not permuted ($e_1 = e_{1\pi} = (l_{x1} \xrightarrow{chan!, \phi_{1\pi}, \psi(\pi(i)), \omega(\pi(i)), R_{1\pi}} l_{y1}) \in E_{\pi(i)}$). The other permuted edge $e_{2\pi} = (l_{x2} \xrightarrow{chan?, \phi_{2\pi}, \psi(\pi(j)), \omega(\pi(j)), R_{2\pi}} l_{y2}) \in E_{\pi(j)}$ exists due to the symmetry of the automata created via template (if $j > x$) or due to the extra automata not being swapped (if $j \leq x$). Both permuted edges are clearly enabled by $\pi(s_1) = (\pi(\vec{l}), \pi(v^c), \pi(v^i))$ as shown in the following.

- $\pi(\vec{l})[\pi(i)] = \vec{l}[\pi(\pi(i))] = \vec{l}[i] = l_{x1}$ and $\pi(\vec{l})[\pi(j)] = \vec{l}[\pi(\pi(j))] = \vec{l}[j] = l_{x2}$,
- $\pi(v^c) \models \phi_{1\pi} \wedge \phi_{2\pi}$ holds, since all global clocks keep their values and as well as the local clocks of the extra automaton $A_{\pi(i)}$. The local clocks of $A_{\pi(j)}$ are assigned the values of their respective local clocks in A_j during the swap. Formally, $\forall c_{\pi(j)} \in C^l_{\pi(j)} : \pi(v^c)(c_{\pi(j)}) = v^c(\pi(c_{\pi(j)})) = v^c(c_{\pi(\pi(j))}) = v^c(c_j)$, where c_x refers local clock $c \in C^l_x$.
- $\pi(v^i) \models \psi_1(\pi(i))$ and $\pi(v^i) \models \psi_2(\pi(j))$ hold due to Lemma B.1.6.

Furthermore, the resulting location vector and valuations after applying the enabled edges $e_{1\pi}$ and $e_{2\pi}$ to state $\pi(s_1)$ results in $\pi(s_2) = (\pi(\vec{l}'), \pi(v^{c\prime}), \pi(v^{i\prime}))$.

- A trivial extension to Lemma B.1.9 guarantees that $\pi(\vec{l}') = \pi(\vec{l})'$, where the latter denotes the location vector obtained by application of edges $e_{1\pi}$ and $e_{2\pi}$ to $\pi(s_1)$.

- Furthermore, the resulting location for automaton $A_{\pi(i)}$ and $A_{\pi(j)}$ are correct. Formally, $\pi(\vec{l}')[\pi(i)] = \vec{l}'[\pi(\pi(i))] = \vec{l}'[i] = l_{y1}$ and $\pi(\vec{l}')[\pi(j)] = \vec{l}'[\pi(\pi(j))] = \vec{l}'[j] = l_{y2}$.

- Lemma B.1.7 guarantees that $\pi(v^{c\prime}) = \pi(v^c[R]) = \pi(v^c)[R_\pi]$ for $R = R_1 \cup R_2$ and permuted $R_\pi = R_{1\pi} \cup R_{2\pi}$.

- It holds true that $\pi(v^{i\prime}) = \pi(v^i[\omega_1(i);\omega_2(j)]) = \pi(v^i)[\omega_1(\pi(i));\omega_2(\pi(j))]$ (using a trivial extension to Lemma B.1.8).

Furthermore, the resulting valuations still satisfy the invariants.

- With $v^{c\prime} \models Inv^c(\vec{l}')$, Lemma B.1.5 states that $\pi(v^{c\prime}) \models Inv^c(\pi(\vec{l}'))$.

- It holds that $v^{i\prime} \models Inv^i(\vec{l}')$. We show that $\pi(v^{i\prime}) \models Inv^i(\pi(\vec{l}'))$. Assume the contrary. Then there exists $m \in \{1, \ldots, x+n\}$ with $\pi(v^{i\prime}) \not\models Inv^i_m(\pi(\vec{l}')[m]) = \psi(m) \in \Psi(\mathcal{IV}, m)$ of location $\pi(\vec{l}')[m]$ in automaton A_m. Lemma B.1.6 states, that $v^{i\prime} \not\models \psi(\pi(m)) \in \Psi(\mathcal{IV}, \pi(m))$ for $A_{\pi(m)}$ and since $\pi(\vec{l}')[m] = \vec{l}'[\pi(m)]$ it holds $v^{i\prime} \not\models Inv^i_{\pi(m)}(\vec{l}'[\pi(m)])$ and, thus, $v^{i\prime} \not\models Inv^i(\vec{l}')$. This is a contradiction to the above assumption. Thus, the integer invariant is satisfied.

The backwards direction simply follows from the fact, that $\pi(\pi(s)) = s$. Note, that the same argumentation holds true, if e_1 is the receiver edge (*chan*?) and e_2 is the sender edge (*chan*!). □

Lemma B.1.5. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Given any swap π (Def. 5.2.1), then $v^c \models Inv^c(\vec{l})$ if and only if $\pi(v^c) \models Inv^c(\pi(\vec{l}))$ for any state $s = (\vec{l}, v^c, v^i)$.*

Proof: We prove that if $\pi(v^c) \models Inv^c(\vec{l})$, then $v^c \models Inv^c(\vec{l})$. The opposite direction follows from the fact that $\pi(\pi(s)) = s$. Assume the contrary, i.e., that $\pi(v^c) \models Inv^c(\pi(\vec{l}))$, but $v^c \not\models Inv^c(\vec{l})$. Then there exists m $(1 \leq m \leq x+n)$ with $v^c \not\models Inv^c_m(\vec{l}[m])$. We show the contradiction that $\pi(v^c) \not\models Inv^c_{\pi(m)}(\pi(\vec{l})[\pi(m)])$ holds true.

If $m \leq x$, i.e., the invariant of the location of an extra automaton is violated, this clearly holds true as π does not swap its location and local clock values, as well as global clock values in state s. Otherwise m refers a symmetric timed automaton, in which case the contradiction also holds true since the local clock values are swapped according to π, as well as the locations. As a result, the same clock values are

applied to the same invariant constraint, only defined over a distinct set of local clocks.

Formally, we start by showing that each of the clocks that can be used in the automaton $A_{\pi(m)}$ has the same value in state $\pi(s)$ as its respective clock used in A_m has in state s. Each global clock $c \in C^g$ has the same value in both states ($\pi(v^c)(c) = v^c(c)$), since their values are not swapped. Each local clock $c \in C^l_{\pi(m)}$ of automaton $A_{\pi(m)}$ has the same value in state $\pi(s)$ as its respective clock $c \in C^l_m$ in state s, formalized as $\forall c_{\pi(m)} \in C^l_{\pi(m)} : \pi(v^c)(c_{\pi(m)}) = v^c(c_{\pi(\pi(m))}) = v^c(c_m)$ where c_x denotes local clock c of C^l_x. With $Inv^c_{\pi(m)}(\pi(\vec{l})[\pi(m)]) = Inv^c_{\pi(m)}(\vec{l}[\pi(\pi(m))]) = Inv^c_{\pi(m)}(\vec{l}[m])$, we know that $Inv^c_{\pi(m)}(\pi(\vec{l})[\pi(m)])$ is the same invariant constraint as $Inv^c_m(\vec{l}[m])$, but defined over the clocks used in $A_{\pi(m)}$. Thus, since the clock values of $A_{\pi(m)}$ in state $\pi(s)$ are the ones of A_m in state s, the invariant $Inv^c_{\pi(m)}(\pi(\vec{l})[\pi(m)])$ is not satisfied by the valuation $\pi(v^c)$. Formally, $\pi(v^c) \not\models Inv^c_{\pi(m)}(\pi(\vec{l})[\pi(m)])$ holds true, which is a contradiction. Thus, the assumption is incorrect and it holds that if $\pi(v^c) \models Inv^c(\pi(\vec{l}))$ then $v^c \models Inv^c(\vec{l})$. □

Lemma B.1.6. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Given any swap π (Def. 5.2.1), it holds $v^i \models \psi(m) \in \Psi(\mathcal{IV}, m)$ if and only if $\pi(v^i) \models \psi(\pi(m))$ for all $m \in \{1, ..., x+n\}$.*

Proof: We prove that if $\pi(v^i) \models \psi(\pi(m))$ then $v^i \models \psi(m)$. The opposite direction follows from the fact that $\pi(\pi(s)) = s$. Assume the contrary, i.e., $\pi(v^i) \models \psi(\pi(m))$, but $v^i \not\models \psi(m)$. There exists at least one integer variable $iv \in \mathcal{IV}$, s.t. one of the conjuncts $iv \bowtie n$ in $\psi(m)$ evaluates to false given the integer valuation v^i.

- If $iv \in \mathcal{IV}_{i\!d\!\!/}$, then $\pi(v^i)(iv) = v^i(iv)$ and, furthermore, the conjunct $iv \bowtie n$ in $\psi(\pi(m))$ is the same as in $\psi(m)$ since Definitions 4.1.2 and 4.1.5 do not take into account the process identifier for identifier unaware integer variables. Thus, $\pi(v^i) \not\models \psi(\pi(m))$ holds true, which is a contradiction.

- If $iv \in \mathcal{IV}_{id}$, then the restrictions $n \in \{0, m\}, \bowtie \in \{=, \neq\}$ apply for the constraint $iv \bowtie n$. Thus, there exist four possibilities, how the constraint may look like.

$iv \neq 0$: With the process identifier m not being used in $iv \neq 0$, this constraint also occurs in $\psi(\pi(m))$. Since the constraint is not satisfied by the value of iv in v^i, it holds true that $v^i(iv) = 0$, which is not affected by the swap ($\pi(v^i)(iv) = v^i(iv) = 0$). Clearly, the swapped constraint is not satisfied with applied swap ($\pi(v^i) \not\models \psi(\pi(m))$), which is a contradiction.

$iv \neq m$: With the process identifier m being used in $iv \neq m$, this constraint does not occur in $\psi(\pi(m))$, but is replaced by $iv \neq \pi(m)$. Since the constraint is not satisfied by the value of iv in v^i, it holds true that $v^i(iv) = m$, which

is swapped to $\pi(v^i)(iv) = \pi(m)$. Clearly, the swapped constraint is not satisfied ($\pi(v^i) \not\models iv \neq \pi(m)$), which is a contradiction.

$iv = 0$: With the process identifier m not being used in $iv = 0$, this constraint also occurs in $\psi(\pi(m))$. Since the constraint is not satisfied by the value of iv in v^i, it holds true that $v^i(iv) \neq 0$, which might be affected by the swap. By Def. 5.2.1 the swapped value is still distinct from zero, formally $\pi(v^i)(iv) \neq 0$. In consequence, the swapped constraint is not satisfied ($\pi(v^i) \not\models \psi(\pi(m))$), which is a contradiction.

$iv = m$: With the process identifier m being used in $iv = m$, this constraint does not occur in $\psi(\pi(m))$, but is replaced by $iv = \pi(m)$. Since the constraint is not satisfied by the value of iv in v^i, it holds true that $v^i(iv) = j \neq m$, which is swapped to $\pi(v^i)(iv) = \pi(j) \neq \pi(m)$. Clearly, the swapped constraint is not satisfied ($\pi(v^i) \not\models iv = \pi(m)$), which is a contradiction.

□

Lemma B.1.7. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Let a set R of clocks to be reset be given using the local clocks C_i^l for automaton A_i. Given any swap π (Def. 5.2.1), it holds true that $\pi(v^c[R]) = \pi(v^c)[R_\pi]$, where R_π equals the set R, but the local clocks C_i^l are replaced by $C_{\pi(i)}^l$.*

Proof: Assume the contrary. Then, there exists a clock c, s.t. $\pi(v^c[R])(c) \neq \pi(v^c)[R_\pi](c)$. This clock can either be global or local.

- Global clock:
  If $c \in C^g$, then $\pi(v^c[R])(c) = v^c[R](c)$. Either c is reset ($c \in R$), then it holds $(v^c[R])(c) = 0 = \pi(v^c)[R_\pi](c)$ since $c \in R_\pi$. Otherwise, c is not reset ($c \notin R$) and it holds that $(v^c[R])(c) = v^c(c) = \pi(v^c)(c) = \pi(v^c)[R_\pi](c)$ since $c \notin R_\pi$ and c being global. Both cases contradict $\pi(v^c[R])(c) \neq \pi(v^c)[R_\pi](c)$.

- Local clock:
  We distinguish two cases.

  - The clock is reset, i.e., $c_i \in R$ and $c_{\pi(i)} \in R_\pi$. Clearly, $\pi(v^c[R])(c_{\pi(i)}) = v^c[R](c_{\pi(\pi(i))}) = v^c[R](c_i) = 0 = \pi(v^c)[R_\pi](c_{\pi(i)})$, where c_x denotes local clock c of automaton A_x ($c \in C_x^l$).
  - The clock is not reset, i.e., $c_i \notin R$ and $c_{\pi(i)} \notin R_\pi$. It clearly holds true that $\pi(v^c[R])(c_{\pi(i)}) = v^c[R](c_{\pi(\pi(i))}) = v^c[R](c_i) = v^c(c_i) = \pi(v^c)(c_{\pi(i)}) = \pi(v^c)[R_\pi](c_{\pi(i)})$.

  Both cases contradict $\pi(v^c[R])(c) \neq \pi(v^c)[R_\pi](c)$.

□

Lemma B.1.8. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Given any swap π (Def. 5.2.1), it holds true that $\pi(v^i[\omega(k)]) = \pi(v^i)[\omega(\pi(k))]$ with $\omega(k) \in \Omega(\mathcal{IV}, k)$ and $\omega(\pi(k)) \in \Omega(\mathcal{IV}, \pi(k))$ being the same assignment but for distinct identifiers k and $\pi(k)$.*

Proof: Assume the contrary. This means that there exists an integer variable iv, s.t. $\pi(v^i[\omega(k)])(iv) \neq \pi(v^i)[\omega(\pi(k))](iv)$. If $iv \in \mathcal{IV}_{\not{id}}$, we have $\pi(v^i[\omega(k)])(iv) = v^i[\omega(k)](iv) = v^i[\omega(\pi(k))](iv) = \pi(v^i)[\omega(\pi(k))](iv)$, which is a contradiction. Otherwise, $iv \in \mathcal{IV}_{id}$ for which we reach the same contradiction easily by considering all possibilities of assignments. Since $iv \in \mathcal{IV}_{id}$, there exist three distinct cases.

- $iv := 0$
  Since iv is reset to the neutral value, the assignment $iv := 0$ is part of $\omega(k)$ and $\omega(\pi(k))$. Clearly, $\pi(v^i[\omega(k)])(iv) = 0 = \pi(v^i)[\omega(\pi(k))](iv)$ since π does not affect the value 0.

- $iv := k$
  The assignment $iv := k$ is part of $\omega(k)$ and the assignment $iv := \pi(k)$ is part of $\omega(\pi(k))$. Clearly, $v^i[\omega(k)](iv) = k$ and $\pi(v^i)[\omega(\pi(k))](iv) = \pi(k)$ and, thus, $\pi(v^i[\omega(k)])(iv) = \pi(k) = \pi(v^i)[\omega(\pi(k))](iv)$.

- $iv \notin \omega$
  Since the value of iv is not affected by the assignment, only the swap needs to be considered. Clearly, $\pi(v^i[\omega(k)])(iv) = \pi(v^i)(iv) = \pi(v^i)[\omega(\pi(k))](iv)$.

All cases contradict the assumption. □

Lemma B.1.9. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Given any swap π (Def. 5.2.1) and location vectors $\vec{l}$, $\vec{l'}$ with $\vec{l}[m] = l_x$, $\vec{l'}[m] = l_y$ and $\forall i \neq m : \vec{l'}[i] = \vec{l}[i]$, the order whether a swap π is applied before or after an edge e of A_m (with source and target location l_x and l_y) is taken does not matter. Note, that the edge is swapped, too, from automaton A_m to $A_{\pi(m)}$. Formally, it holds true that $\pi(\vec{l})' = \pi(\vec{l'})$, where $\pi(\vec{l})'[\pi(m)] = l_y$ and $\forall i \neq \pi(m) : \pi(\vec{l})'[i] = \pi(\vec{l})[i]$.*

Proof: Clearly, $\pi(\vec{l})[\pi(m)] = \vec{l}[\pi(\pi(m))] = \vec{l}[m] = l_x$. Thus, the edge in $A_{\pi(m)}$ is applicable (regarding the locations) for $\pi(\vec{l})$, if and only if the edge in A_m is applicable (regarding the locations) for $\vec{l}$. Furthermore, the resulting location vectors match regardless of the order in which the edge and the swap are applied. In case the swap is applied first, the location of $A_{\pi(m)}$ is defined as $\pi(\vec{l})'[\pi(m)] = l_y$, which matches the location when the edge is applied first ($\pi(\vec{l'})[\pi(m)] = \vec{l'}[m] = l_y$). The same holds true for the locations of all other automata. In case the swap is applied first, the locations are defined as $\forall i \neq \pi(m) : \pi(\vec{l})'[i] = \pi(\vec{l})[i]$. In case the edge is applied first, the locations match ($\forall i \neq \pi(m) : \pi(\vec{l'})[i] = \vec{l'}[\pi(i)] = \vec{l}[\pi(i)] = \pi(\vec{l})[i]$). Note, that $\pi(\pi(m)) = m$. □

B.2 Proof of Termination Theorem

The following lemmata are used in the proof of the Termination Theorem (Theorem 5.2.1). Theorem 4.5.1, presented in Section 4, is a special case of it, as it employs parameterized timed systems modeled as networks of timed automata without synchronization and extra timed automata. The first lemma below is employed to prove symmetric membership of states in the extrapolated inductive strengthening. The latter ones refer to state membership of reduced states as defined in Def. B.2.1.

Lemma B.2.1. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2. Let an inductive strengthening $\|F\|$ of a symmetric safety property ρ^n (Def. 5.2.5) invariant in NTA_n be given as computed by our algorithm* IC3 with Zones. *For any larger model with $m \geq n$ symmetric timed automata, let $TS = (S, s_0, \rightarrow)$ be the concrete semantics of NTA_m. For state $s \in S$, it holds true that $s \in \pi_u(\|F\|)$ if and only if $\pi(s) \in \pi(\pi_u(\|F\|))$ for any swap π (Def. 5.2.1), where $\pi_u(\|F\|)$ is a swapped inductive strengthening $\|F\|$ that refers $x + n$ automata A_{i_1} to $A_{i_{x+n}}$, while $\pi(\pi_u(\|F\|))$ refers $A_{\pi(i_1)}$ to $A_{\pi(i_{x+n})}$.*

Proof: We prove that if $\pi(s) \in \pi(\pi_u(\|F\|))$, then $s \in \pi_u(\|F\|)$, the opposite direction follows from the fact that $\pi(\pi(s)) = s$. Assume the contrary, i.e., $\pi(s) \in \pi(\pi_u(\|F\|))$ and $s \notin \pi_u(\|F\|)$.

Let s be given as $s = (\vec{l}, v^c, v^i)$, then $\pi(s)$ is defined as in Definition 5.2.1. Note, that the formula $\pi_u(\|F\|)$ is permuted (Def. 4.5.1) according to the operation *swap*. Thus, trivially $\pi(s) \notin \pi(\pi_u(\|F\|))$ holds true. We discuss a proof in more detail below. The inductive strengthening $\|F\|$ is in CNF form, i.e., a conjunction of clauses. Clearly, if $s \notin \pi_u(\|F\|)$, then there exists a clause c that is not satisfied by the interpretation of variables representing state s. Each clause is a disjunction of location-, clock and integer-literals. None of the literals in c is satisfied by the interpretation representing state s, as c is a disjunction. We show, that the clause $\pi(c)$, which is a clause in $\pi(\pi_u(\|F\|))$ is not satisfied by the interpretation representing state $\pi(s)$. Recall, that $\pi(i) = i$ for all extra automata and that these automata are not swapped.

- **Location-literal:** Every location literal l_m^k in clause c evaluates to false when considering the interpretation representing state s. Clearly, the boolean representation of the location identifier for location $\vec{l}[k]$ of automaton A_k in state s has value 0 at bit m. Trivially, the boolean representation of the location identifier for location $\pi(\vec{l})[\pi(k)] = \vec{l}[k]$ of automaton $A_{\pi(k)}$ in state $\pi(s)$ has also value 0 at bit m. The permuted literal in clause $\pi(c)$ is $l_m^{\pi(k)}$, which, thus, evaluates to false when considering the interpretation representing state $\pi(s)$. The same argumentation holds true for a negated literal $\neg l_m^k$ with value 1.

- **Clock-literal:** The value of global clocks is the same in state $\pi(s)$ as in state s. In addition, the clock variables of global clocks are not affected by the swap of the formula. The satisfiability of a clock literal, thus, remains the same in terms of global clocks. For local clocks, the following holds true. The value

of local clock $c_m \in C^l_k$ (with identifier m) in state s is the same as the one of local clock $c_m \in C^l_{\pi(k)}$ in state $\pi(s)$. The clock variable c^k_m representing the former local clock with index m in the formula is changed to clock variable $c^{\pi(k)}_m$ representing the latter local clock with index m. Thus, when checking satisfiability of the permuted clock literal using the interpretation representing state $\pi(s)$, the same value applies at the same position. As a consequence, the satisfiability remains unchanged and the permuted clock literal still evaluates to false for the permuted state $\pi(s)$.

- **Integer-literal:** Since swap does not affect the values of identifier unaware integer variables, they remain unchanged. In addition, it does not change the value with which these variables are compared in the clause. Thus, the satisfiability remains unchanged concerning identifier unaware integer variables. For identifier aware integer variables, the argumentation is as follows. The literals for this kind of variables are an element of $\psi(m)$ for some timed automaton A_m. Lemma B.1.6 states that the permuted integer valuation $\pi(v^i)$ in state $\pi(s)$ does not satisfy the permuted literal $\psi(\pi(m))$. Thus, all permuted integer literals with identifier aware and unaware integer variables evaluate to false, when considering the interpretation representing the permuted state $\pi(s)$.

As a consequence, every permuted literal evaluates to false, when considering the interpretation representing the permuted state $\pi(s)$. In summary, the clause $\pi(c)$ is not satisfied by $\pi(s)$ and, thus, $\pi(s) \notin \pi(\pi_u(\|F\|))$. Note, that the timed automata referred in the formula are swapped according to π. □

Using the above lemma, the following corollary can be defined. It states, that the extrapolated formulae are symmetric, i.e., a state s is a member of an extrapolated formula, if and only if all permutations $\pi(s)$ of the state are a member. Obviously, this corollary holds true due to the extrapolated formulae $\|F\|^{exp(m)}$ including all permutations of $\|F\|$.

Corollary B.2.2. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2. Let an inductive strengthening $\|F\|$ of a symmetric safety property ρ^n (Def. 5.2.5) invariant in NTA_n be given as computed by our algorithm* IC3 with Zones. *For any larger model with $m \geq n$ symmetric timed automata, let $TS = (S, s_0, \rightarrow)$ be the concrete semantics of NTA_m. For state $s \in S$, it holds true that $s \in \|F\|^{exp(m)}$ if and only if $\pi(s) \in \|F\|^{exp(m)}$ for any swap π as defined in Definition 5.2.1.*

The same argumentation holds true for symmetric safety properties. If a state s violates the safety property, then it satisfies one of the symmetric error state specifications, i.e., one permutation $err^n_{\pi_e}$ of the error state specification is satisfied. Clearly, due to symmetry, a permuted state $\pi(s)$ satisfies the permutation $err^n_{\pi(\pi_e)}$ of the above mentioned specification and, thus, also violates the symmetric safety property. Note, that the timed automata referred in the permuted error state specification are changed according to π as formalized in the following corollary.

Corollary B.2.3. *Let $NTA_n = \langle A_1, \ldots, A_{x+n} \rangle$ be an extended network of n symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 with concrete semantics $TS = (S, s_0, \rightarrow)$. Let ρ^n be a symmetric safety property (Def. 5.2.5) defined over $m \leq n$ timed automata, i.e., the largest error state specification is defined for NTA_m. For state $s \in S$, it holds true that $s \in err^n_{\pi_e}$ if and only if $\pi(s) \in err^n_{\pi(\pi_e)}$ for any swap π, where $err^n_{\pi_e}$ specifies values for automata A_{i_1} to $A_{i_{x+m}}$ while $err^n_{\pi(\pi_e)}$ specifies values for $A_{\pi(i_1)}$ to $A_{\pi(i_{x+m})}$.*

In the following, we define how to reduce a state that includes values for $n+2$ symmetric timed automata to a state that is only defined over $n+1$ symmetric automata.

Definition B.2.1. Let $NTA_{n+2} = \langle A_1, \ldots, A_{x+n+2} \rangle$ be an extended network of $n+2$ symmetric timed automata with x extra timed automata as defined in Definition 5.1.2. Let $s \in S$ be a state in the concrete semantics $TS = (S, s_0, \rightarrow)$ of NTA_{n+2}. We define the reduced state $s|_{n+1}$ to be equal to the state s in all locations and values with the location of A_{x+n+2} being discarded, as well as the values of its local clocks. Formally, given $s = (\vec{l}, v^c, v^i)$ we define $s|_{n+1} = (\vec{l}|_{n+1}, v^c|_{n+1}, v^i|_{n+1})$ as follows.

- $\vec{l}|_{n+1}$ is a vector of $x+n+1$ locations with $\forall 1 \leq i \leq x+n+1 : \vec{l}|_{n+1}[i] = \vec{l}[i]$
- $v^c|_{n+1} \in \mathbb{R}^{C^g \cup C^l_1 \cup \cdots \cup C^l_{x+n+1}}$ with
  - $\forall c \in C^g : v^c|_{n+1}(c) = v^c(c)$
  - $\forall 1 \leq i \leq x+n+1 : \forall c \in C^l_i : v^c|_{n+1}(c) = v^c(c)$
- $\forall iv \in \mathcal{IV} : v^i|_{n+1}(iv) = v^i(iv)$

The reduced state does not include a location for automaton A_{x+n+2} and also no values for local clocks in C^l_{x+n+2}.

When reducing a state, it obviously becomes a state of the concrete state space of the smaller model NTA_{n+1}. Furthermore, the reduced state satisfies its locations' invariant if the original one satisfied its' own. This is due to the invariant being a conjunction. We formalize this fact in Corollary B.2.4.

Corollary B.2.4. *Let $NTA_{n+2} = \langle A_1, \ldots, A_{x+n+2} \rangle$ be an extended network of $n+2$ symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 and let $NTA_{n+1} = \langle A_1, \ldots, A_{x+n+1} \rangle$ be the respective model with $n+1$ symmetric timed automata. Let $TS^{n+1} = (S^{n+1}, s^{n+1}_0, \rightarrow^{n+1})$ denote the concrete semantics of NTA_{n+1} and let $s \in S$ be a state in the concrete semantics $TS = (S, s_0, \rightarrow)$ of NTA_{n+2}. Clearly, it holds true that $s|_{n+1} \in S^{n+1}$. Furthermore, $v^c \models Inv^c(\vec{l})$ and $v^i \models Inv^i(\vec{l})$ implies $v^c|_{n+1} \models Inv^c(\vec{l}|_{n+1})$ and $v^i|_{n+1} \models Inv^i(\vec{l}|_{n+1})$.*

We use this reduce operation on states to argue about set membership. We start with the membership in the set of initial states.

Lemma B.2.5. *Let $NTA_{n+2} = \langle A_1, \dots, A_{x+n+2} \rangle$ be an extended network of $n+2$ symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 and let $NTA_{n+1} = \langle A_1, \dots, A_{x+n+1} \rangle$ be the respective model with $n+1$ symmetric timed automata. Let $TS^{n+2} = (S^{n+2}, s_0^{n+2}, \rightarrow^{n+2})$ and $TS^{n+1} = (S^{n+1}, s_0^{n+1}, \rightarrow^{n+1})$ denote the concrete semantics of NTA_{n+2} and of NTA_{n+1}, respectively. It holds true that $\forall s \in S^{n+2} : s = s_0^{n+2}$ implies $s|_{n+1} = s_0^{n+1}$.*

Proof: Assume the contrary, i.e., $s = s_0^{n+2}$ and $s|_{n+1} \neq s_0^{n+1}$. With the latter fact, we know that $s|_{n+1}$ must be distinct from s_0^{n+1}. We show that this implies s to be distinct from s_0^{n+2}, which is a contradiction. Let $s = (\vec{l}, v^c, v^i)$ be given. One of the following facts must hold for $s|_{n+1}$ to not be initial.

- There exists a location in the location vector $\vec{l}|_{n+1}$ that is not the initial location of the respective automaton, formally $\exists i \in \{1, \dots, x+n+1\}, s.t. \vec{l}|_{n+1}[i] \neq l_{0i}$. With $\forall i \in \{1, \dots, x+n+1\} : \vec{l}|_{n+1}[i] = \vec{l}[i]$, we know that the state s is not initial.
- There exists a clock value in $v^c|_{n+1}$ that is not equal to 0, formally $\exists c \in C^g \cup C_1^l \cup \dots \cup C_{x+n+1}^l : v^c|_{n+1}(c) \neq 0$. With Definition B.2.1, we conclude that $v^c(c) \neq 0$ and, thus, s is not initial.
- There exists an integer variable with a value that is not initial, formally $\exists iv \in \mathcal{IV} : v^i|_{n+1}(iv) \neq v_0^i(iv)$. With Definition B.2.1, we conclude that $v^i(iv) \neq v_0^i(iv)$ and, thus, s is not initial.

In all cases, s is not initial, which is a contradiction. □

Furthermore, if a state is a member of an extrapolated inductive strengthening for the model with $n+2$ symmetric timed automata, then the reduced state is a member of the extrapolated inductive strengthening for the respective model with $n+1$ symmetric timed automata. We formalize this fact in the following.

Lemma B.2.6. *Let $NTA_{n+2} = \langle A_1, \dots, A_{x+n+2} \rangle$ be an extended network of $n+2$ symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 and let $NTA_{n+1} = \langle A_1, \dots, A_{x+n+1} \rangle$ be the respective model with $n+1$ symmetric timed automata. Let $TS^{n+2} = (S^{n+2}, s_0^{n+2}, \rightarrow^{n+2})$ and $TS^{n+1} = (S^{n+1}, s_0^{n+1}, \rightarrow^{n+1})$ denote the concrete semantics of NTA_{n+2} and of NTA_{n+1}, respectively. Furthermore, let $\|F\|$ be the inductive strengthening of the symmetric safety property ρ^n computed by* IC3 with Zones *for NTA_n. It holds true that $\forall s \in S^{n+2} : s \in \|F\|^{exp(n+2)}$ implies $s|_{n+1} \in \|F\|^{exp(n+1)}$.*

Proof: Assume the contrary, i.e., $s \in \|F\|^{exp(n+2)}$ and $s|_{n+1} \notin \|F\|^{exp(n+1)}$. With the latter fact, we know that $s|_{n+1}$ does not satisfy one of the permuted formulae $\pi(\|F\|)$ in $\|F\|^{exp(n+1)}$. Clearly, this formula is part of the conjunction of $\|F\|^{exp(n+2)}$. With $\pi(\|F\|)$ being a part of formula $\|F\|^{exp(n+1)}$, it only refers the automata A_1 to A_{x+n+1}. Def. B.2.1 specifies that s and $s|_{n+1}$ agree on the respective values, as well as the integer valuation. Thus, $\pi(\|F\|)$ is not satisfied by s, which is a contradiction to $s \in \|F\|^{exp(n+2)}$. □

The opposite direction of the above lemma does not hold true in general. However, within certain restrictions the same idea is applicable. We show that if state s is not a member of the set of states represented by a permuted formula $\pi(\|F\|)$ that does not refer automaton A_{x+n+2}, then the reduced state is not a member either.

Lemma B.2.7. *Let $NTA_{n+2} = \langle A_1, \ldots, A_{x+n+2} \rangle$ be an extended network of $n+2$ symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 and let $NTA_{n+1} = \langle A_1, \ldots, A_{x+n+1} \rangle$ be the respective model with $n+1$ symmetric timed automata. Let $TS^{n+2} = (S^{n+2}, s_0^{n+2}, \rightarrow^{n+2})$ and $TS^{n+1} = (S^{n+1}, s_0^{n+1}, \rightarrow^{n+1})$ denote the concrete semantics of NTA_{n+2} and of NTA_{n+1}, respectively. Furthermore, let $\|F\|$ be the inductive strengthening of the symmetric safety property ρ^n computed by* IC3 with Zones *for the respective NTA_n. It holds true that $\forall s \in S^{n+2} : s \notin \pi(\|F\|)$ implies $s|_{n+1} \notin \pi(\|F\|)$ for any $\pi(\|F\|)$ that reasons only about $x+n$ timed automata in $\{A_1, \ldots, A_{x+n+1}\}$.*

Proof: Assume the contrary, i.e., $s \notin \pi(\|F\|)$ and $s|_{n+1} \in \pi(\|F\|)$. With the latter fact, we know that $s|_{n+1}$ satisfies the permuted formulae $\pi(\|F\|)$. Def. B.2.1 specifies that s and $s|_{n+1}$ agree on the locations and clocks of the automata A_1 to A_{x+n+1}, as well as the integer valuation. With $\pi(\|F\|)$ only referring timed automata A_1 to A_{x+n+1}, it is clearly satisfied by s, which is a contradiction. □

Lemma B.2.7 can not be applied for general permutations $\pi(\|F\|)$ of the inductive strengthening. In particular, it can not be applied if $\pi(\|F\|)$ refers timed automaton A_{x+n+2}, as the reduced state does not specify the required values and no conclusion about the set membership can be drawn.

The same argumentation as above is valid when reasoning about the safety property and error state specifications. Again, restrictions apply in form of the considered permuted error state specifications.

Lemma B.2.8. *Let $NTA_{n+2} = \langle A_1, \ldots, A_{x+n+2} \rangle$ be an extended network of $n+2$ symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 and let $NTA_{n+1} = \langle A_1, \ldots, A_{x+n+1} \rangle$ be the respective model with $n+1$ symmetric timed automata. Let $TS^{n+2} = (S^{n+2}, s_0^{n+2}, \rightarrow^{n+2})$ and $TS^{n+1} = (S^{n+1}, s_0^{n+1}, \rightarrow^{n+1})$ denote the concrete semantics of NTA_{n+2} and of NTA_{n+1}, respectively. Furthermore, let the error state specification err be given, defined over $m \leq n$ symmetric timed automata. It holds true that $\forall s \in S^{n+2} : s \in err_\pi^{n+2}$ implies $s|_{n+1} \in err_\pi^{n+1}$, where err_π^{n+2} is a permutation that specifies only values for $x+m$ timed automata in $\{A_1, \ldots, A_{x+n+1}\}$.*

Proof: Assume the contrary, i.e., $s \in err_\pi^{n+2}$ and $s|_{n+1} \notin err_\pi^{n+1}$. With the latter fact, we know that $s|_{n+1}$ is not included in the permuted error state specification err_π^{n+1}. Def. B.2.1 specifies that s and $s|_{n+1}$ agree on the locations and clocks of the automata A_1 to A_{x+n+1}, as well as the integer valuation. With err_π^{n+2} only reasoning about these values, we know that it does not include the state s, which is a contradiction. □

Lastly, we need to reason about transitions using reduced states. We reason about delay and edge transitions separately.

Lemma B.2.9. *Let $NTA_{n+2} = \langle A_1, \ldots, A_{x+n+2} \rangle$ be an extended network of $n+2$ symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 and let $NTA_{n+1} = \langle A_1, \ldots, A_{x+n+1} \rangle$ be the respective model with $n+1$ symmetric timed automata. Let $TS^{n+2} = (S^{n+2}, s_0^{n+2}, \rightarrow^{n+2})$ and $TS^{n+1} = (S^{n+1}, s_0^{n+1}, \rightarrow^{n+1})$ denote the concrete semantics of NTA_{n+2} and of NTA_{n+1}, respectively. It holds true that $\forall s, t \in S^{n+2} : (s,t) \in \rightarrow_d^{n+2}$ implies $(s|_{n+1}, t|_{n+1}) \in \rightarrow_d^{n+1}$.*

Proof: Assume the contrary, i.e., $(s,t) \in \rightarrow_d^{n+2}$ and $(s|_{n+1}, t|_{n+1}) \notin \rightarrow_d^{n+1}$. With the latter fact, we know that $(s|_{n+1}, t|_{n+1})$ is not a correct delay transition. There exist several possible reasons, namely distinct locations, integer valuations, improper clock valuations or a violated invariant. With $s = (\vec{l_s}, v_s^c, v_s^i)$ and $t = (\vec{l_t}, v_t^c, v_t^i)$, we show the details below for each of the reasons.

- If the location vectors of $s|_{n+1}$ and $t|_{n+1}$ are distinct, we conclude using Def. B.2.1 that $\vec{l_s}$ and $\vec{l_t}$ are also distinct, which is a contradiction to $(s,t) \in \rightarrow_d^{n+2}$.
- If the integer valuations of $s|_{n+1}$ and $t|_{n+1}$ are distinct, we conclude using Def. B.2.1 that v_s^i and v_t^i are also distinct, which is a contradiction to $(s,t) \in \rightarrow_d^{n+2}$.
- If the clock valuations of $s|_{n+1}$ and $t|_{n+1}$ are improper, meaning $\nexists \delta \geq 0$, s.t. $\forall c \in C^g \cup C_1^l \cup \cdots \cup C_{x+n+1}^l : v_t^c|_{n+1}(c) = v_s^c|_{n+1}(c) + \delta$, clearly the same holds true for v_t^c and v_s^c due to Def. B.2.1. This is a contradiction to $(s,t) \in \rightarrow_d^{n+2}$.
- If an invariant is violated by $s|_{n+1}$ and $t|_{n+1}$, then $\exists 0 \leq \delta' \leq \delta : v_s^c|_{n+1} + \delta' \nvDash Inv^c(\vec{l_s}|_{n+1})$. Using Def. B.2.1 and the fact that $Inv^c(\vec{l_s}|_{n+1})$ is included in $Inv^c(\vec{l_s})$ via conjunction, we conclude that $\exists 0 \leq \delta' \leq \delta : v_s^c + \delta' \nvDash Inv^c(\vec{l_s})$. This is a contradiction to $(s,t) \in \rightarrow_d^{n+2}$.

□

Lemma B.2.10. *Let $NTA_{n+2} = \langle A_1, \ldots, A_{x+n+2} \rangle$ be an extended network of $n+2$ symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 and let $NTA_{n+1} = \langle A_1, \ldots, A_{x+n+1} \rangle$ be the respective model with $n+1$ symmetric timed automata. Let $TS^{n+2} = (S^{n+2}, s_0^{n+2}, \rightarrow^{n+2})$ and $TS^{n+1} = (S^{n+1}, s_0^{n+1}, \rightarrow^{n+1})$ denote the concrete semantics of NTA_{n+2} and of NTA_{n+1}, respectively. It holds true that $\forall s, t \in S^{n+2} : (s,t) \in \rightarrow_e^{n+2}$ implies $(s|_{n+1}, t|_{n+1}) \in \rightarrow_e^{n+1}$ via an edge e of automaton A_i with $i \neq x+n+2$.*

Proof: Let $s = (\vec{l_s}, v_s^c, v_s^i)$ and $t = (\vec{l_t}, v_t^c, v_t^i)$ be given with $(s,t) \in \rightarrow_e^{n+2}$ via edge $e = (l_x \xrightarrow{\epsilon, \phi, \psi(i), \omega(i), R} l_y) \in E_i$ of automaton A_i with $i \neq x+n+2$. We show that $(s|_{n+1}, t|_{n+1}) \in \rightarrow_e^{n+1}$ via the same edge. With edge e being enabled in s, it clearly is enabled in $s|_{n+1}$ as shown below.

- In state s, the location vector includes location l_x for A_i. Since $i < x+n+2$, the same location is also given in the reduced state, formally $\vec{l_s}|_{n+1}[i] = \vec{l_s}[i] = l_x$.

- The clock constraint ϕ only reasons about the global clocks and local clocks of A_i ($C^g \cup C^l_i$). Clearly, $v^c_s \models \phi$ holds true. Using Def. B.2.1, we know that s and $s|_{n+1}$ agree on the values for these clocks and, thus, $v^c_s|_{n+1} \models \phi$.

- We know that $v^i_s \models \psi(i)$ and since the integer valuation remains unchanged $v^i_s|_{n+1} \models \psi(i)$.

Furthermore, the resulting valuations and locations have to be met.

- We know that $\vec{l_t}[i] = l_y$ and $\forall j \in \{1, \dots, x+n+2\} \setminus \{i\} : \vec{l_t}[j] = \vec{l_s}[j]$. With Definition B.2.1 we conclude that $\forall j \in \{1, \dots, x+n+1\} \setminus \{i\} : \vec{l_t}|_{n+1}[j] = \vec{l_t}[j] = \vec{l_s}[j] = \vec{l_s}|_{n+1}[j]$ and $\vec{l_t}|_{n+1}[i] = \vec{l_t}[i] = l_y$. This is the correct location vector after application of edge e to state $s|_{n+1}$.

- The set R of clocks that are reset when applying edge e only contains global clocks and local clocks from A_i (a subset of $C^g \cup C^l_i$). We know that $\forall c \in C : v^c_t(c) = \begin{cases} 0 & if\ c \in R \\ v^c_s(c) & else \end{cases}$. Using Def. B.2.1 we conclude that $\forall c \in C^g \cup C^l_1 \cup \dots \cup C^l_{x+n+1} : v^c_t|_{n+1}(c) = v^c_t(c) = \begin{cases} 0 & if\ c \in R \\ v^c_s(c) & else \end{cases}$. Using $\forall c \in C^g \cup C^l_1 \cup \dots \cup C^l_{x+n+1} : v^c_s|_{n+1}(c) = v^c_s(c)$, we conclude that $\forall c \in C^g \cup C^l_1 \cup \dots \cup C^l_{x+n+1} : v^c_t|_{n+1}(c) = \begin{cases} 0 & if\ c \in R \\ v^c_s|_{n+1}(c) & else \end{cases}$, which is the correct valuation after application of edge e to state $s|_{n+1}$.

- Since the integer valuation is unchanged ($v^i_s|_{n+1} = v^i_s$) and the assignment as well, the resulting valuation is correct ($v^i_s|_{n+1}[\omega(i)] = v^i_s[\omega(i)] = v^i_t = v^i_t|_{n+1}$).

With Corollary B.2.4 we conclude that the invariants are met. Thus, the edge can be applied for the reduced states as formalized in the lemma. □

Lemma B.2.11. *Let $NTA_{n+2} = \langle A_1, \dots, A_{x+n+2} \rangle$ be an extended network of $n+2$ symmetric timed automata with x extra timed automata as defined in Definition 5.1.2 and let $NTA_{n+1} = \langle A_1, \dots, A_{x+n+1} \rangle$ be the respective model with $n+1$ symmetric timed automata. Let $TS^{n+2} = (S^{n+2}, s^{n+2}_0, \rightarrow^{n+2})$ and $TS^{n+1} = (S^{n+1}, s^{n+1}_0, \rightarrow^{n+1})$ denote the concrete semantics of NTA_{n+2} and of NTA_{n+1}, respectively. It holds true that $\forall s, t \in S^{n+2} : (s,t) \in \rightarrow^{n+2}_e$ implies $(s|_{n+1}, t|_{n+1}) \in \rightarrow^{n+1}_e$ via synchronized edges e_1, e_2 of automata A_i and A_j with $i \in \{1, \dots, x\}$ and $j \neq x+n+2$.*

Proof: Let $s = (\vec{l_s}, v^c_s, v^i_s)$ and $t = (\vec{l_t}, v^c_t, v^i_t)$ be given with $(s,t) \in \rightarrow^{n+2}_e$ via edges $e_1 = (l_{x1} \xrightarrow{chan!, \phi_1, \psi_1(i), \omega_1(i), R_1} l_{y1}) \in E_i$ of automaton A_i with $i \in \{1, \dots, x\}$ and $e_2 = (l_{x2} \xrightarrow{chan?, \phi_2, \psi_2(j), \omega_2(j), R_2} l_{y2}) \in E_j$ of automaton A_j with $j \neq x+n+2$ synchronized by channel $chan$. We show that $(s|_{n+1}, t|_{n+1}) \in \rightarrow^{n+1}_e$ via the same edges.

To this end, we first show that these edges are enabled in $s|_{n+1}$ since they are enabled in s.

- In state s, the location vector includes location l_{x1} for A_i and l_{x2} for A_j, respectively. Since $i \leq x$ and $j < x+n+2$, the same locations are also given in the reduced state, formally $\vec{l_s}|_{n+1}[i] = \vec{l_s}[i] = l_{x1}$ and $\vec{l_s}|_{n+1}[j] = \vec{l_s}[j] = l_{x2}$.
- The clock constraint $\phi_1 \wedge \phi_2$ only reasons about the global clocks and local clocks of A_i and A_j ($C^g \cup C^l_i \cup C^l_j$). Clearly, $v^c_s \models \phi_1 \wedge \phi_2$ holds true. Using Def. B.2.1, we know that s and $s|_{n+1}$ agree on the values for these clocks and, thus, $v^c_s|_{n+1} \models \phi_1 \wedge \phi_2$.
- We know that $v^i_s \models \psi_1(i) \wedge \psi_2(j)$ and since the integer valuation remains unchanged $v^i_s|_{n+1} \models \psi(i) \wedge \psi_2(j)$.

Furthermore, the resulting valuations and locations have to be met.

- We know that $\vec{l_t}[i] = l_{y1}$, $\vec{l_t}[j] = l_{y2}$ and $\forall h \in \{1,\ldots,x+n+2\} \setminus \{i,j\} : \vec{l_t}[h] = \vec{l_s}[h]$. With Definition B.2.1 we conclude that $\forall h \in \{1,\ldots,x+n+1\} \setminus \{i,j\} : \vec{l_t}|_{n+1}[h] = \vec{l_t}[h] = \vec{l_s}[h] = \vec{l_s}|_{n+1}[h]$ and $\vec{l_t}|_{n+1}[i] = \vec{l_t}[i] = l_{y1}$ and $\vec{l_t}|_{n+1}[j] = \vec{l_t}[j] = l_{y2}$. This is the correct location vector after application of edges e_1, e_2 to state $s|_{n+1}$.
- The set $R = R_1 \cup R_2$ of clocks that are reset when applying edges e_1, e_2 only contains global clocks and local clocks from A_i and A_j (a subset of $C^g \cup C^l_i \cup C^l_j$). We know that $\forall c \in C : v^c_t(c) = \begin{cases} 0 & if\ c \in R \\ v^c_s(c) & else \end{cases}$. Using Def. B.2.1 we conclude that $\forall c \in C^g \cup C^l_1 \cup \cdots \cup C^l_{x+n+1} : v^c_t|_{n+1}(c) = v^c_t(c) = \begin{cases} 0 & if\ c \in R \\ v^c_s(c) & else \end{cases}$. Using $\forall c \in C^g \cup C^l_1 \cup \cdots \cup C^l_{x+n+1} : v^c_s|_{n+1}(c) = v^c_s(c)$, we conclude that $\forall c \in C^g \cup C^l_1 \cup \cdots \cup C^l_{x+n+1} : v^c_t|_{n+1}(c) = \begin{cases} 0 & if\ c \in R \\ v^c_s|_{n+1}(c) & else \end{cases}$, which is the correct valuation after application of edges e_1, e_2 to state $s|_{n+1}$.
- Since the integer valuation is unchanged ($v^i_s|_{n+1} = v^i_s$) and the assignments as well, the resulting valuation is correct ($v^i_s|_{n+1}[\omega_1(i), \omega_2(j)] = v^i_s[\omega_1(i), \omega_2(j)] = v^i_t = v^i_t|_{n+1}$).

With Corollary B.2.4 we conclude that the invariants are met. Thus, the edges can be applied for the reduced states as formalized in the lemma. Note, that the same argumentation holds true, if e_1 is the receiver edge (*chan*?) and e_2 is the sender edge (*chan*!). □

Bibliography

[AAH05] Jean-Raymond Abrial, Jean-Raymond Abrial, and A Hoare. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[Abd+02a] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. "Regular tree model checking". In: *Computer Aided Verification*. Springer. 2002, pp. 555–568.

[Abd+02b] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. "Regular Model Checking Made Simple and Effcient*". In: *CONCUR 2002—Concurrency Theory*. Springer, 2002, pp. 116–131.

[Abd+03] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. "Algorithmic improvements in regular model checking". In: *Computer Aided Verification*. Springer. 2003, pp. 236–248.

[Abd+04] Parosh Aziz Abdulla, Johann Deneux, Pritha Mahata, and Aletta Nylén. "Forward reachability analysis of timed Petri nets". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 343–362.

[Abd+99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. "Handling global conditions in parametrized system verification". In: *Computer Aided Verification*. Springer. 1999, pp. 134–145.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David Dill. "Model-checking for real-time systems". In: *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*. IEEE. 1990, pp. 414–425.

[AD90] Rajeev Alur and David Dill. "Automata for modeling real-time systems". In: *Automata, languages and programming*. Springer, 1990, pp. 322–335.

[AD94] Rajeev Alur and David L Dill. "A theory of timed automata". In: *Theoretical computer science* 126.2 (1994), pp. 183–235.

[ADM04] Parosh Aziz Abdulla, Johann Deneux, and Pritha Mahata. “Multi-clock timed networks”. In: *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*. IEEE. 2004, pp. 345–354.

[AH94] Rajeev Alur and Thomas A Henzinger. “A really temporal logic”. In: *Journal of the ACM (JACM)* 41.1 (1994), pp. 181–203.

[AJ03] Parosh Aziz Abdulla and Bengt Jonsson. “Model checking of systems with many identical timed processes”. In: *Theoretical Computer Science* 290.1 (2003), pp. 241–264.

[AK83] S Aggarwal and Robert P Kurshan. “Modelling Elapsed Time in Protocol Specification.” In: *Protocol Specification, Testing, and Verification*. Vol. 3. 1983, pp. 51–62.

[AK86] Krzysztof R Apt and Dexter C Kozen. “Limits for automatic verification of finite-state concurrent systems”. In: *Information Processing Letters* 22.6 (1986), pp. 307–309.

[Ake78] Sheldon B. Akers. “Binary decision diagrams”. In: *Computers, IEEE Transactions on* 100.6 (1978), pp. 509–516.

[Alu+92] Rajeev Alur, Costas Courcoubetis, David Dill, Nicolas Halbwachs, and Howard Wong-Toi. “An implementation of three algorithms for timing verification based on automata emptiness”. In: *Real-Time Systems Symposium, 1992*. IEEE. 1992, pp. 157–166.

[Aro+01] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Ying Xu, and Lenore Zuck. “Parameterized verification with automatically computed inductive assertions?” In: *Computer Aided Verification*. Springer. 2001, pp. 221–234.

[AS12] Gilles Audemard and Laurent Simon. “Glucose 2.1: Aggressive-but Reactive-Clause Database Management, Dynamic Restarts”. In: *International Workshop of Pragmatics of SAT (Affiliated to SAT)*. 2012.

[Asa+97] Eugene Asarin, Marius Bozga, Alain Kerbrat, Oded Maler, Amir Pnueli, and Anne Rasse. “Data-structures for the verification of timed automata”. In: *Hybrid and Real-Time Systems*. Springer. 1997, pp. 346–360.

[Ast+15] Lacramioara Astefanoaei, Souha Ben Rayana, Saddek Bensalem, Marius Bozga, and Jacques Combaz. “Compositional Verification of Parameterised Timed Systems”. In: *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*. Vol. 9058. Springer. 2015, p. 66.

[Aud+02] Gilles Audemard, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. “Bounded model checking for timed systems”. In: *Formal Techniques for Networked and Distributed Sytems?FORTE 2002*. Springer, 2002, pp. 243–259.

[B+05] Sami Beydeda, Matthias Book, Volker Gruhn, et al. *Model-driven software development*. Vol. 15. Springer, 2005.

[Bac+13] John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. "Regression verification using impact summaries". In: *Model Checking Software*. Springer, 2013, pp. 99–116.

[Bar+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "Cvc4". In: *Computer aided verification*. Springer. 2011, pp. 171–177.

[Bau+12] Jason Baumgartner, Alexander Ivrii, Arie Matsliah, and Hari Mony. "IC3-guided abstraction". In: *Formal Methods in Computer-Aided Design (FMCAD), 2012*. IEEE. 2012, pp. 182–185.

[Bay+13] Sam Bayless, Celina G Val, Thomas Ball, Holger H Hoos, and Alan J Hu. "Efficient modular SAT solving for IC3". In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE. 2013, pp. 149–156.

[BBW14] Johannes Birgmeier, Aaron R Bradley, and Georg Weissenbacher. "Counterexample to induction-guided abstraction-refinement (CTIGAR)". In: *Computer Aided Verification*. Springer. 2014, pp. 831–848.

[BC00] Per Bjesse and Koen Claessen. "SAT-based verification without state space traversal". In: *Formal Methods in Computer-Aided Design*. Springer. 2000, pp. 409–426.

[BC10] Armin Biere and K Claessen. "Hardware model checking competition". In: *Hardware Verification Workshop*. 2010.

[Beh+04] Gerd Behrmann, Patricia Bouyer, Kim G Larsen, and Radek Pelánek. "Lower and upper bounds in zone based abstractions of timed automata". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 312–326.

[Beh+11] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. "Developing uppaal over 15 years". In: *Software: Practice and Experience* 41.2 (2011), pp. 133–142.

[Beh+99] Gerd Behrmann, Kim G Larsen, Justin Pearson, Carsten Weise, and Wang Yi. "Efficient timed reachability analysis using clock difference diagrams". In: *Computer aided verification*. Springer. 1999, pp. 341–353.

[Ben+96] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "Verification of an Audio Protocol with Bus Collision Using UPPAAL". In: *CAV96*. Ed. by Rajeev Alur and Thomas A. Henzinger. LNCS 1102. Springer–Verlag, July 1996, pp. 244–256.

[Ben02] Johan Bengtsson. *Clocks, dbms and states in timed systems*. Citeseer, 2002.

[Bey+13] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. "Precision reuse for efficient regression verification". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 389–399.

[Bey01] Dirk Beyer. “Improvements in BDD-based reachability analysis of timed automata”. In: *FME 2001: Formal Methods for Increasing Software Productivity*. Springer, 2001, pp. 318–343.

[BG14] Nikolaj Bjørner and Arie Gurfinkel. “Property Directed Polyhedral Abstraction”. In: *Verification, Model Checking, and Abstract Interpretation*. Springer. 2014, pp. 263–281.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[Bie08] Armin Biere. “PicoSAT essentials”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), pp. 75–97.

[Bie12] Armin Biere. “Lingeling and friends entering the SAT challenge 2012”. In: *Proceedings of SAT Challenge* (2012), pp. 33–34.

[BM07] Aaron R Bradley and Zohar Manna. “Checking safety by inductive generalization of counterexamples to induction”. In: *Formal Methods in Computer Aided Design, 2007. FMCAD’07*. IEEE. 2007, pp. 173–180.

[BOR13] Marcel Böhme, Bruno C d S Oliveira, and Abhik Roychoudhury. “Partition-based regression verification”. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 302–311.

[Bou+00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. “Regular model checking”. In: *Computer Aided Verification*. Springer. 2000, pp. 403–418.

[Bou+04] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. “Updatable timed automata”. In: *Theoretical Computer Science* 321.2 (2004), pp. 291–345.

[Bou09] Patricia Bouyer. “From Qualitative to Quantitative Analysis of Timed Systems”. Mémoire d’habilitation. Université Paris 7, Paris, France, Jan. 2009.

[Boz+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. “Kronos: A model-checking tool for real-time systems”. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer. 1998, pp. 298–302.

[BR00] Dirk Beyer and Heinrich Rust. “A tool for modular modelling and verification of hybrid systems”. In: *Proceedings of the 25th IFAC/IFIP Workshop on Real-Time Programming*. 2000, pp. 169–174.

[Bra11] Aaron R Bradley. “SAT-based model checking without unrolling”. In: *Verification, Model Checking, and Abstract Interpretation*. Springer. 2011, pp. 70–87.

[Bra12a] Aaron R Bradley. “IC3 and beyond: Incremental, Inductive Verification.” In: *CAV*. 2012, p. 4.

[Bra12b] Aaron R Bradley. "Understanding ic3". In: *Theory and Applications of Satisfiability Testing–SAT 2012*. Springer, 2012, pp. 1–14.

[Bru+12] Roberto Bruttomesso, Alessandro Carioni, Silvio Ghilardi, and Silvio Ranise. "Automated analysis of parametric timing-based mutual exclusion algorithms". In: *NASA Formal Methods*. Springer, 2012, pp. 279–294.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010.

[BT02] Ahmed Bouajjani and Tayssir Touili. "Extrapolating tree transformations". In: *Computer Aided Verification*. Springer. 2002, pp. 539–554.

[Bur90] Jerry R Burch. "Combining CTL, trace theory and timing models". In: *Automatic Verification Methods for Finite State Systems*. Springer. 1990, pp. 334–348.

[Cab+09] Gianpiero Cabodi, Paolo Camurati, Luz Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. "Speeding up model checking by exploiting explicit and hidden verification constraints". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association. 2009, pp. 1686–1691.

[CG12] Alessandro Cimatti and Alberto Griggio. "Software model checking via IC3". In: *Computer Aided Verification*. Springer. 2012, pp. 277–293.

[CGJ95] Edmund M Clarke, Orna Grumberg, and Somesh Jha. "Verifying parameterized networks using abstraction and regular languages". In: *CONCUR'95: Concurrency Theory*. Springer, 1995, pp. 395–407.

[CGR10] Alessandro Carioni, Silvio Ghilardi, and Silvio Ranise. "MCMT in the Land of Parametrized Timed Automata." In: *VERIFY@ IJCAR*. 2010, pp. 47–64.

[Cha+02] Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis". In: *Formal Methods in Computer-Aided Design*. Springer. 2002, pp. 33–51.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. "SMTInterpol: An interpolating SMT solver". In: *Model Checking Software*. Springer, 2012, pp. 248–254.

[Cho+11] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. "Incremental formal verification of hardware". In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc. 2011, pp. 135–143.

[CHR91] Zhou Chaochen, Charles Anthony Richard Hoare, and Anders P Ravn. "A calculus of durations". In: *Information processing letters* 40.5 (1991), pp. 269–276.

[Cim+13a] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. "Parameter synthesis with IC3". In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE. 2013, pp. 165–168.

[Cim+13b] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver". In: *Proceedings of TACAS*. Ed. by Nir Piterman and Scott Smolka. Vol. 7795. LNCS. Springer, 2013.

[Cim+14] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. "Ic3 modulo theories via implicit predicate abstraction". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 46–61.

[Cla+02] Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. "SAT based abstraction-refinement using ILP and machine learning techniques". In: *Computer Aided Verification*. Springer. 2002, pp. 265–279.

[Cla+96] Edmund M Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. "Exploiting symmetry in temporal logic model checking". In: *Formal Methods in System Design* 9.1-2 (1996), pp. 77–104.

[CNQ09] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. "Strengthening model checking techniques with inductive invariants". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 28.1 (2009), pp. 154–158.

[Con+05] Christopher L Conway, Kedar S Namjoshi, Dennis Dams, and Stephen A Edwards. "Incremental algorithms for inter-procedural analysis of safety properties". In: *Computer Aided Verification*. Springer. 2005, pp. 449–461.

[Coo71] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM. 1971, pp. 151–158.

[Daw+96] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. "The tool KRONOS". In: *Hybrid Systems III*. Springer, 1996, pp. 208–219.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[Dij78] EdsgerW. Dijkstra. "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs". English. In: *Programming Methodology*. Ed. by David Gries. Texts and Monographs in Computer Science. Springer New York, 1978, pp. 166–175. ISBN: 978-1-4612-6317-3.

[Dil90] David L Dill. "Timing assumptions and verification of finite-state concurrent systems". In: *Automatic verification methods for finite state systems*. Springer. 1990, pp. 197–212.

[DKP09] Stephanie Drzevitzky, Uwe Kastens, and Marco Platzner. "Proof-carrying hardware: Towards runtime verification of reconfigurable modules". In: *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on*. IEEE. 2009, pp. 189–194.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[DP60] Martin Davis and Hilary Putnam. "A computing procedure for quantification theory". In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

[DRS03] Leonardo De Moura, Harald Rueß, and Maria Sorea. "Bounded model checking and induction: From refutation to verification". In: *Computer Aided Verification*. Springer. 2003, pp. 14–26.

[DT98] Conrado Daws and Stavros Tripakis. "Model checking of real-time reachability properties using abstractions". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998, pp. 313–329.

[Dut14] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification*. Springer. 2014, pp. 737–744.

[DY96] Conrado Daws and Sergio Yovine. "Reducing the number of clock variables of timed automata". In: *Real-Time Systems Symposium, 1996., 17th IEEE*. IEEE. 1996, pp. 73–81.

[EK00] E Allen Emerson and Vineet Kahlon. "Reducing model checking of the many to the few". In: *Automated Deduction-CADE-17*. Springer, 2000, pp. 236–254.

[EMB11] Niklas Een, Alan Mishchenko, and Robert Brayton. "Efficient implementation of property directed reachability". In: *Formal Methods in Computer-Aided Design (FMCAD), 2011*. IEEE. 2011, pp. 125–134.

[EN95] E Allen Emerson and Kedar S Namjoshi. "Reasoning about rings". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1995, pp. 85–94.

[ES05] Niklas Een and Niklas Sörensson. "MiniSat: A SAT solver with conflict-clause minimization". In: *Sat* 5 (2005).

[ES96] E Allen Emerson and A Prasad Sistla. "Symmetry and model checking". In: *Formal methods in system design* 9.1-2 (1996), pp. 105–131.

[ES97] E. Allen Emerson and A. Prasad Sistla. "Utilizing symmetry when model-checking under fairness assumptions: an automata-theoretic approach". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.4 (1997), pp. 617–638.

[Fel+14] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. "Automating regression verification". In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM. 2014, pp. 349–360.

[Fel04] Joachim Feld. "PROFINET-scalable factory communication for all applications". In: *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*. IEEE. 2004, pp. 33–38.

[Flo62] Robert W Floyd. "Algorithm 97: shortest path". In: *Communications of the ACM* 5.6 (1962), p. 345.

[Göl+94] Aleks Göllü, Aleks Gollu, Anuj Puri, and Pravin Varaiya. "Discretization of timed automata". In: *in: Proceedings of the 33rd IEEE conferene on decision and control*. Citeseer. 1994.

[GS13] Benny Godlin and Ofer Strichman. "Regression verification: proving the equivalence of similar programs". In: *Software Testing, Verification and Reliability* 23.3 (2013), pp. 241–258.

[GS97] Viktor Gyuris and A Prasad Sistla. "On-the-fly model checking under fairness that exploits symmetry". In: *Computer Aided Verification*. Springer. 1997, pp. 232–243.

[Hal93] Nicolas Halbwachs. "Delay analysis in synchronous programs". In: *Computer Aided Verification*. Springer. 1993, pp. 333–346.

[Hav+97] Klaus Havelund, Arne Skou, Kim Guldstrand Larsen, and Kristian Lund. "Formal modeling and analysis of an audio/video protocol: An industrial case study using UPPAAL". In: *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*. IEEE. 1997, pp. 2–13.

[HB12] Kryštof Hoder and Nikolaj Bjørner. "Generalized property directed reachability". In: *Theory and Applications of Satisfiability Testing–SAT 2012*. Springer, 2012, pp. 157–171.

[HBS12] Zyad Hassan, Aaron R Bradley, and Fabio Somenzi. "Incremental, inductive CTL model checking". In: *Computer Aided Verification*. Springer. 2012, pp. 532–547.

[HBS13] Zyad Hassan, Aaron R Bradley, and Fabio Somenzi. "Better generalization in IC3". In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE. 2013, pp. 157–164.

[Hen+03] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco AA Sanvido. "Extreme model checking". In: *Verification: Theory and Practice*. Springer, 2003, pp. 332–358.

[Hen+04] Martijn Hendriks, Gerd Behrmann, Kim Larsen, Peter Niebert, and Frits Vaandrager. *Adding symmetry reduction to uppaal*. Springer, 2004.

[Hen00] Thomas A Henzinger. *The theory of hybrid automata*. Springer, 2000.

[HHH10] Christian Heinzemann, Stefan Henkler, and Martin Hirsch. *Refinement checking of self-adaptive embedded component architectures*. Tech. rep. Technical Report tr-ri-10-313, University of Paderborn, 2010.

[Hoc] Hochschule Ostwestfalen-Lippe. *Hochschule Ostwestfalen-Lippe - University of Applied Science*. URL: `https://www.hs-owl.de/` (visited on 12/15/2015).

[ID96] C Norris Ip and David L Dill. "Better verification through symmetry". In: *Formal methods in system design* 9.1-2 (1996), pp. 41–75.

[inI] inIT: Lemgoer Modellfabrik. *inIT: Lemgoer Modellfabrik*. URL: `https://www.hs-owl.de/init/en/smart-factory/lmf.html` (visited on 12/15/2015).

[Ise15] Tobias Isenberg. "Incremental Inductive Verification of Parameterized Timed Systems". In: *Application of Concurrency to System Design (ACSD), 2015 15th International Conference on*. IEEE. 2015, pp. 1–9.

[IW14] Tobias Isenberg and Heike Wehrheim. "Timed Automata Verification via IC3 with Zones". In: *Formal Methods and Software Engineering*. Springer, 2014, pp. 203–218.

[JM12] Taylor T Johnson and Sayan Mitra. "A Small Model Theorem for Rectangular Hybrid Automata Networks." In: *FMOODS/FORTE*. Springer. 2012, pp. 18–34.

[JN12] JÜrgen Jasperneite and Oliver Niggemann. "Systemkomplexität in der Automation beherrschen". In: *atp edition-Automatisierungstechnische Praxis* 54.09 (2012), pp. 36–45.

[Kes+97] Y Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. "Symbolic model checking with rich assertional languages". In: *Computer Aided Verification*. Springer. 1997, pp. 424–435.

[KJN12a] Roland Kindermann, Tommi Junttila, and Ilkka Niemelä. "Beyond lassos: Complete SMT-based bounded model checking for timed automata". In: *Formal Techniques for Distributed Systems*. Springer, 2012, pp. 84–100.

[KJN12b] Roland Kindermann, Tommi Junttila, and Ilkka Niemelä. "SMT-based induction methods for timed systems". In: *Formal Modeling and Analysis of Timed Systems*. Springer, 2012, pp. 171–187.

[Klo+13] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. "Incremental, inductive coverability". In: *Computer Aided Verification*. Springer. 2013, pp. 158–173.

[KM89] Robert P Kurshan and Ken McMillan. "A structural induction theorem for processes". In: *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*. ACM. 1989, pp. 239–247.

[Lam87] Leslie Lamport. "A fast mutual exclusion algorithm". In: *ACM Transactions on Computer Systems (TOCS)* 5.1 (1987), pp. 1–11.

[Lar+97] Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "Efficient verification of real-time systems: compact data structure and state-space reduction". In: *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*. IEEE. 1997, pp. 14–24.

[Lar+98] Kim G Larsen, Carsten Weise, Wang Yi, and Justin Pearson. "Clock difference diagrams". In: *BRICS Report Series* 5.46 (1998).

[Lee59] Chang-Yeong Lee. "Representation of Switching Circuits by Binary-Decision Programs". In: *Bell System Technical Journal* 38.4 (1959), pp. 985–999.

[Leo12] Leonardo de Moura. *Z3 for Java*. 2012. URL: `https://leodemoura.github.io/blog/2012/12/10/z3-for-java.html` (visited on 12/15/2015).

[LHR97] David Lesens, Nicolas Halbwachs, and Pascal Raymond. "Automatic verification of parameterized linear networks of processes". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1997, pp. 346–357.

[LP10] Daniel Le Berre and Anne Parrain. "The Sat4j library, release 2.2". In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 59–64.

[LPY95] Kim G Larsen, Paul Pettersson, and Wang Yi. "Model-checking for real-time systems". In: *Fundamentals of computation theory*. Springer. 1995, pp. 62–88.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. "Uppaal in a Nutshell". In: *International Journal on Software Tools for Technology Transfer (STTT)* 1.1 (1997), pp. 134–152.

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. "Formal design and analysis of a gear controller". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998, pp. 281–297.

[Mah05] Pritha Mahata. "Model checking parameterized timed systems". In: (2005).

[Mai14] Alexander Maier. "Identification of Timed Behavior Models for Diagnosis in Production Systems". PhD thesis. University of Paderborn, 2014.

[McM02] Ken L McMillan. "Applying SAT methods in unbounded symbolic model checking". In: *Computer Aided Verification*. Springer. 2002, pp. 250–264.

[McM03] Kenneth L McMillan. "Interpolation and SAT-based model checking". In: *Computer Aided Verification*. Springer. 2003, pp. 1–13.

[McM05] Kenneth L McMillan. "Applications of Craig interpolants in model checking". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 1–12.

[MGS13] Andreas Morgenstern, Manuel Gesell, and Klaus Schneider. "Solving games using incremental induction". In: *Integrated Formal Methods*. Springer. 2013, pp. 177–191.

[Mil80] Robin Milner. "A calculus of communicating systems". In: (1980).

[Mil99] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[Min67] Marvin L Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.

[Mur89] Tadao Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580.

[Nec02] George C Necula. *Proof-carrying code. design and implementation*. Springer, 2002.

[Nig+12] Oliver Niggemann, Benno Stein, Asmir Vodencarevic, Alexander Maier, and Hans Kleine Büning. "Learning Behavior Models for Hybrid Timed Systems." In: *AAAI*. Vol. 2. 2012, pp. 1083–1090.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t)". In: *Journal of the ACM (JACM)* 53.6 (2006), pp. 937–977.

[PAT] PAT. *Benchmarks of Timed Automata*. URL: `http://www.comp.nus.edu.sg/~pat/bddlib/timedexp.html` (visited on 12/15/2015).

[PEM11] Hans-Jörg Peter, Rüdiger Ehlers, and Robert Mattmüller. "Synthia: Verification and Synthesis for Timed Automata". In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer-Verlag, 2011, pp. 649–655.

[PG86] David A Plaisted and Steven Greenbaum. "A structure-preserving clause form translation". In: *Journal of Symbolic Computation* 2.3 (1986), pp. 293–304.

[PRZ01] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. "Automatic deductive verification with invisible invariants". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2001, pp. 82–97.

[PWZ02] Wojciech Penczek, Bożena Woźna, and Andrzej Zbrzezny. "Towards bounded model checking for the universal fragment of TCTL". In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer. 2002, pp. 265–288.

[Ram73] Chander Ramchandani. "Analysis of asynchronous concurrent systems by timed petri nets." PhD thesis. Massachusetts Institute of Technology, 1973.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[RR86] George M Reed and A William Roscoe. "A timed model for communicating sequential processes". In: *Automata, Languages and Programming*. Springer, 1986, pp. 314–323.

[SA92] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[SB11] Fabio Somenzi and Aaron R Bradley. "IC3: where monolithic and incremental meet." In: *FMCAD*. 2011, pp. 3–8.

[SFS12] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. "Incremental upgrade checking by means of interpolation-based function summaries". In: *Formal Methods in Computer-Aided Design (FMCAD), 2012*. IEEE. 2012, pp. 114–121.

[Sor03] Maria Sorea. "Bounded model checking for timed automata". In: *Electronic Notes in Theoretical Computer Science* 68.5 (2003), pp. 116–134.

[SS94] Oleg V Sokolsky and Scott A Smolka. "Incremental model checking in the modal mu-calculus". In: *Computer Aided Verification*. Springer. 1994, pp. 351–363.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking safety properties using induction and a SAT-solver". In: *Formal Methods in Computer-Aided Design*. Springer. 2000, pp. 127–144.

[Sud13] Martin Suda. "Triggered clause pushing for IC3". In: *arXiv preprint arXiv:1307.4966* (2013).

[Sud14] Martin Suda. "Property directed reachability for automated planning". In: *Journal of Artificial Intelligence Research* 50.1 (2014), pp. 265–319.

[Sun+09] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. "PAT: Towards Flexible Verification under Fairness". In: *Proceedings of the 21rd International Conference on Computer Aided Verification (CAV)*. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 709–714.

[Tse83] Grigori S Tseitin. "On the complexity of derivation in propositional calculus". In: *Automation of reasoning*. Springer, 1983, pp. 466–483.

[TY96] Stavros Tripakis and Sergio Yovine. "Analysis of timed systems based on time-abstracting bisimulations". In: *Computer Aided Verification*. Springer. 1996, pp. 232–243.

[UPP] UPPAAL. *Run-Time Data for Uppaal*. URL: `https://www.it.uu.se/research/group/darts/uppaal/benchmarks/` (visited on 12/15/2015).

[VGD12] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. "Green: reducing, reusing and recycling constraints in program analysis". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012, p. 58.

[Wan00] Farn Wang. "Region encoding diagram for fully symbolic verification of real-time systems". In: *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*. IEEE. 2000, pp. 509–515.

[Wan01a] Farn Wang. *Clock Restiction Diagram: Yet Another Data-Structure for Fully Symbolic Verification of Timed Automata*. Tech. rep. TRIIS-01-002, Institute of Information Science, Academia Sinica, Tapei, Taiwan, 2001. 83, 2001.

[Wan01b] Farn Wang. "RED: Model-checker for timed automata with clock-restriction diagram". In: *Workshop on Real-Time Tools, Aug*. 2001, pp. 2001–014.

[Wan02] Farn Wang. "Symbolic verification of complex real-time systems with clock-restriction diagram". In: *Formal Techniques for Networked and Distributed Systems*. Springer. 2002, pp. 235–250.

[Wan03] Farn Wang. "Efficient verification of timed automata with BDD-like data-structures". In: *Verification, model checking, and abstract interpretation*. Springer. 2003, pp. 189–205.

[Wil99] HX Willems. "Compact timed automata for PLC programs". In: (1999).

[WL90] Pierre Wolper and Vinciane Lovinfosse. "Verifying properties of large sets of processes with network invariants". In: *Automatic Verification Methods for Finite State Systems*. Springer. 1990, pp. 68–80.

[Yov98] Sergio Yovine. "Model checking timed automata". In: *Lectures on Embedded Systems*. Springer, 1998, pp. 114–152.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. "Automatic verification of real-time communicating systems by constraint-solving." In: *FORTE*. Vol. 6. Citeseer. 1994, pp. 243–258.